A Unified Framework for Efficiently Processing Ranking Related Queries

Muhammad Aamir Cheema[‡†], Zhitao Shen[◇†], Xuemin Lin[†§], Wenjie Zhang[†]
[‡]*Clayton School of Information Technology, Monash University, Australia*
[†]*School of Computer Science and Engineering, The University of New South Wales, Australia*
[◇]*Cisco China Research and Development Center*
[§]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China*
aamir.cheema@monash.edu, {shenz, lxue, zhangw}@cse.unsw.edu.au

ABSTRACT

The computation of k-lower envelope is a classical problem and has been very well studied for main memory non-indexed data. In this paper, we study the problem from the database perspective and present the first algorithm which utilizes the presence of the index and achieves access optimality, i.e., it accesses a node of the index only if the correctness of the results cannot be guaranteed without accessing this node. We also demonstrate the applications of k-lower envelope in ranking systems. Let an object be called *valuable* if it is one of the top-k objects according to at least one linear scoring function. In this paper, we answer the following important questions that may be asked by different users: *1) I am not sure what scoring function I should use, therefore, return me the set of valuable objects so that I can select an object I like the most; 2) How can I modify the attributes (e.g., price) of my product such that it becomes a valuable object; 3) What are the preference functions for which a given object is among the top-k objects*. These three questions are formalized and called k-snippet, k-depth contour and reverse top-k query, respectively. We propose a unified framework to solve these queries by utilizing *k-lower envelope* as a common foundation. Our main algorithm is access optimal for k-snippet and k-lower envelope computation. We also demonstrate its access optimality for the k-depth contour problem when k is smaller than the minimum number of objects in any leaf node of the index structure. Our algorithms outperform state-of-the-art algorithms by more than an order of magnitude in terms of both CPU and I/O cost.

1. INTRODUCTION

Consider a set of lines $\mathcal{L}$. Lower score of a point p is the number of lines that lie strictly below p. k-lower envelope is the closure of the set of points that have lower scores equal to $k-1$. In Fig. 1, assume that $\mathcal{L}$ consists of the five lines a^* to e^*. The lower score of point p is 1. The 2-lower envelope is shown using bold lines.

k-lower envelope (also known as k-level arrangement) is a classical problem and its computation is a core component of a wide variety of algorithms (e.g., see [1] for a nice survey). In such algorithms, the given set of points O is mapped to a set of lines $\mathcal{L}$ using a dual space mapping [2] (details to be provided in Section 2). Then, a k-lower envelope is computed on $\mathcal{L}$ and its properties are exploited to solve the underlying question. The effectiveness of k-lower envelope has been established by utilizing it in a wide range of applications such as in computing kth-order Voronoi diagrams [3, 4, 5], designing data structures for half-space range searching [6, 7], processing ranking queries [8, 9, 10] and solving hyper-plane partitioning problems such as sandwich cuts [11] and weak line-separators [12], to name a few.

Although the efficient computation of k-lower envelope has received significant research attention, all of the existing algorithms are designed based on the following two assumptions: 1) the algorithms are *index agnostic* and are designed assuming non-indexed data, i.e., these algorithms do not exploit the availability of the prebuilt indexes; 2) the algorithms assume that the data sets reside in the main memory. These two assumptions seriously affect the suitability of the existing algorithms for the scenarios where the data set is indexed on disk. Note that even if sufficient main memory is available, the existing algorithms need to access the whole data set at least once in order to load it in the main memory. Furthermore, the extension of the existing algorithms for the indexed data is either non-trivial or inefficient as demonstrated by our experimental study.

In this paper, we study the problem of k-lower envelope computation from database perspective. More specifically, we assume that the set of objects O is disk-resident and is indexed by a branch-and-bound data structure (e.g., R-tree, Quad-tree etc.). We develop two CPU and I/O efficient *index-aware* algorithms namely SkyRider and KnightRider. Our main algorithm, KnightRider, introduces a *must-first* paradigm and is access optimal, i.e., no algorithm can guarantee the correctness of the results unless it accesses at least all the nodes that are accessed by our algorithm. In the must-first paradigm, in each round, the algorithm identifies one or more nodes of the index that must be accessed in order to guarantee the correctness. Then, the algorithm accesses all such entries and proceeds to the next round. In any round, if the algorithm does not find any such entry, the algorithm terminates by returning the correct results.

Like most of the existing work on k-lower envelope, the focus of this paper is on 2-dimensional data sets. Nevertheless, in Section 4.3, we demonstrate that our claim of access optimality and the pruning rules hold even for higher dimensionality[1]. In the past,

[1]We remark that the lower bound computational complexity of k-lower envelope increases exponentially with dimensionality [13]. Hence, there is a need for approximate but efficient algorithms for d-dimensional data sets and we intend to study this in an extended version.

the computation of 2-dimensional k-lower envelope has been extensively studied because it is not only of stand-alone interest but is also used in a variety of other algorithms. For instance, the database community has utilized 2-dimensional k-lower envelope for answering various ranking queries [8, 9, 10]. Next, we present a few other novel ranking questions that can be efficiently answered using k-lower envelope.

1.1 Representative Applications

Given a linear scoring function f and a set of objects O, a top-k query returns k objects from O that have the smallest scores where score of each object is computed using f. A linear scoring function can be represented as a weighting vector $\overrightarrow{w} = (w_1, w_2)$. The score of a point $p = (p_1, p_2)$ with respect to $\overrightarrow{w}$ is $w_1 p_1 + w_2 p_2$. For the ease of presentation, in this section, we restrict our discussion to the case when only non-negative weights are allowed. Later in Section 2, we demonstrate that our techniques can also handle the case when both negative and positive weights are allowed.

PROBLEM 1 : **k-snippet**. Given a set of objects O, k-snippet is a set consisting of every object $o \in O$ such that o is among the top-k objects for *at least* one linear scoring function.

Consider the example of Fig. 2 where O consists of 10 objects a to j. 1-snippet is $\{a, b\}$ because no other object can be the top-1 object no matter what linear scoring function is used. 2-snippet is $\{a, b, c, d\}$. Note that k-snippet is different from *k-skyband* [14]. In Fig. 2, 1-skyband (i.e., skyline) is $\{a, h, c, b\}$ whereas 1-snippet is $\{a, b\}$. Section 5.4 provides more details about the relationship between k-snippet and k-skyband.

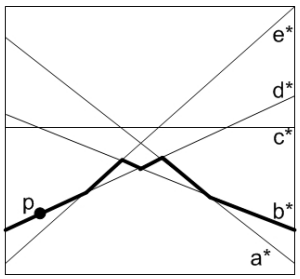


Figure 1: k-lower envelope ($k = 2$)

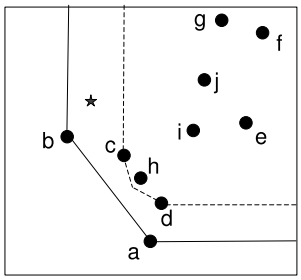


Figure 2: k-snippet, k-depth contour and reverse top-k

Applications: k-snippet serves as a data summarization tool that returns only the objects that may be important for the users or the ranking system. Consider a user who wants to retrieve an object of his choice from a set of objects. He may not be willing/able to define suitable scoring functions due to various reasons such as lack of knowledge about the data domain or due to incompatible attributes on each dimension (e.g., dollars vs inches) [15]. The system may help such users by returning the smallest subset of objects that guarantees that no matter what linear scoring function is used the top-k objects are present in this subset. The users may scan this subset of objects to choose a suitable object. k-snippet is also important for ranking systems because every top-m ($m \leq k$) query (involving linear scoring function) can be answered using k-snippet instead of accessing the whole database (e.g., see [16]).

PROBLEM 2 : **k-depth contour.** Given a set of objects O, k-depth contour is a region such that i) every object p (not necessarily in O) that lies outside it or on its boundary is in the k-snippet of $\{p \cup O\}$ and ii) every object p' (not necessarily in O) that lies inside this region is not in the k-snippet of $\{p' \cup O\}$.

Consider the example of Fig. 2 where 1-depth contour is the region shown using solid lines and encloses all of the objects. 2-depth contour is the region shown using broken lines and contains all objects except a and b. Consider an object q which is not in the data set O (i.e., the star in Fig. 2). q is outside 2-depth contour and is one of the top-2 objects for at least one scoring function, e.g., for the weighting vector $\overrightarrow{w} = (1, 0)$, only b has a better score than q. Hence, q is in 2-snippet. Furthermore, q is inside 1-depth contour and it can be confirmed that q is not the top-1 object for any linear scoring function, i.e., q is not in 1-snippet. Note that the vertices of k-depth contour are not necessarily the data objects (e.g., see 2-depth contour in Fig. 2).

Applications. Consider that the user of the ranking system is a businessman who wants the system to help him in designing a competitive product. More specifically, the businessman wants to make sure that his product is among the top-k products for at least one linear scoring function. The system may give him suggestions on how to set the specifications of the product to achieve this goal, i.e., the system can return him k-depth contour that enables him to visualize all possible settings that can improve his product. For instance, in Fig. 2, if the businessman wants his product q (shown as a star) to be the top-1 product for at least one scoring function, he must change the attributes of q such that it lies outside 1-depth contour, e.g., if x-coordinate corresponds to the price, the businessman may reduce the price of q to ensure that q is outside 1-depth contour and hence is a top-1 object for at least one scoring function. k-depth contour may also help if the businessman wants to increase the price of the product q but wants to make sure that it remains one of the top-2 products for at least one scoring function. He may increase the price of q ensuring that q remains outside the 2-depth contour.

k-depth contour is also known as k-hull [17] in computational geometry and is a well studied problem. The k-depth contour on 2d data sets has many applications such as in outlier detection [18, 19, 20], regression analysis [21], clustering [22] and data visualization [23]. Specifically, it is a robust tool for data picturization [23] and can be used for a visual representation of location, spread, correlation, skewness and tails of data [24]. The k-depth contour has also been used for outlier detection [18] and has a nice feature that it does not rely on the probability distribution of the underlying data set. Ruts *et al.* [22] demonstrated the applications of k-depth contours in clustering and discriminant analysis.

