Determining the Impact Regions of Competing Options in Preference Space

Bo Tang
Hong Kong Polytechnic Uni.
csbtang@comp.polyu.edu.hk

Kyriakos Mouratidis
Singapore Management Uni.
kyriakos@smu.edu.sg

Man Lung Yiu
Hong Kong Polytechnic Uni.
csmlyiu@comp.polyu.edu.hk

ABSTRACT

In rank-aware processing, user preferences are typically represented by a numeric weight per data attribute, collectively forming a *weight vector*. The score of an option (data record) is defined as the weighted sum of its individual attributes. The highest-scoring options across a set of alternatives (dataset) are shortlisted for the user as the recommended ones. In that setting, the user input is a vector (equivalently, a point) in a d-dimensional preference space, where d is the number of data attributes. In this paper we study the problem of determining in which regions of the preference space the weight vector should lie so that a given option (*focal record*) is among the top-k score-wise. In effect, these regions capture all possible user profiles for which the focal record is highly preferable, and are therefore essential in market impact analysis, potential customer identification, profile-based marketing, targeted advertising, etc. We refer to our problem as k-*Shortlist Preference Region* identification (kSPR), and exploit its computational geometric nature to develop a framework for its efficient (and exact) processing. Using real and synthetic benchmarks, we show that our most optimized algorithm outperforms by three orders of magnitude a competitor we constructed from previous work on a different problem.

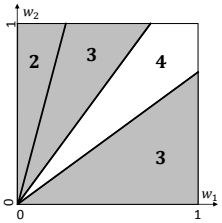
1. INTRODUCTION

Recommendation systems, multi-criteria decision making and ranking queries have been widely explored in the past few years. In the most common preference model, recommendations are produced by top-k queries with linear scoring functions [9, 19, 20]. Consider a user who visits an online portal, such as Yelp or Zagat, to choose a restaurant based on its *value*, *service* and *ambiance* ratings. The user may specify a numeric weight w_i for each criterion, where the higher the value of w_i, the higher the relative significance of that criterion in her decision. Essentially, the user specifies a *weight vector* $\mathbf{w} = (w_1, w_2, w_3)$ in a 3-dimensional *preference space*. That vector associates each restaurant with a numeric score, equal to the weighted sum of its ratings. The k (say, the 10) highest-scoring restaurants are reported by the portal as recommended.

Different vectors in preference space produce different restaurant rankings and, thus, different recommendations to the user. From the perspective of a restaurant owner, it is essential to know in which regions of the preference space her restaurant **p** is among the recommended, i.e., among the top-10. First, these regions indicate the profiles of users that would be most interested in **p**. For example, if the regions concentrate in the area where w_3 is larger than w_1 and w_2, it means that **p** is most appealing to customers that care primarily about the ambiance, rather than value and service. This fact could help the restaurant owner anticipate the type of her clientele (e.g., people planning for a romantic dinner or a fancy business meal) and/or target her advertisement efforts to the right crowd (e.g., users of dating sites, managerial employees, etc). Furthermore, using these regions we can compute the probability that restaurant **p** belongs to the top-k list for a random user, which in turn is a direct measure of market impact. If the weight vector **w** is equally likely to be anywhere in the preference space, that probability is equal to the summed volume of the regions divided by the total volume of the preference space. Even more practically, if the probability density function (PDF) of **w** is known (e.g., extracted from past user queries [27, 12, 6]), the probability is computed by integrating the PDF across the extent of the regions.