PROBLEM 3 : **Reverse top-k query** [25]. Given a set of objects O and a query object q (not necessarily in O), find every linear scoring function for which q is one of the top-k objects in $\{q \cup O\}$.

Consider the example of Fig. 2 and assume that the query object is b and $k = 1$. Note that b is the top-1 object when weighting vector is $\overrightarrow{w} = (1, 0)$. In fact, b is the top-1 object when the weight of first attribute w_1 (i.e., x-coordinate) is within range 0.5 to 1 (the weight w_2 changes accordingly assuming that the sum of both weights is 1). Hence, the answer of the reverse top-1 query is the range of scoring functions where w_1 varies from 0.5 to 1.

Applications. Consider the example of a businessman who wants to analyze the impact of his product in the presence of other products in the market. The reverse top-k query returns him the scoring functions for which his product is one of the top-k products so that he can target the users that have similar preferences. For instance, in Fig. 2, the product b is the top-1 product when the weight of first attribute varies from 0.5 to 1. Assume that the first attribute corresponds to the price. This means that the product b is the best product for the users who give price more importance than the other

attribute. Hence, the businessman may target such users and advertise his product by highlighting its low price.

The reverse top-k query can also be used to mine user preferences. For instance, a user that selects a product $o \in O$ is likely to have preferences similar to the scoring functions for which o is among the top-k products. Hence, even when the users do not specify any scoring function, the system may learn about their preferences based on the objects they select. For a more detailed description of the applications of reverse top-k queries, see [25, 8].

1.2 Contributions

Below are the details of our contributions for each problem.

k-lower envelope. We are the first to propose an access optimal algorithm for computing k-lower envelope on disk-resident data sets. Based on an intuitive preliminary algorithm (called Rider), we develop two CPU and I/O efficient algorithms namely SkyRider and KnightRider. KnightRider is proven to be access optimal, i.e., it only accesses the nodes that must be accessed in order to guarantee the correctness. While KnightRider outperforms SkyRider in terms of I/O cost, SkyRider is slightly more efficient than KnightRider in terms of CPU cost. This is because, in order to achieve access optimality, KnightRider needs to spend more computational efforts to prune the unnecessary nodes. Both algorithms outperform the existing algorithms by more than an order of magnitude in terms of both I/O and CPU cost.

k-snippet. To the best of our knowledge, we are the first to study the problem of k-snippet that serves as a data summarization tool for queries involving linear scoring functions. Our main algorithm, KnightRider, is access optimal for this problem.

k-depth contour. We are the first to propose an index-aware algorithm for k-depth contour on disk-resident data. One of our proposed algorithms is access optimal for the case when k is smaller than the minimum number of children in any leaf node of the R-tree. Although we are unable to claim the optimality for larger values of k, our experimental results show that the I/O cost of our algorithm is almost the same as the lower bound I/O cost even when k is very large, i.e., $k = 100,000$. Our proposed algorithms perform up to three orders of magnitude better than the existing algorithms.

Reverse top-k query. Our algorithm is up to two orders of magnitude faster than the state-of-the-art algorithm [25]. Furthermore, once the k-lower envelope has been computed, the cost of answering the reverse top-k query is very low. Hence, if multiple reverse top-k queries are issued, our algorithms can answer all queries with the total cost quite close to the cost of answering a single query because k-lower envelope is required to be computed only once.

The rest of the paper is organized as follows. Section 2 demonstrates how to answer various ranking related queries using k-lower envelope. Our first algorithm called SkyRider is presented in Section 3. Our main algorithm, KnightRider, is presented in Section 4. In Section 5, we present our experimental study. The related work is presented in Section 6 followed by conclusion in Section 7.

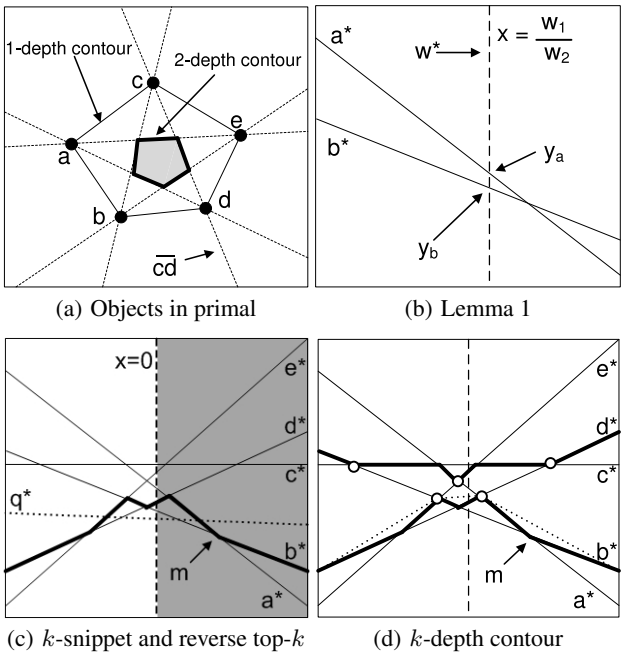
2. PRELIMINARIES

In this section, we show that k-snippet, reverse top-k queries and k-depth contour can be solved using k-lower envelope and *k-upper envelope*.

DEFINITION 1 : **k-lower (resp. upper) envelope**. Consider a set of lines $\mathcal{L}$. Lower (resp. upper) score of a point p is the number of lines below (resp. above) p. k-lower (resp. upper) envelope is the closure of the set of points that have lower (resp. upper) score equal to $k-1$.

Fig. 3(d) shows 2-lower envelope and 2-upper envelope using bold lines. Next, we briefly describe the concept of dual mapping.

Dual Mapping. A point $p = (u, v)$ in primal space is mapped to a line $p^* : y = ux + v$ in the dual space and a line $L : y = -ux + v$ in the primal space is mapped to a point $L^* = (u, v)$ in the dual space (e.g., see [2]). The transformation from/to primal to/from dual is denoted by using a superscript *, e.g., a point p in primal is denoted as p^* in dual and a line L in primal is denoted as L^* in dual. The weighting vector $\overrightarrow{w} = (w_1, w_2)$ is mapped to a vertical line $x = \frac{w_1}{w_2}$. A weighting vector $\overrightarrow{w}$ is denoted as w^* in dual. Fig. 3(a) shows 5 objects a to e in primal space and Fig. 3(c) shows their corresponding dual lines a^* to e^* in dual space.



(a) Objects in primal (b) Lemma 1

(c) k-snippet and reverse top-k (d) k-depth contour

Figure 3: Solving the queries in dual space

Consider two points $a = (a_1, a_2)$ and $b = (b_1, b_2)$ in the primal space. Given a weighting vector $\overrightarrow{w} = (w_1, w_2)$, the score of a is $a.score = w_1a_1 + w_2a_2$. Similarly, $b.score = w_1b_1 + w_2b_2$. The two points a and b are mapped to dual space as $a^* : y = a_1x + a_2$ and $b^* : y = b_1x + b_2$ (see a^* and b^* in Fig. 3(b)). The weighting vector is mapped as a vertical line (see the broken line in Fig. 3(b)). Let the y-coordinate value of the point where a^* (resp. b^*) meets the vertical line w^* be called y_a (resp. y_b). The following lemma shows that the relative ranking of a an b w.r.t. the weighting vector $\overrightarrow{w}$ can be determined by comparing y_a and y_b.

LEMMA 1 : Consider a weighting vector $\overrightarrow{w} = (w_1, w_2)$. When $w_2 > 0$, $a.score < b.score$ iff $y_a < y_b$. When $w_2 < 0$, $a.score < b.score$ iff $y_a > y_b$.

PROOF. Since y_a is the point where $a^* : y = a_1x + a_2$ intersects $w^* : x = w_1/w_2$, the value of y_a can be determined by replacing x in the equation of a^* by w_1/w_2, i.e., $y_a = a_1\frac{w_1}{w_2} + a_2$ or $y_a = \frac{1}{w_2}(a_1w_1 + a_2w_2)$. Since $a.score = a_1w_1 + a_2w_2$, $y_a = \frac{a.score}{w_2}$. Similarly, $y_b = \frac{b.score}{w_2}$. Hence, it is immediate that, when $w_2 > 0$, $a.score < b.score$ iff $y_a < y_b$. Also, it is obvious that when $w_2 < 0$, $a.score < b.score$ iff $y_a > y_b$. □

Consider the example of Fig. 3(b) and assume that $\overrightarrow{w} = (0.2, 0.8)$. The score of b is better (i.e., smaller) than a because b^* intersects

w^* below the point where a^* intersects w^*. On the other hand, if $\overrightarrow{w} = (-0.2, -0.8)$ then a has a better rank because $y_a > y_b$. We remark that the case when $w_2 = 0$ can be easily handled by treating w_2 as having a positive but infinitely small value.

LEMMA 2 : Consider a weighting vector $\overrightarrow{w} = (w_1, w_2)$. When $w_2 > 0$, top-k objects correspond to the k-lowest lines intersecting w^* in the dual space. When $w_2 < 0$, top-k objects correspond to the k-highest lines intersecting w^*.

The proof immediately follows from Lemma 1. Consider the example of Fig. 3(a), where 5 objects a to e are shown. In Fig. 3(c), these objects are mapped to lines a^* to e^* in dual space (ignore the dotted line q^* for this example). Consider the weighting vector $\overrightarrow{w} = (0, 1)$ (w^* is shown as a vertical broken line in Fig. 3(c)). The top-2 objects are b and d because b^* and d^* are the two lowest lines intersecting w^*. If the weighting vector $\overrightarrow{w} = (0, -1)$ then the top-2 objects are c and e because c^* and e^* are the two highest lines intersecting w^*.

2.1 Computing k-snippet in dual space

First, we assume that only non-negative weights are allowed. The shaded area in Fig. 3(c) corresponds to the space for which $x \geq 0$ and is called the positive dual space. The white area is called the negative dual space. Note that if the weighting vector is allowed to have only non-negative weights then every weighting vector $\overrightarrow{w}$ has its dual w^* in the positive dual space. Next two lemmas show how to compute k-snippet in dual space. The proofs are omitted but can be easily obtained by using Lemma 2.

LEMMA 3 : If only non-negative weights are allowed, an object $o \in O$ is an object in k-snippet if and only if at least one point of o^* lies on or below the k-lower envelope in the positive dual space.

In Fig. 3(c), 2-snippet is $\{a, b, d\}$ because each of a^*, b^* and d^* have at least one point on or below the 2-lower envelope in the positive dual space (the shaded area).

LEMMA 4 : If both positive and negative weights are allowed then an object $o \in O$ is in k-snippet if and only if at least one point p of o^* satisfies one of the following two conditions: 1) p lies on or below k-lower envelope; 2) p lies on or above k-upper envelope.

In Fig. 3(d), 2-snippet consists of all the objects because each line contains at least one point that satisfies at least one of the two conditions.

2.2 Computing reverse top-k in dual space

Consider a query point q that is mapped to a line q^* in dual space as shown in Fig. 3(c). Assume that we consider only non-negative weights. According to Lemma 2, in Fig. 3(c), q^* is among the top-k objects for every weighting vector w^* lying in the positive dual space where q^* (the dotted line) lies below k-lower envelope. Hence, a reverse top-k query can be answered by computing the intersection of q^* with k-lower envelope in the positive dual space.

If both the positive and negative weights are allowed then the reverse top-k query can be answered by computing the segments of q^* that lie below k-lower envelope and the segments of q^* that lie above k-upper envelope.

2.3 Computing k-depth contour in dual space

It has been shown that k-depth contour can be computed using the k-lower and k-upper envelopes [20]. Before we describe the procedure, we introduce a few concepts.

DEFINITION 2 : **Upper (lower) hull**. Let Z be the convex hull of a set of points P. The Upper (resp. lower) hull of P is the set of edges of Z that lie on or above (resp. on or below) every point $p \in P$.

In Fig. 3(a), convex hull of the set of objects a to e is the outer polygon. The upper hull consists of the edges ac and ce and the lower hull is the set of edges ab, bd and de. In Fig. 3(d), the upper hull of the points in k-lower envelope is shown using dotted lines.

DEFINITION 3 : **Convex vertices**. Let UH be the upper hull of the points on k-lower envelope and LH be the lower hull of the points on k-upper envelope. The vertices of UH and LH are called the convex vertices.

Fig. 3(d) shows all the convex vertices as hollow circles. Now, we describe how to compute k-depth contour in the dual space. For a more detailed description, see [20].

1. Map all objects in O to lines in a dual space.

2. Compute k-lower envelope and k-upper envelope of these lines and determine the convex vertices (by computing upper and lower hulls).

3. Map the convex vertices to lines in primal space and use these lines to obtain the k-depth contour. In Fig. 3(a), the dotted lines correspond to the convex vertices of Fig. 3(d). For instance, the vertex in Fig. 3(d) where c^* and d^* intersect each other corresponds to the line $\overline{cd}$ in Fig. 3(a). The shaded area in Fig. 3(a) is 2-depth contour that is obtained by using the dotted lines.

When only the non-negative weights are allowed, the computation of the k-depth contour is exactly the same except that only the k-lower envelope and only the positive dual space is considered.

3. THE SKYRIDER ALGORITHM

In the rest of the paper, we present efficient algorithms to compute k-lower envelope. The computation of k-upper envelope is similar. Although our techniques and optimality claims hold for all branch-and-bound data structures, in the rest of the paper, we restrict our discussion to the case when objects are indexed by R-tree. Also, we present our techniques for the case when $k \leq n/2$ where n is total number of objects. This is because k-lower envelope is the same as $(n-k+1)$-upper envelope [2]. Hence, if $k > n/2$, the k-lower envelope can be obtained by computing k'-upper envelope where $(k' = n - k + 1) \leq n/2$.

Like most of the existing techniques (e.g., see [26]), we assume that no two lines are parallel and no three lines are concurrent when the objects are mapped to the dual space. We remark that this assumption is made only for the ease of presentation. Later in Section 4.3, we show that such situations can be handled easily.

3.1 The Rider: An Elementary Algorithm

First, we present a basic disk-based algorithm for computing k-lower envelope. This is called Rider algorithm and is also used as a subroutine in our main algorithms, SkyRider and KnightRider.

Intuitive description. Assume that all objects in O have been mapped to lines in a dual space. Let origin line L_o be the line with the k-th smallest slope. Let destination line L_d be the line with the k-th largest slope. Assume that all lines are roads and a bike rider starts traveling from the right most point on the origin line (i.e., at $x = \infty$). The rider always travels towards his left (i.e., towards decreasing value of x). Whenever he reaches at an intersection of two lines, he makes a turn. The rider keeps traveling until it reaches the left most point of the destination line (i.e., at $x = -\infty$). It is easy to verify that the path that the rider travels on corresponds to the

k-lower envelope. This is because when the rider starts at the right most point on the origin line, there are exactly $k-1$ lines above him and each turn ensures that he remains below exactly $k-1$ lines. The proof is straightforward and intuitive and is omitted.

In Fig. 3(c), assuming $k = 2$, the origin line is b^* and the destination line is d^*. The rider starts from the right most point of b^* and travels towards left. When he reaches at the intersection m, he makes a turn and continues traveling on a^*. The algorithm continues until the rider reaches the left most point of d^*. The path (shown in bold) is the path traveled by the rider and corresponds to k-lower envelope.

Algorithm 1 presents a more formal description.

Algorithm 1: Rider Algorithm

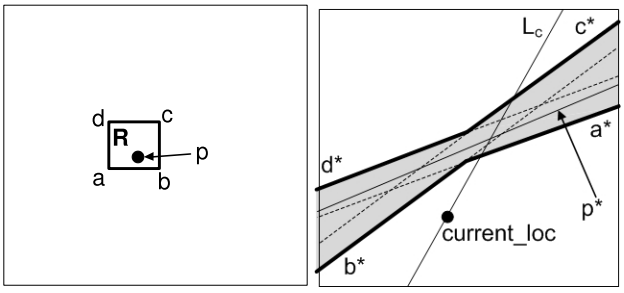
1 Find the origin line and call it L_c. Set the current location $current\_loc$ as the point on L_c with $x = \infty$;
2 Among the lines that intersect L_c on left of $current\_loc$, find the line L' that intersects L_c at the right most point. Let z be the intersection of L' and L_c. Add z to k-lower envelope;
3 Terminate the algorithm if there does not exist any such L' at line 2. Otherwise, set the current location as z (i.e., $current\_loc \leftarrow z$) and current line as L' (i.e., $L_c \leftarrow L'$) and go to line 2;

Implementing rider algorithm on disk-resident data. In this section, we show how to implement the rider algorithm when the data objects (i.e., the lines in dual space) are indexed by a disk-resident branch-and-bound data structure (i.e., R-tree). More specifically, we briefly describe how to implement the first two lines of the algorithm (the third line is self describing).

Line 1. Note that the origin line is the line with the k-th smallest slope and it corresponds to the object in primal space that has the k-th smallest x-coordinate value. Such an object can be easily found using a best-first search algorithm on R-tree.

Line 2. Before we present the details, we discuss how a rectangle in primal space is mapped to dual space.

Spectrum of a rectangle. Consider the rectangle R shown in Fig. 4(a). In Fig. 4(b), we map the four corners of the rectangle (a to d) to four lines in dual space (a^* to d^*). The 1-upper envelope and 1-lower envelope of these four lines are shown using bold lines. The space between the $1-$upper envelope and 1-lower envelope is called the *spectrum* of rectangle R and is denoted as R^*. In Fig. 4(b), the spectrum of R is shown shaded. It is easy to verify that, for any point $p \in R$, its corresponding line p^* in dual space lies entirely in R^*. In Fig. 4, the point p lies in the rectangle R and p^* lies in R^*.