Restaurant	Record	Value	Service	Amb.
L'Entrecôte	$\mathbf{r}_1$	3	8	8
Beirut Grill	$\mathbf{r}_2$	9	4	4
El Coyote	$\mathbf{r}_3$	8	3	4
La Braceria	$\mathbf{r}_4$	4	3	6
Kyma	**p**	5	5	7

(a) Data records



(b) Preference space

Figure 1: Restaurant records and kSPR result for $k = 3$

We refer to the problem of finding all regions in preference space where a *focal record* **p** belongs to the top-k recommendation as k-*Shortlist Preference Region* identification (kSPR). To exemplify, we use the dataset in Figure 1(a), where each record corresponds to a restaurant and contains its ratings (on a scale of 1 to 10) in terms of value, service, and ambiance. For ease of visualization, we consider only value and service; the ambiance ratings will be used in a subsequent example. Suppose that the focal record is *Kyma* and that $k = 3$. Figure 1(b) shows the rank of *Kyma* in different regions of the preference space. The kSPR query reports the part of the preference space shown in gray, i.e., *Kyma* is among the top-3 restaurants for any weight vector in the gray area. Observe that in general there are multiple, disconnected kSPR regions. Also, in different settings there may be more than just two dimensions.

The kSPR query finds application in rank-aware scenarios that instead of restaurants may involve hotels, properties for rent/sale, or even players of competitive sports. For example, in Section 7.2

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3064044

we present a case study based on actual NBA statistics for the 2014-2015 and 2015-2016 seasons. The kSPR regions for *Dwight Howard* reveal that he is among the top-3 NBA players for a broad spectrum of preferences in both seasons. The most interesting insight, however, is that what makes him stand out in 2014-2015 is his point-scoring prowess while, conversely, in 2015-2016 it is his defensive skills. This type of information could help a manager to effectively market the player in each season.

Previous work includes reverse top-k processing and related queries [31, 32, 36], where a finite set of specific weight vectors is given, and the objective is to identify those of the vectors that rank a focal record the highest. These methods are tailored to fixed, discrete weight vectors, and are unable to consider the entire, continuous preference space. That said, [31] includes a technique applicable to a degenerate case of kSPR where the dimensionality of the preference space is effectively 1, but it does not extend to more dimensions. Some recent studies do consider the (continuous) preference space, but none suits our problem. E.g., they identify the most representative top-k results when the weight vector is uncertain/unknown [28, 26], compute a region around a given weight vector where the top-k result remains the same [22, 35], or derive the highest rank attainable by a specific record for any possible weight vector [23]. While in theory an incremental version of [23] could solve the kSPR problem, it is particularly ineffective, scaling only to small kSPR instances and being three orders of magnitude slower than our approach.

In this paper we propose a general kSPR methodology, which produces exact answers and is designed for high efficiency and scalability. We model kSPR as a computational geometric problem in an arrangement of hyperplanes, and develop a data structure ($CellTree$) to maintain that arrangement. A key principle in our approach is that the exact geometry of cells in the arrangement is not computed, unless they are guaranteed to be in the kSPR result. Instead, each cell is implicitly represented as a set of linear inequalities or, equivalently, halfspaces. As opposed to considering all competing options to **p** at once, we follow a particular processing order that allows pruning a large fraction of them. To further enhance pruning effectiveness, we show how this order can be virtually individualized per cell, so that local, stricter pruning criteria can be imposed for each of them. We additionally devise look-ahead techniques that work in the data space (and exploit the index on the data records) to disqualify parts of the $CellTree$ from consideration. As we demonstrate with experiments on standard (real and synthetic) benchmarks, the above components render our approach three orders of magnitude faster than a baseline we constructed from previous work.

2. RELATED WORK

The term preference-based querying refers to the shortlisting of a number of options (data records) from a set of available alternatives (dataset) based on their attributes (dimensions). The most common preference-based operators are the skyline [7] and the top-k query [9, 19, 15]. The former requires no user input, while the latter provides personalized results based on stated user preferences. In particular, the skyline of a dataset includes those records that are dominated by no other; we say that a record dominates another if it is no less preferable in all dimensions and better in at least one. Many algorithms have been proposed for skyline processing with and without indices [29, 21, 25]. On the other hand, the top-k query requires the user to specify a scoring function and reports the k records with the highest scores in the dataset. The most intuitive and common type of scoring functions are linear, i.e., the user specifies a weight per dimension and the score of a record is defined as the weighted sum of its attributes. Ilyas et al. [20] review a flurry of top-k processing techniques. Between the two standard preference-based operators, the top-k query, and generally the rank-aware processing model it defines, is closest related to our work. In the following we describe variants and auxiliary features to rank-aware querying.