(a) Primal Space (b) Dual Space

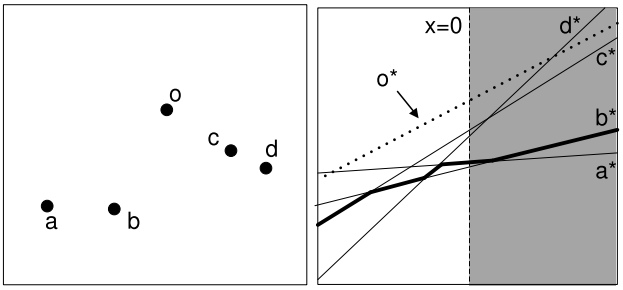
Figure 4: Mapping a rectangle to dual space

Consider a line L_c and a spectrum R^* as shown in Fig. 4. For every point $p \in R$, p^* intersects the line L_c on the segment of L_c that lies inside the spectrum. A branch-and-bound algorithm can be used to execute the line 2 of Algorithm 1. Since our main algorithm is KnightRider (presented in Section 4.2), we only give a high level idea of this algorithm. The algorithm iteratively processes the entries of the R-tree. For each accessed entry e, the algorithm checks the intersection of L_c with e^* and decides whether to prune e or insert its children for processing in next round. Specifically, an entry e can be pruned if the segment of L_c that lies within e^* lies on the right of $current\_loc$, i.e., no line in e^* can intersect L on the left of $current\_loc$ (as shown in Fig. 4(b)). Hence, e can be pruned.

3.2 The SkyRider: An I/O Efficient Rider

A major problem with the rider algorithm is that it traverses R-tree using a branch-and-bound algorithm as many times as the number of vertices of the k-lower envelope. This results in a very high I/O cost. Next, we present an observation that significantly reduces the I/O cost of the rider algorithm.

Consider the example of Fig. 5 where 5 objects are shown in primal space and dual space. Note that o^* does not contribute to the 2-lower envelope and can be pruned. Lemma 5 identifies the conditions that can be used to prune such objects. For a point p, we use p_x and p_y to denote the value of p in x and y coordinates, respectively.



(a) Primal Space (b) Dual Space

Figure 5: Pruning irrelevant data points

LEMMA 5 : An object o can be pruned if both of the following conditions hold: i) there exist at least k objects such that for each such object r, $r_x < o_x$ and $r_y < o_y$ (see objects a and b in Fig. 5(a)); and ii) there exist at least k objects such that for each such object s, $s_x > o_x$ and $s_y < o_y$ (see objects c and d in Fig. 5(a)).

PROOF. For an object o mapped to a line o^* in dual space, o_x corresponds to the slope of o^* and o_y corresponds to the y-intercept of o^*. If the first condition is satisfied then it implies that there are at least k lines that lie strictly below o^* in the positive dual space (the shaded area in Fig. 5(b)). This is because each such line r^* has y-intercept below the y-intercept of o^* and slope smaller than the slope of o^* (see objects a and b in Fig. 5(a) and lines a^* and b^* in Fig. 5(b)). Hence, o^* cannot be a part of the k-lower envelope in the positive dual space. Following the similar arguments, it can be shown that o^* cannot be a part of the k-lower envelope in the negative dual space (see objects c and d in Fig. 5(a) and c^* and d^* in Fig. 5(b)). □

Note that the conditions defined in Lemma 5 have similarity to the concept of *dominance* [14, 27]. An object o' dominates another object o if o' is preferable to o on every attribute. A k-skyband [14] consists of every object that is dominated by at most $k-1$ objects. Assume that the preference function f_1 prefers smaller values on both coordinates x and y. Then, an object o satisfies the first condition of Lemma 5 if it is dominated by at least k objects according to f_1 (i.e., in Fig. 5(a), o is dominated by a and b according

to f_1). In other words, o satisfies the first condition if it is not a k-skyband object according to preference function f_1. Similarly, assume that another preference function f_2 prefers smaller values on y-coordinate and larger values on x-coordinate. The object o satisfies the second condition of Lemma 5 if o is not a k-skyband according to f_2.

The above discussion implies that the objects that are k-skyband objects according to f_1 or f_2 are sufficient be used to correctly compute the k-lower envelope. Hence, SkyRider algorithm first computes these two k-skybands using BBS [14] that stores k-skyband objects in a main-memory R-tree which is then used by the rider algorithm (Algorithm 1).

4. THE KNIGHTRIDER ALGORITHM

4.1 The Algorithm

4.1.1 Outline

In this paper, we propose a *must-first* paradigm. An entry e is called a must-entry if it must be accessed in order to guarantee the correct results, i.e., no algorithm can guarantee the correctness of the results unless e is accessed. In each round, a must-first algorithm identifies and processes all must-entries and proceeds to the next round. Note that an optimal algorithm must produce the correct results and terminate when there does not exist any must-entry.

Note that an optimal best-first algorithm (e.g., [14]) is also a must-first algorithm because in such algorithms the best entry is always a must-entry. However, not all the best-first algorithms are must-first algorithms. There is another major difference between the must-first paradigm and the best-first paradigm. That is, the best-first algorithms access one entry at a time. In contrast, the must-first algorithm may open all the must-entries at the same time because all such entries are to be accessed anyway before the algorithm can terminate. Hence, the must-first algorithm allows bulk access of the entries. Next, we briefly outline our algorithm.

Initially, the root node of the R-tree is inserted in a queue. In each round, we identify the must-entries present in the queue and, for each must-entry, we insert its children (i.e., corresponding spectrums) in the queue. In each round, by using the spectrums in the queue, we compute two approximations of k-lower envelope named *best envelope* and *worst envelope* such that the k-lower envelope is guaranteed to lie on or above the best envelope and is guaranteed to lie on or below the worst envelope. The algorithm terminates by reporting the correct results when no must-entry is found in the queue.

Before we present the algorithmic details, we introduce the best and worst envelopes (Section 4.1.2) and show how to identify the must-entries (Section 4.1.3).

4.1.2 Best (worst) envelope

Top-layer of a spectrum R^* is the upper boundary of the spectrum and *bottom-layer* of a spectrum is the lower boundary of the spectrum. In Fig. 4(b), the top-layer of R^* is the upper boundary shown in bold and the bottom-layer is the lower boundary (also shown in bold). The cardinality of a spectrum R^* is the number of objects in the corresponding node R of the R-tree.

Assuming that the lower the k-lower envelope is the better it is, the best (resp. worst) envelope denotes the best (resp. worst) possible k-lower envelope. Next, we give a formal definition.

DEFINITION 4 : **Best k-lower envelope**. Assume a set of bottom-layers where each layer is assigned a number that denotes the cardinality of the corresponding spectrum. For a point p, *lower cardinality* (resp. *upper cardinality*) is the sum of the cardinalities of all bottom-layers that lie below (resp. above) it. Best k-lower envelope is the closure of the set of points that lie on bottom-layers, and have lower cardinality at most $k-1$ and upper cardinality at most $n-k$ where n is the total number of objects.

In Fig. 6, three spectrums R_1^* (the dotted spectrum), R_2^* (the shaded spectrum) and R_3^* (the spectrum with broken thin lines) are shown with cardinalities 3, 4 and 2, respectively. The lower cardinality of the point p is 0 and its upper cardinality is 6 because the bottom layers of R_2 and R_3 lie above p. Since the total number of objects n is 9, the point p is a point on best 2-lower envelope. The worst k-lower envelope is defined similarly with the only difference that top-layers are used instead of bottom-layers in Defintion 4. Unless specifically mentioned, hereafter we use the term best (resp. worst) envelope to refer to best (resp. worst) k-lower envelope. In Fig. 6, solid bold lines are used to show the best envelope and broken bold lines are used for the worst envelope ($k=2$).

LEMMA 6 : Every point p of k-lower envelope lies between the best and worst envelopes, i.e., p lies on or above the best envelope and lies on or below the worst envelope.

We omit the proof due to space limitation. However, we give an intuitive explanation. Note that every object $o \in R$ has its dual line o^* inside R^*. This implies that o^* is on or below the top-layer of R^* and on or above the bottom-layer of R^*. Hence, it is easy to verify that the k-lower envelope lies on or above the best envelope and lies on or below the worst envelope.

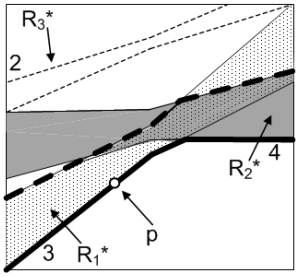


Figure 6: Best and worst k-lower envelopes

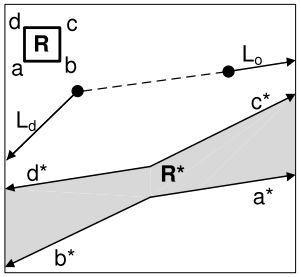


Figure 7: Illustration of Lemma 8

We remark that the best (resp. worst) envelopes can be easily computed using a slightly modified version of the rider algorithm (Algorithm 1) applied on the set of bottom (resp. top) layers. More specifically, at each intersection, the rider decides whether to make a turn or not based on which of the two lines satisfies the definition of best (resp. worst) envelope.

4.1.3 Identifying the must-entries

In this section, we define how to identify a must-entry, i.e., an entry that must be accessed in order to produce correct results.

DEFINITION 5 : **Underlap.** A spectrum R^* *underlaps* an envelope if at least one point of R^* lies on or below the envelope.

Note that if a spectrum R^* does not underlap an envelope it means that every point of R^* lies strictly above the envelope. In Fig. 6, the best envelope is shown using solid bold lines. R_1^* and R_2^* underlap the best envelope. On the other hand, R_3^* does not underlap the best envelope because each point of R_3^* lies strictly above the best envelope.

LEMMA 7 : An entry R is a must-entry if its spectrum R^* underlaps k-lower envelope.

PROOF. There may be two cases: i) R^* lies completely below k-lower envelope (i.e., every point of R^* lies strictly below k-lower envelope); ii) R^* does not lie completely below k-lower envelope. We show that the first case can never happen (Lemma 8). For the second case, we show that such R^* must be accessed. We prove this by contradiction. Assume that k-lower envelope can be computed without accessing such a rectangle R. Since R^* underlaps the k-lower envelope and satisfies the second case, there exists at least one point p of k-lower envelope that lies inside the spectrum R^*. Since p is a point inside R^*, its lower score depends on the locations of objects in R, i.e., for each object $o \in R$, o^* may or may not lie below p. Hence, the lower score of p cannot be computed unless such rectangle R is accessed, i.e., it cannot be determined whether p is a point on k-lower envelope or not unless R is accessed. □

LEMMA 8 : There does not exist any rectangle R such that every point of R^* lies strictly below k-lower envelope.

PROOF. As stated earlier in Section 3, two lines (origin line L_o and the destination line L_d) are always the lines on k-lower envelope. Origin line L_o has the k-th smallest slope and the destination line L_d has the k-th largest slope. Consider a rectangle R and its spectrum R^* (both shown in Fig. 7). For a line L, let $L.slope$ denote its slope. R^* cannot lie completely below k-lower envelope unless both of the following hold: i) c^* lies below L_o as c^* tends to ∞ (i.e., $c^*.slope \leq L_o.slope$); and ii) d^* lies below L_d as d^* tends to $-\infty$. (i.e., $d^*.slope \geq L_d.slope$).

We show that these two conditions cannot hold simultaneously. Without loss of generality, assume that the first condition holds, i.e., $c^*.slope \leq L_o.slope$. We prove by contradiction that the second condition cannot hold. Note that $L_o.slope \leq L_d.slope$ because L_o is the k-th smallest slope whereas L_d is the k-th largest slope (recall $k \leq n/2$ as stated in Section 3). This implies that $c^*.slope \leq L_d.slope$. This means that if the second condition holds then $c^*.slope \leq d^*.slope$. However, we know that $c^*.slope > d^*.slope$ because[2] c^* corresponds to the upper right corner of R in primal space and d^* corresponds to the lower right corner of R (see Fig. 7). Hence, the two conditions cannot hold simultaneously. □

Lemma 7 shows how to determine whether an entry R^* is a must entry or not by checking whether it underlaps k-lower envelope or not. However, the algorithm cannot employ Lemma 7 because k-lower envelope is not known. The next lemma resolves this issue by showing that a must entry can be identified by using the best envelope.

LEMMA 9 : An entry R is a must-entry if its spectrum R^* underlaps the best envelope.

PROOF. Recall that k-lower envelope is always on or above the best envelope (Lemma 6). Since R^* underlaps the best envelope, the spectrum R^* also underlaps the k-lower envelope. Hence, R^* is a must-entry. □

4.1.4 Pseudocode

Before we present the algorithm, we define a condition that prunes the entries of R-tree that are not required to compute k-lower envelope. Pruning these entries improves the efficiency of the algorithm mainly because fewer entries are to be considered to update the best and worst envelopes in each round.

[2]Note that the proof does not hold if $c^*.slope = d^*.slope = L_o.slope = L_d.slope$, i.e., R is not a rectangle but a vertical line and the four lines c^*, d^*, L_o and L_d are parallel which is against our assumption that no two lines are parallel. Later in Section 4.3, we show that such special cases can be easily handled.

PRUNING RULE 1 : An entry R can be pruned if its spectrum R^* does not underlap the worst envelope.

PROOF. Let p be a point on k-lower envelope. According to Lemma 6, p lies on or below the worst envelope. Since R^* does not underlap the worst envelope, for every object $o \in R$, o^* passes above p. Hence, o^* does not affect the lower score of any point p of k-lower envelope and can be pruned. □

In Fig. 6, the worst envelope is shown using bold broken lines. R_3 can be pruned because its spectrum does not underlap the worst envelope.

Algorithm 2 provides the details of our access optimal algorithm. The algorithm initializes two sets Q and S where Q contains the root of the R-tree and S contains top and bottom layers of the root node. Throughout the execution of the algorithm, Q maintains the entries of R-tree that are to be accessed in next round (i.e., must-entries) and S maintains the spectrums that are used to compute the best and worst envelopes. For each entry e in Q, the algorithm first removes its corresponding top and bottom layers from S (line 5). Then, the algorithm uses pruning rule 1 and inserts every child c in S that cannot be pruned (line 8).

Algorithm 2: KnightRider Algorithm

```
1  initialize a queue Q with root of the R-tree;
2  insert top and bottom layers of root of R-tree in S;
3  while Q is not empty do
4      for each entry e in Q do
5          remove top and bottom layers of e* from S;
6          for each child c of e do
7              if c cannot be pruned using pruning rule 1 then
8                  insert top and bottom layers of c* in S;
9      recompute best and worst envelopes using S;
10     remove entries from S using pruning rule 1;
11     Q ← intermediate or leaf nodes of S that underlap best envelope;
12 Return the best envelope;
```

After every entry e of Q is processed as described above, the algorithm recomputes the best and worst envelopes using the updated S (line 9). Since the worst envelope has been recomputed, there may be some entries in S that can be pruned using pruning rule 1. Hence, the algorithm prunes such entries (line 10). Then, the algorithm identifies the must-entries that are to be opened in next iteration. More specifically, during the computation of the best envelope at line 9, the algorithm keeps track of each entry e that underlaps the best envelope. Among these entries, the entries that are intermediate or leaf nodes of the R-tree are inserted in Q and will be accessed in the next iteration (line 11). The while loop terminates when no such entry e is found in Q. Lemma 10 shows that the best envelope at this stage is the same as k-lower envelope. Hence, the best envelope is returned (line 12).

4.1.5 Proof of Correctness

LEMMA 10 : Best envelope is the same as k-lower envelope when Algorithm 2 terminates.

PROOF. According to Lemma 6, every point that lies on k-lower envelope either lies on or lies above the best envelope. We prove the correctness by showing that, when the algorithm terminates, every point p' that lies above the best envelope cannot be a point on k-lower envelope (hence, k-lower envelope is the same as the best envelope).

Consider a vertical line passing through p' that intersects the best envelope at a point p (see Fig. 8). By definition, the point p on the best envelope has lower cardinality L at most $k-1$ and upper cardinality U at most $n-k$, i.e., $L \leq k-1$ and $U \leq n-k$ (see Fig. 8). Since the total number of lines is n, the sum of cardinalities of the bottom-layers passing through p is $X = n-(U+L)$.

Note that the algorithm terminates when there does not exist an intermediate or leaf node such that its spectrum underlaps the best envelope. This implies that every spectrum R^* that underlaps p has cardinality 1, i.e., R is a data point and the spectrum R^* is its corresponding line in dual. Hence, the total number of lines that lie on or below p is $X+L = n-U$. Since p' is a point above p, the number of lines that lie strictly below p' is at least $X+L = n-U$. Recall that $U \leq n-k$ which implies that $(X+L = n-U) \geq k$. Hence, the lower score of p' is at least k. Hence, p' is not a point on k-lower envelope. □

4.2 Optimality Claims

Due to the space limitations, we restrict the proofs for the general case of when both negative and positive weights are allowed. The proofs for the case when only non-negative weights are allowed can be obtained by following similar arguments.

4.2.1 k-lower envelope

KnightRider is access optimal for the problem of k-lower envelope because it accesses only the must-entries (see line 11 of Algorithm 2 and Lemma 9).

4.2.2 k-depth contour

KnightRider is access optimal for the case when k is smaller than the minimum number of objects in a leaf node of the R-tree. To compute k-depth contour, we need to compute both k-upper and k-lower envelopes. First, we briefly describe how KnightRider can be used to compute both k-upper and k-lower envelopes in one traversal of the R-tree. Then, we prove the optimality.

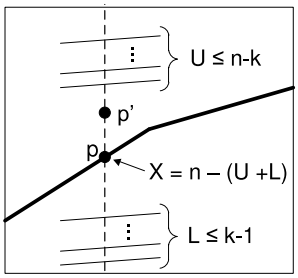


Figure 8: Proof of correctness

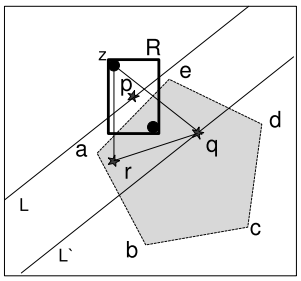


Figure 9: Illustration of Lemma 11

It is easy to modify Algorithm 2 such that it computes both k-upper and k-lower envelopes in one traversal of R-tree. Specifically, we say that a spectrum R^* *overlaps* an envelope if at least one point of R^* lies on or above the envelope. In each iteration of the algorithm, every intermediate or leaf node that either underlaps the best k-lower envelope or overlaps the best k-upper envelope is chosen to be accessed in the next round. Hence, the algorithm only accesses the spectrums that either underlap k-lower envelope or overlap k-upper envelope.

Although the algorithm is access optimal for computing k-upper and k-lower envelopes, it cannot be shown optimal for k-depth contour computation. This is because k-depth contour computation does not require exact computation of k-upper (lower) envelope (only the convex vertices are to be computed as stated in Section 2.3). Nevertheless, we show that this k-depth contour algorithm is access optimal when k is smaller than the minimum number of objects in a leaf node of the R-tree. We prove this by showing that: i) every rectangle R must be accessed if its cardinality is greater than k and it does not completely lie within the k-depth contour (Lemma 11); and ii) for a rectangle R that does not lie completely within k-depth contour, its spectrum either underlaps k-lower envelope or overlaps k-upper envelope (Lemma 12). As described above, our algorithm only accesses an entry if it either underlaps k-lower envelope or overlaps k-upper envelope.