In [13] Das et al. consider the processing of ad-hoc top-k queries in a dynamic buffer of data records, e.g., a sliding window over a data stream. Their objective is to maintain and process only a subset of the valid records. To achieve that, they exploit a transformation (applicable exclusively to two dimensions) where records are represented by lines and top-k queries by vertical rays. In that transformed space, only records whose lines are among the closest to the horizontal axis could appear in the top-k result of an ad-hoc query. Yu et al. [34] exploit a similar transformation to facilitate the processing of continuous top-k queries. At the heart of their approach lies a piecewise linear surface that codifies the score of the k-th record for all top-k queries, very similar in spirit to the k-level construct in computational geometry [11, 5]. They also propose a method for approximate top-k processing based on a carefully chosen subset of the data records.

Vlachou et al. [30, 31] introduce the reverse top-k query. Starting with a set of different weight vectors (i.e., user preferences), this query identifies those of the weight vectors that rank a specific data record **p** among their top-k. In follow-up work, the same authors improve the performance of their operator [33] and consider related formulations, such as [32], where they identify the records that rank among the top-k for most of the input weight vectors. Observe that these problems are discrete by nature, in the sense that a finite set of specific weight vectors is given (as opposed to considering any possible weight vector in the preference space).

That said, [30, 31] also discuss a 2-dimensional version of the problem, which they call monochromatic, that is not bound to a given set of weight vectors. Specifically, in two dimensions the scoring function can be expressed in the form $a \cdot r_1 + (1-a) \cdot r_2$, where r_1, r_2 are the data attributes and a represents the user's preference ($a \in [0, 1]$). Here the preference space is the (1-dimensional) line segment from 0 to 1. The authors compute the intervals of a values for which **p** ranks among the top-k records. That is essentially a kSPR problem for the special case of 2-dimensional records. Their solution relies on the fact that for any two records that do not dominate each other, there is exactly one value of a where their relative order (score-wise) changes. Thus, by comparing **p** with every other data record, an equal number of such switching values are derived. These values impose a partitioning of the preference space (line segment) into disjoint intervals. By scanning the intervals from $a = 0$ to $a = 1$, it is easy to incrementally maintain how many records score higher than **p** in each of them, and therefore report those where **p** ranks among the top-k. This algorithm capitalizes on the 1-dimensional preference space and does not extend to higher dimensions. The authors recognize this and indicate the challenges involved in more dimensions. Our methodology, in addition to applying to higher dimensions, is faster even for this special case of $d = 2$, as we show in the experiments.

Similar to the standard reverse top-k query, Zhang et al. [36] also consider a finite set of specific weight vectors. Among them, they identify the m vectors that rank a given record **p** the highest. They refer to this query as reverse k-ranks. They also consider the reverse k-scores variant, where they identify the m weight vectors for which the score of **p** is the highest. To efficiently disqualify (i.e., prune) some of the weight vectors, they index them with a regular grid. Their solutions work purely in the data space and have little to do with the geometry of the problem, since they consider specific

weight vectors for which exact scores can be derived readily and inexpensively.

Relevant to our problem is also the work on immutable regions [22, 35]. These regions define an area around the user's weight vector $\mathbf{w}$ where the top-k result remains the same. The crux of these methods is to determine a small fraction of non-result records that bound the immutable regions, be them 1-dimensional (i.e., defined locally for each of the d weights) or d-dimensional (i.e., defined as a region around $\mathbf{w}$ in the preference space). In the latter case, each retained non-result record corresponds to a halfspace in preference space, and the immutable region is derived as the intersection of these halfspaces.