LEMMA 11 : In order to compute the k-depth contour, every rectangle R that does not completely lie within the k-depth contour and has a cardinality greater than k must be accessed.

PROOF. We prove by contradiction. Assume that k-depth contour has been computed without accessing such a rectangle R. Fig. 9 shows this k-depth contour (the shaded polygon $abcde$) and such a rectangle R. Since R has not been accessed, we do not know anything about the orientation of objects inside it (except that it contains more than k objects and is a minimum bounding rectangle of these objects). We show that k-depth contour may be incorrect if R is not accessed, i.e., there exists at least one point p outside the contour that is not among top-k objects for *any* linear scoring function.

Let L be a line passing through a point p. Let $|\overrightarrow{L}|$ and $|\underrightarrow{L}|$ be the number of objects lying above and below L, respectively. It is known that a point p cannot be among the top-k objects for *any* linear scoring function if and only if, for every line L that passes through p, $|\overrightarrow{L}| \geq k$ and $|\underrightarrow{L}| \geq k$ (e.g., see [16]). To complete the proof, we show this for at least one point p outside k-depth contour.

Since k-depth contour is always a convex polygon [20], at least one corner of R lies outside it (e.g., z in Fig. 9). Without loss of generality, assume that k objects lie infinitely close to the this corner and the remaining objects lie elsewhere (e.g., on the opposite corner). Construct a triangle by joining z with any two arbitrary points inside the k-depth contour (e.g., see $\triangle qrz$). Since z lies outside k-depth contour, there exists at least one point p that lies inside $\triangle qrz$ and outside the k-depth contour. Note that any line L passing through p has the corner z on one side and at least one of q or r on the other side. Without loss of generality, assume that z lies above L and q lies below L (see Fig. 9). Since z contains k objects, it is obvious that $|\overrightarrow{L}| \geq k$. Now, we show that $|\underrightarrow{L}| \geq k$. We draw a line L' that is parallel to L and passes through point q. Since q lies inside k-depth contour, $|\underrightarrow{L'}| \geq k$ (because otherwise q is among top-k objects for at least one linear scoring function). Note that $|\underrightarrow{L'}| \geq k$ implies $|\underrightarrow{L}| \geq k$ which completes the proof. □

LEMMA 12 : For every rectangle R that does not lie completely within k-depth contour, R^* either underlaps k-lower envelope or overlaps k-upper envelope.

PROOF. Let $p \in R$ be a point outside or on k-depth contour. By definition of k-depth contour, p is one of the top-k objects for at least one linear scoring function. This implies that, in dual space, there exists at least one point z on p^* such that z either lies on or below k-lower envelope or lies on or above k-upper envelope. Hence, p^* either underlaps k-lower envelope or overlaps k-upper envelope. □

4.2.3 k-snippet

We compute k-lower envelope and k-upper envelope in one traversal of the R-tree by accessing only the entries that either underlap k-lower envelope or overlap k-upper envelope (as described in Section 4.2.2). The next lemma proves the optimality.

LEMMA 13 : To compute k-snippet, every optimal algorithm must access every spectrum R^* that either underlaps k-lower envelope or overlaps k-upper envelope.

PROOF. Unless the entry R is accessed, the algorithm cannot determine whether there exists an object $o \in R$ such that o^* intersects k-lower envelope or k-upper envelope. As stated in Section 2.1, every object o is a k-snippet object if at least one point p of o^* lies on or below k-lower envelope or lies on or above k-upper envelope. Hence, the algorithm may miss an object of k-snippet if R is not accessed. □

4.3 Discussion

4.3.1 Handling special cases

For the ease of presentation, we assumed that the dual mapping does not contain more than two concurrent lines and no two lines are parallel. Our techniques can be easily applied even when these assumption do not hold. More specifically, the former case can be handled by modifying the rider algorithm such that when the rider reaches an intersection of more than two lines it continues traveling on a line that has lower score equal to $k-1$. The latter case can be handled by assuming that the parallel lines intersect each other at infinity [26].

Recall that the proof of Lemma 8 does not hold if, for a rectangle R, $L_o.slope = L_d.slope = c^*.slope = d^*.slope$ where L_o, L_d, c and d are the origin line, destination line, upper right corner of R and upper left corner of R, respectively. We can prune such rectangle R if Lemma 8 does not hold, i.e., R^* lies strictly below the best envelope. This is because k-lower envelope lies on or above the best envelope, therefore, R^* does not affect any point of the k-lower envelope.

4.3.2 k-lower envelope in higher dimensionality

k-lower envelop in higher dimensionality ($d > 2$) is defined as follows. Given a set of d-dimensional hyper-planes $\mathcal{H}$, lower score of a point p is the number of hyper-planes that lie strictly below p. The k-lower envelope is the closure of the set of points that have lower scores equal to $k-1$. Below, we briefly describe how our framework can be used to compute k-lower envelope in d-dimensional space and outline the challenges.

In a d-dimensional space, a point in primal is mapped to a hyper-plane in dual and a hyper-plane in primal is mapped to a point in dual (e.g., see [8]). The pruning rules presented in this paper can be immediately applied in higher dimensionality because the basic properties of dual space mapping are preserved in higher dimensionality, e.g., a point p in primal lies above a hyper-plane H if and only if p^* lies above H^*. Hence, our framework can be used to prune the intermediate and leaf nodes of the R-tree. In fact, it can be proved that all the pruning rules and lemmas presented in Section 4 are applicable in d-dimensional space (by following the similar arguments as in Section 4). These guarantee the access optimality of our algorithm for d-dimensional space.

However, a major computational challenge is to efficiently compute the best and worst envelope in higher dimensional space. This is because the computational complexity of k-lower envelope increases exponentially with the increase in dimensionality [13]. Although the existing techniques [1, 28] can be extended and embedded in our framework to achieve the access optimality, the algorithms remain computationally very expensive due to the exponential increase in the computational complexity. In future, we intend to explore the development of approximate but efficient algorithms to compute k-lower envelope in d-dimensional space

5. EXPERIMENTS

5.1 Experimental settings

All algorithms are implemented in C++ and complied by GNU GCC. The experiments are performed on PCs with Intel Xeon 2.66 GHz CPU and 4GB memory under Debian Linux.

Synthetic data. We generate several data sets each following a different data distribution. More specifically, we generate data sets following Normal (norm for short), Correlated (corr), Anti-correlated (anti) [29] and Uniform (unif) distributions in a unit square. We also generate data sets that follow Uniform distribution in a unit circle (circ for short).

Real data. We use roads data set which contains 2,249,727 streets of California (*http://www.rtreeportal.org*). We generate 5 million objects such that each street contains around 2 objects on average. Each object represents a house and has two attributes. The first attribute indicates its distance to the nearest beach and the second attribute corresponds to the distance to nearest airport. The locations of beaches and airports are taken from a collection of points of interest in California [30]. The users may prefer houses close to (or far from) a beach and an airport. k-snippet then represents the set of houses such that each house is among top-k houses for at least one linear scoring function.

The table 1 shows different parameters used in our experiments and *the bold values are the default values used in the experiments unless mentioned otherwise*. The objects are indexed by an R-tree with page size set to 4KB (the minimum number of objects in any leaf node was 36).

Parameter	Range
Data distribution	real, norm, unif, **anti**, corr, circ
# of objects n (in millions)	1, 2, **5**, 10, 15, 20
k	1, 10, **25**, 50, 75, 100

Table 1: Experiment Parameters

5.2 Evaluation for k-lower envelope, k-snippet and k-depth contour

Recall that k-snippet and k-depth contour can be cheaply computed if k-lower envelope and k-upper envelope are known. Hence, as a benchmark, we evaluate our algorithms for the problem of k-depth contour and compare with state-of-the-art algorithms for k-depth-contour and k-lower envelopes. Later in Section 5.4, we present a more detailed evaluation specific to the problem of k-snippet. Below are the competitors we consider in this section.

BELT [26]. Computational geometry community proposed several algorithms with complexities close to the optimal. Despite the fact that these algorithms provide nice complexity guarantees, unfortunately, these do not work well in practice. Nevertheless, we choose one such algorithm called BELT algorithm [26] that computes k-upper and k-lower envelopes with computational complexity guarantees close to optimal.

FDC [18]. We compare our algorithm with FDC algorithm which is known to be the most efficient algorithm for k-depth contour.

FDC-Index. Note that both BELT and FDC are main memory algorithms and do not exploit the pre-built indexes. While it is non-trivial to extend BELT to utilize the index, we enhance FDC such that it utilizes the pre-built R-tree. In each round, FDC requires computing convex hull of a set of points. To do so, FDC employs a main-memory index-agnostic convex hull algorithm. To improve the performance, we replace the convex hull computation with an index-aware algorithm [31] that is I/O optimal for convex hull com-

putation on the data sets indexed by R-tree. This variation of FDC is called FDC-Index.

5.2.1 Comparison with lower bound I/O cost

As shown in Section 4.2, KnightRider algorithm is access optimal for k-lower envelope and k-snippet problems. Unfortunately, KnightRider is not access optimal for k-depth contour problem when k is large. Nevertheless, our experimental results demonstrate that the I/O cost of KnightRider is quite close to a lower bound cost for the k-depth contour problem even when k is very large.