Geometric observations in preference space have also been utilized in the context of uncertain weight vectors. [28] proposes methods to identify the most representative top-k result when the weight vector is unknown. Several formulations are considered, but its main focus is on deriving the most probable top-k result when the weight vector is equally likely to be anywhere in the preference space. [26] assumes that the distribution of the weight vectors is known and describes a technique to select a specific number of records from the dataset, such that the top-1 answer for a random weight vector has the highest probability to be in the chosen subset. Faced with the high complexity of the geometric operations involved, the authors employ sampling and provide approximate results with probabilistic guarantees.

A recent study on the maximum rank query [23] is also related to our work. That query computes the best rank, k^*, that a certain record $\mathbf{p}$ could achieve under any possible weight vector, and it also identifies the regions of the preference space which correspond to that rank. Every record $\mathbf{r}$ in the dataset is mapped to a halfspace (in preference space) where it scores higher than $\mathbf{p}$. The produced halfspaces partition the preference space into cells, and each cell is associated with the count of halfspaces that include it. The minimum of all counts is reported as k^*, together with the cells that have that count. The proposed technique partitions the preference space with a Quad-tree, and processes its leaves one by one, in increasing order of the number of halfspaces they lie in. That allows the pruning of some leaves, i.e., regions of the preference space, which are guaranteed to have counts greater than k^*. The processing within each leaf solves multiple halfspace intersection problems in order to derive the most promising cells that are in the leaf.

The method in [23] extends easily to incremental reporting. That is, if k^* is the best $\mathbf{p}$ could rank, processing can continue in the same fashion for ranks k^*+1, k^*+2, etc, to produce the corresponding regions in preference space. The computation cost, however, grows exponentially for any increase by 1. Although not suited to our problem, this incremental method could in theory answer a kSPR instance by computing k^*, incrementing it up to k, and reporting the cells produced for each rank between k^* and k to form the kSPR regions. We use this as a baseline competitor in the experiments for some small kSPR instances, as it fails to terminate for our full-scale settings.

Related is also the work on why-not queries, both for top-k [18] and for reverse top-k formulations [16]. A why-not top-k query seeks to amend the weight vector $\mathbf{w}$ and the k value of a given top-k query so that a certain non-result record becomes part of the top-k result, by incurring the minimum penalty. The penalty function takes into account the Euclidean distance between the original and the amended weight vector, and the required increase in k. A why-not reverse top-k query, on the other hand, determines how to modify a weight vector $\mathbf{w}$ or the focal record $\mathbf{p}$ or the value of k, so that $\mathbf{w}$ is included in the reverse top-k result of $\mathbf{p}$. Although both [18] and [16] exploit geometric properties in preference space, these why-not problems (and the techniques proposed for their processing) are fundamentally different from ours.

Cai et al. [8] develop greedy algorithms to compute a d-dimensional box around a data record $\mathbf{p}$, such that the box includes at least a certain number of other records, and $\mathbf{p}$ ranks the highest possible among the records in the box. This work defines rank in terms of a stored data attribute (instead of the value of an aggregate scoring function), it computes a region in data space (as opposed to regions in preference space), and reports heuristically derived (i.e., inexact) answers.

3. PRELIMINARIES

3.1 Problem Definition

Each record $\mathbf{r}$ in the dataset D is represented as a vector $\mathbf{r} = (r_1, r_2, \cdots, r_d)$. The user's preferences are captured by a weight vector $\mathbf{w} = (w_1, w_2, \cdots, w_d)$. The score of record $\mathbf{r}$ is defined as:

$$S(\mathbf{r}) = \mathbf{r} \cdot \mathbf{w} = \sum_{i=1}^{d} r_i w_i \tag{1}$$

Given a dataset D, a weight vector $\mathbf{w}$, and an integer k, the top-k result includes the k records with the highest scores in D. For ease of presentation, we ignore ties. Without loss of generality, we assume that (i) $w_i > 0$ for every dimension, and (ii) $\sum_{i=1}^{d} w_i = 1$. Note that the normalization of $\mathbf{w}$ does not restrict the semantics of the ranking function in any way [20]. Our problem is defined as:

PROBLEM 1. *The k-Shortlist Preference Region problem (kSPR) takes as input a dataset D, a focal record $\boldsymbol{p} = (p_1, p_2, \cdots, p_d)$, and an integer k. It reports all the regions in preference space where if the weight vector lies, $\boldsymbol{p}$ ranks among the top-k records.*

Consider a record $\mathbf{r}$ that dominates $\mathbf{p}$, i.e., in every dimension the value of $\mathbf{r}$ is no smaller than that of $\mathbf{p}$, and there is at least one dimension where the value of $\mathbf{r}$ is greater. It holds that $\mathbf{r}$ scores higher than $\mathbf{p}$ for any weight vector [7]. Therefore, the kSPR solution on D is the same as the kSPR solution if we ignore the records that dominate $\mathbf{p}$ and reduce k by their number. On the other hand, any record that is dominated by $\mathbf{p}$ always scores lower than $\mathbf{p}$, and does not affect kSPR processing at all. Records in both aforementioned categories can be easily identified and disregarded using an index on D. To keep presentation simple, the following discussion assumes that the records that dominate or are dominated by $\mathbf{p}$ have already been removed from D.

We assume that D is indexed by a spatial access method, such as an R-tree [4], and that data and index are kept in main memory. However, in an extra set of experiments (in Appendix A) we also consider the scenario where they reside in secondary storage.

3.2 Problem Reduction

The normalization of the weight vectors as described in Section 3.1 allows us to reduce the dimensionality of the preference space by 1. Specifically, since $\sum_{i=1}^{d} w_i = 1$, the d-th weight is defined as $w_d = 1 - \sum_{i=1}^{d-1} w_i$. Thus, we may work in a transformed preference space, with axes $w_1, w_2, \cdots, w_{d-1}$. This reduction of the dimensionality to $d' = (d-1)$ is important, since the running time of the costliest operations in our methodology depends on the dimensionality; in Appendix C we assess the gains from the reduction. In the following, unless otherwise specified, we refer to the transformed preference space.

Consider a record $\mathbf{r}$ and the focal record $\mathbf{p}$. Equation $S(\mathbf{r}) = S(\mathbf{p})$ corresponds to a hyperplane h in the preference space. Every

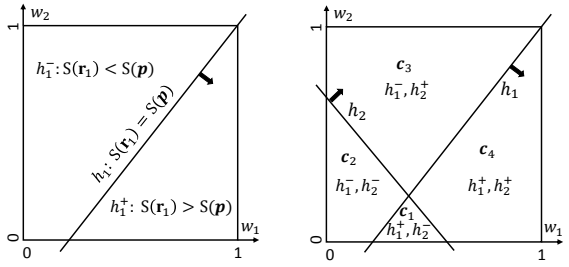
weight vector that falls on this hyperplane renders **r** and **p** equally preferable. Specifically, h is defined as:

$$S(\mathbf{r}) = S(\mathbf{p}) \Longleftrightarrow \sum_{i=1}^{d} r_i w_i = \sum_{i=1}^{d} p_i w_i$$
$$\Longleftrightarrow r_d + \sum_{i=1}^{d-1}(r_i - r_d)w_i = p_d + \sum_{i=1}^{d-1}(p_i - p_d)w_i$$
$$\Longleftrightarrow \sum_{i=1}^{d-1}(r_i - r_d - p_i + p_d)w_i = p_d - r_d$$

Hyperplane h partitions the transformed preference space into two complementary halfspaces:

- *positive halfspace* h^+ where **r** scores higher than **p**, i.e., $S(\mathbf{r}) > S(\mathbf{p})$, and
- *negative halfspace* h^- where **r** scores lower than **p**, i.e., $S(\mathbf{r}) < S(\mathbf{p})$.