It is easy to show that every optimal k-depth contour algorithm must access every spectrum R^* that contains a convex vertex (this can be shown by following similar arguments as in the proof of Lemma 7). Hence, to obtain the lower bound I/O cost, we assume that an oracle computes all the convex vertices (without incurring any I/O). Then, we traverse R-tree accessing only the entries such that their spectrums overlap one of these convex vertices. These I/Os are counted and correspond to the lower bound I/O cost.

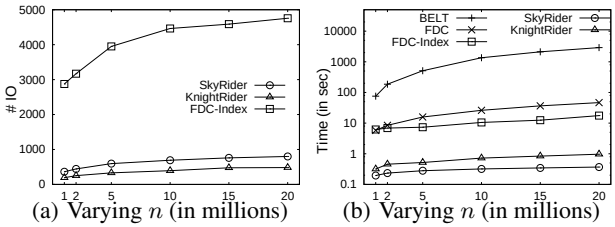
k	1	10	100	1000	10,000	100,000
SkyRider	147	400	933	2081	9864	53907
KnightRider	114	241	453	670	2444	12294
Lower Bound	114	241	452	667	2442	12293

Table 2: Number of I/Os for k-depth contour problem

Table 2 demonstrates that the I/O cost of KnightRider algorithm is almost the same as the lower bound I/O cost even when $k = 100,000$. Recall that we were unable to show the optimality because k-depth contour may be computed correctly even if only the convex vertices are computed instead of k-upper and k-lower envelopes. We observe that it is extremely rare that a rectangle affects the k-upper and k-lower envelopes but does not affect the convex vertices. This is the reason that the I/O cost of our algorithm is almost the same as the lower bound I/O cost.

5.2.2 Effect of data set size

In Fig. 10, we vary the data set size and evaluate I/O and CPU costs of all algorithms. Since FDC and BELT assume that the data reside in main-memory, they need to access the whole data set at least once in order to load it in the memory. We remark that, for FDC and BELT, we do not count the time it takes to load the data in memory.



(a) Varying n (in millions) (b) Varying n (in millions)

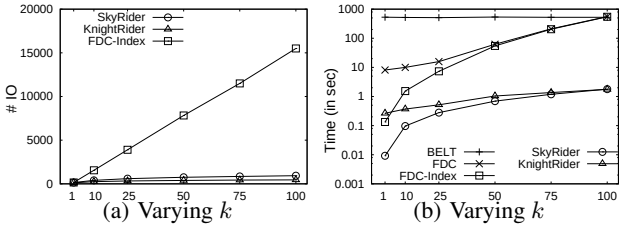
Figure 10: Effect of data sizes

Fig. 10(a) shows the I/O cost for each algorithm. The number of I/Os for FDC and BELT varies from $14,634$ to $291,533$ and is not shown in Fig. 10(a) for a better illustration of the comparison between the other algorithms. The I/O cost for FDC-Index is much higher than our algorithms. KnightRider has a lower I/O cost than SkyRider because KnightRider is I/O optimal for smaller values of k and is almost I/O optimal for larger values.

Fig. 10(b) shows the CPU time for each algorithm. Our algorithms are up to three orders of magnitude faster than the other algorithms. Although KnightRider is access optimal and outperforms SkyRider in terms of I/O cost, it can be noted that SkyRider is more efficient than KnightRider in terms of CPU cost. This is mainly because KnightRider needs to spend more computational efforts in order to prune the nodes and achieve access optimality.

5.2.3 Effect of k

In Fig. 11, we vary the value of k and evaluate the performance of each algorithm. Fig. 11(a) shows that our algorithms do not only have much lower I/O cost but also scale much better. The number of I/Os for FDC and BELT is $73,133$ and is not shown in the figure.



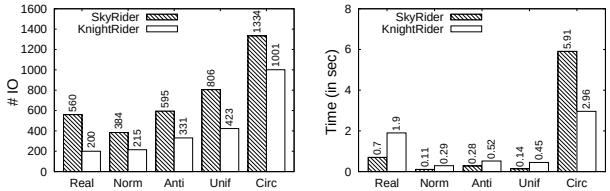
(a) Varying k (b) Varying k

Figure 11: Effect of k

Fig. 11(b) shows that our algorithms are up to three orders of magnitude faster than the other algorithms. Note that when $k = 1$, the problem of k-depth contour is the same as the problem of computing the convex hull. Also, FDC-Index in this case is the same as the I/O optimal algorithm proposed in [31]. Although our algorithms are designed to solve a more complicated problem, our algorithms perform quite well even for the convex hull problem (i.e., $k = 1$).

5.2.4 Effect of data distribution

Fig. 12 studies the effect of data distribution on our algorithms. Since the I/O and CPU costs of BELT, FDC and FDC-Index are quite high, in Fig. 12, we only show the costs of our algorithms for a better illustration of their comparison. Fig. 12(a) shows that the I/O cost of KnightRider is significantly smaller than the I/O cost of SkyRider. Fig. 12(b) shows that CPU cost of SkyRider is lower than KnightRider for all data sets except the circ data set. This is because the difference in the number of unpruned data points by the two algorithm becomes more significant (e.g., Fig. 15(b) shows the difference of almost 7 times for circ data set). Consequently, the computational cost of the Rider algorithm dominates the pruning cost which results in a larger overhead for SkyRider.



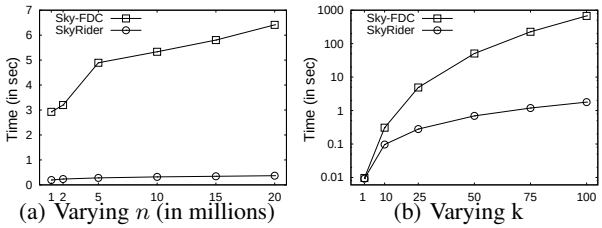
(a) Varying distribution (b) Varying distribution

Figure 12: Effect of different data distributions

5.2.5 Effectiveness of the rider algorithm

An alternative approach to compute the k-lower envelope is to first compute k-skybands using BBS (as discussed in Section 3.2)

and then use some existing main-memory algorithm (e.g., FDC) instead of the rider algorithm. We call such algorithm Sky-FDC algorithm. In this section, we demonstrate the effectiveness of the rider algorithm by showing that SkyRider is significantly more efficient than Sky-FDC algorithm. Fig. 13 shows the results for varying n and varying k and demonstrates that SkyRider is up to two orders of magnitude faster than Sky-FDC (note that Fig. 13(b) uses logscale). I/O costs are not shown because both SkyRider and Sky-FDC have the same I/O cost (both algorithms use BBS to compute k-skybands).

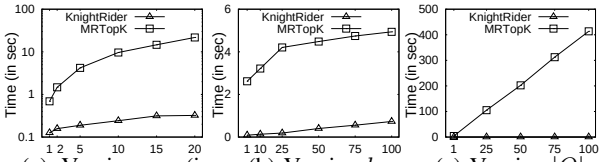


(a) Varying n (in millions) (b) Varying k

Figure 13: Effectiveness of the rider algorithm

5.3 Reverse top-k queries

We compare our algorithm with MRTopK [25] which is the state-of-the-art algorithm for answering a reverse top-k query. Since MRTopK studies the queries involving only non-negative weights, we also conduct the experiments for the non-negative weights. I/O costs are not shown because MRTopK accesses the whole data set at least once and has significantly higher I/O cost than our algorithm. We issue a single reverse top-k query for each setting and study the effect of varying n and varying k in Fig. 14(a) and Fig. 14(b), respectively. KnightRider is up to two orders of magnitude faster than MRTopK. In Fig, 14(c), we issue multiple reverse top-k queries having the same value of k, i.e., $k = 25$ for each query. Note that when the number of queries $|Q|$ increases, our algorithm scales much better. This is mainly because, once our algorithm computes k-lower envelope, each query q can be answered cheaply by merely computing the intersection of q^* and the envelope.



(a) Varying n (in millions) (b) Varying k (c) Varying $|Q|$

Figure 14: Reverse top-k queries

5.4 k-snippet vs k-skyband

k-snippet and k-skyband are closely related and both serve as data summaries that can be used to answer all top-m ($m < k$) queries. The difference between these two problems is that k-skyband considers all monotonic scoring functions whereas k-snippet considers only the linear monotonic functions. Although we do not claim that k-snippet is superior to k-skyband in all applications, it should be preferred to k-skyband when only linear scoring functions are to be considered. This is mainly because the size of k-snippet is up to 7 times smaller than the size of k-skyband as shown in Fig. 15(a) and 15(b). Hence, k-snippet provides a more compact summary.

Next, we compare the performance of our KnightRider algorithm with BBS [14] which is an access optimal algorithm for k-skyband under the same settings as used by our algorithm, i.e., for the disk-resident data indexed by R-tree. We compare the I/O costs of both algorithms in Fig. 15(c) and 15(d) and the CPU costs of both algorithms in Fig. 15(e) and 15(f). Since both algorithms are I/O optimal for their respective problems, the I/O cost of KnightRider is lower because it requires to return a smaller set of objects. In terms of CPU cost, BBS performs better than KnightRider except for circ data set which is the most challenging data set for both of the algorithms. The performance of BBS is better because the requirement to prune objects for k-snippet is significantly more complicated than k-skyband.

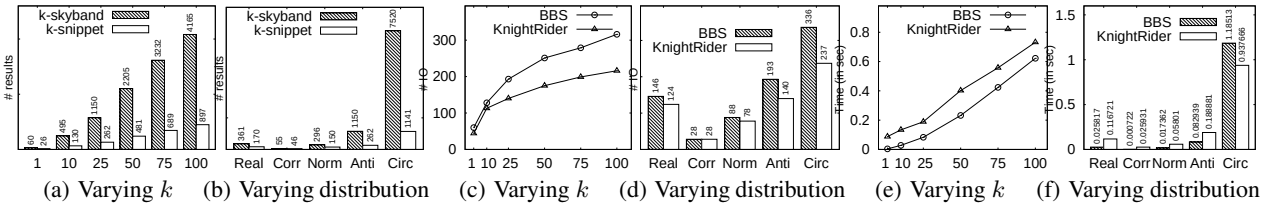
6. RELATED WORK

k-lower envelope. k-lower envelope has received significant research attention from the computational geometry community (see [1] for a nice survey). As mentioned earlier, all the existing algorithms assume that the data reside in main memory and is not indexed. In [9, 8], k-lower envelope has been used to answer the top-k queries involving linear scoring functions. However, they rely on the existing techniques to compute k-lower envelope.

k-snippet. To the best of our knowledge, we are the first to propose the concept of k-snippet. k-skyband [14] is a similar problem has received significant research attention and serves as a data summarization tool for all monotonic scoring functions. While k-skyband handles more general functions than the k-snippet, it contains unnecessary objects if the users are only interested in linear scoring functions. Several techniques [32, 33, 34] have been developed to return a fix number of objects from 1-skyband. However, these techniques cannot guarantee that the object that the user prefers the most is among the returned objects.

k-depth contour. Due to its various applications in statistics, the computation of k-depth contour on two dimensional data has received significant research attention from the statistics community. Ruts and Rousseeuw developed a series of algorithms to compute k-depth contour (e.g., ISODEPTH [22], HALFMED [35] and BAGPLOT [24]). The researchers from the computational geometry community have also proposed several efficient algorithms for k-depth contour computation [17, 20, 19]. Inspired by the usefulness of k-depth contour in outlier detection, Johnson *et al.* [18] proposed an algorithm that was shown to outperform all of the existing algorithms. Unfortunately, all of the above algorithms are main-memory algorithms and are not suitable for disk-resident data sets. Böhm and Kriegel [31] proposed I/O optimal algorithms for computing convex hull which is a special case of k-depth contour where $k = 1$. We remark that solving k-depth contour is considerably more challenging and it is non-trivial to extend their techniques to solve this problem.

Reverse top-k queries. Vlachou *et al.* [25] are the first to introduce the reverse top-k queries. They proposed two variants of the reverse top-k queries namely *monochromatic* and *bichromatic* reverse top-k queries. A monochromatic query returns every possible scoring function for which q is one of the top-k objects. On the other hand, in a bichromatic query, a set of scoring functions F is given and every function $f \in F$ is returned for which q is one of the top-k objects. Note that we study the monochromatic reverse top-k query that is more general, i.e., bichromatic query can be easily answered if the results of the monochromatic query are known. Chester *et al.* [36] propose an index specific for a fixed value k such that multiple reverse top-k queries can be efficiently answered using this index. If the value of k is unknown at the query time (which is usually the case), the index is to be constructed on-the-fly



(a) Varying k (b) Varying distribution (c) Varying k (d) Varying distribution (e) Varying k (f) Varying distribution

Figure 15: Comparison with k-skyband and BBS [14]: a-b) Result size; c-d) I/O cost; e-f) CPU time

which becomes the bottleneck. Furthermore, the proposed index pays off only when numerous queries use the same value of k. Vlachou *et al.* [37] proposed a branch-and-bound algorithm to solve bichromatic reverse top-k queries. However, the proposed algorithm is not applicable to the monochromatic reverse top-k query. Yu *et al.* [8] focused on presenting efficient solutions for continuous bichromatic reverse top-k queries for the case when the set of objects O and/or the set of scoring functions F change.

7. CONCLUSIONS

We are the first to study the problem of k-lower envelope computation on disk-resident indexed data sets. We develop two efficient I/O and CPU efficient algorithms including an access optimal algorithm. We also complement the ranking systems by proposing solutions to several interesting queries namely k-snippet, k-depth contour and reverse top-k queries. Our main algorithm is access optimal for the problem of k-lower envelope and k-snippet. It is also optimal for k-depth contour for the smaller values of k. Our experiments demonstrate that our algorithms are up to several orders of magnitude better than the existing algorithms for these problems.

Acknowledgments. Muhammad Aamir Cheema is supported by ARC DE130101002 and DP130103405. Xuemin Lin is supported by NSFC61232006, NSFC61021004, ARC DP120104168 and DP 110102937. Wenjie Zhang is supported by ARC DE120102144 and DP120104168. This research was partially conducted while the first two authors were with The University of New South Wales.

8. REFERENCES

[1] P. K. Agarwal and M. Sharir, "Arrangements and their applications," in *Handbook of Computational Geometry*, 1998, pp. 49–119.

[2] J. Matoušek, *Lectures on discrete geometry*. Springer Verlag, 2002.

[3] P. K. Agarwal, M. de Berg, J. Matousek, and O. Schwarzkopf, "Constructing levels in arrangements and higher order voronoi diagrams," *SIAM J. Comput.*, vol. 27, no. 3, pp. 654–667, 1998.

[4] T. M. Chan, "Random sampling, halfspace range reporting, and construction of (<= k)-levels in three dimensions," *SIAM J. Comput.*, 2000.

[5] B. Chazelle and H. Edelsbrunner, "An improved algorithm for constructing *k* th-order voronoi diagrams," *IEEE Trans. Computers*, 1987.

[6] B. Chazelle and F. P. Preparata, "Halfspace range search: An algorithmic application of k-sets," *Discrete & Computational Geometry*, 1986.

[7] K. L. Clarkson, "New applications of random sampling in computational geometry," *Discrete & Computational Geometry*, 1987.

[8] A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-*k* queries," in *SIGMOD Conference*, 2012.

[9] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas, "Ad-hoc top-k query answering for data streams," in *VLDB*, 2007, pp. 183–194.

[10] Z. Shen, M. A. Cheema, and X. Lin, "Loyalty-based selection: Retrieving objects that persistently satisfy criteria," in *CIKM*, 2012.

[11] A. Borobia, "Mirror property for nonsingular mixed configurations of lines and points in r^3," *Discrete & Computational Geometry*, 1994.

[12] H. Everett, J.-M. Robert, and M. J. van Kreveld, "An optimal algorithm for the (<= k)-levels, with applications to separation and transversal problems," *Int. J. Comput. Geometry Appl.*, 1996.

[13] P. Agarwal and J. Matoušek, "Dynamic half-space range reporting and its applications," *Algorithmica*, vol. 13, no. 4, pp. 325–345, 1995.

[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, 2005.

[15] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *SIGMOD*, 2003.

[16] D. Xin, C. Chen, and J. Han, "Towards robust indexing for ranked queries," in *VLDB*, 2006, pp. 235–246.

[17] R. Cole, M. Sharir, and C.-K. Yap, "On k-hulls and related problems," in *STOC*, 1984, pp. 154–166.

[18] T. Johnson, I. Kwok, and R. T. Ng, "Fast computation of 2-dimensional depth contours," in *KDD*, 1998, pp. 224–228.

[19] S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian, "Hardware-assisted computation of depth contours," in *SODA*, 2002.

[20] K. Miller, S. Ramaswami, P. Rousseeuw, J. A. Sellarès, D. L. Souvaine, I. Streinu, and A. Struyf, "Fast implementation of depth contours using topological sweep," in *SODA*, 2001, pp. 690–699.

[21] L. Kong and Y. Zuo, "Smooth depth contours characterize the underlying distribution," *J. Multivariate Analysis*, 2010.

[22] I. Ruts and P. Rousseeuw, "Computing depth contours of bivariate point clouds," *Computational Statistics & Data Analysis*, 1996.

[23] J. W. Tukey, "Mathematics and picturing of data," in *International Congress of Mathematicians*, 1974.

[24] P. J. Rousseeuw, I. Ruts, and J. W. Tukey, "The bagplot: A bivariate boxplot," *The American Statistician*, vol. 53, no. 4, Nov. 1999.

[25] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg, "Reverse top-k queries," in *ICDE*, 2010, pp. 365–376.

[26] H. Edelsbrunner and E. Welzl, "Constructing belts in two dimensional arrangements with applications," *SIAM J. Comput.*, 1986.

[27] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang, "A safe zone based approach for monitoring moving skyline queries," in *EDBT*, 2013.

[28] K. Mulmuley, "On levels in arrangements and voronoi diagrams," *Discrete & Computational Geometry*, 1991.

[29] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.

[30] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *SSTD*, 2005.

[31] C. Böhm and H.-P. Kriegel, "Determining the convex hull in large multidimensional databases," in *DaWaK*, 2001, pp. 294–306.

[32] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu, "Regret-minimizing representative databases," *PVLDB*, 2010.

[33] Y. Tao, L. Ding, X. Lin, and J. Pei, "Distance-based representative skyline," in *ICDE*, 2009, pp. 892–903.

[34] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting stars: The k most representative skyline operator," in *ICDE*, 2007, pp. 86–95.

[35] P. J. Rousseeuw and I. Ruts, "Constructing the bivariate tukey median," *Statistica Sinica*, pp. 827–839, 1998.

[36] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Indexing reverse top-k queries in two dimensions," in *DASFAA*, 2013.

[37] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis, "Branch-and-bound algorithm for reverse top-k queries," SIGMOD 2013.