Assuming the restaurant records in Figure 1(a), and considering all three data attributes (value, service, ambiance), the transformed preference space has $(d-1) = 2$ dimensions as shown[1] in Figure 2(a). Record $\mathbf{r}_1$ (i.e., restaurant *L'Entrecôte*), when compared to the focal record **p** (i.e., *Kyma*), corresponds to hyperplane h_1 with equation $S(\mathbf{r}_1) = S(\mathbf{p})$ that divides the preference space into two halfspaces, i.e., h_1^+ where $S(\mathbf{r}_1) > S(\mathbf{p})$, and h_1^- where $S(\mathbf{r}_1) < S(\mathbf{p})$. The black arrow points to the positive halfspace.



(a) Hyperplane h_1 (b) Arrangement

Figure 2: Hyperplanes, halfspaces, and cells in $d' = 2$

Let n be the cardinality of D. If every data record $\mathbf{r}_i \in D$ is mapped into a hyperplane h_i, the n produced hyperplanes define an arrangement Γ that partitions the preference space into $O(n^{d'})$ cells [3]. Each of the cells falls inside a total of n positive and negative halfspaces. The number of positive halfspaces (incremented by 1) defines the rank of the focal record for any weight vector that falls inside the cell.

Formally, we say that a halfspace, say h^+, covers a cell c if $h^+ \cap c = c$ and denote this as $h^+ \succ c$. Lemma 1 follows directly from the definition of positive and negative halfspaces.

LEMMA 1. *Let c be a cell in Γ. If the weight vector falls in c, the records that score higher than **p** correspond to the positive halfspaces that cover c. The rank of **p** in c is equal to their number plus 1, i.e.,*

$$Rank(c) = 1 + \texttt{COUNT}\{h_i \in \Gamma : h_i^+ \succ c\}$$

To exemplify, Figure 2(b) demonstrates the arrangement produced by two data records, $\mathbf{r}_1, \mathbf{r}_2$, and their respective hyperplanes, h_1, h_2. The arrangement includes four cells. Cell c_3, for instance, is the intersection of $h_1^- \cap h_2^+$. The only positive halfspace that covers c_3 is h_2^+, thus, only $\mathbf{r}_2$ scores higher than **p** in that cell, and the rank of **p** in c_3 is $Rank(c_3) = 1 + 1 = 2$. In summary, the rank of **p** in cells c_1, c_2, c_3, c_4 is 2, 1, 2, 3, respectively.

Based on Lemma 1, we could solve the kSPR problem by mapping each data record into a hyperplane and reporting the cells of the arrangement where the rank of **p** is no greater than k. This approach, however, is impractical because the best known algorithms for arrangement computation take $O(n^{d'})$ time, which furthermore involves large hidden constants [3].

4. CELL TREE APPROACH

In this section we present our first kSPR method, termed *Cell Tree Approach* (CTA), that constitutes the backbone on which we build our methodology. The main idea in CTA is to map each record $\mathbf{r}_i \in D$ into a hyperplane h_i and to insert the hyperplanes one by one into $CellTree$. The role of $CellTree$ is to incrementally maintain the arrangement Γ as new hyperplanes are inserted. When the mapping is complete, the cells in Γ with $Rank(c) \leq k$ form the kSPR result.

The $CellTree$ is a binary tree with as many levels as hyperplanes inserted so far. The root of the tree corresponds to the entire (transformed) preference space. Assuming that the first inserted hyperplane is h_1, it divides the preference space into two cells. Accordingly, the root is split[2] into two children, corresponding to halfspaces h_1^- and h_1^+ respectively. When the second hyperplane, say h_2, is inserted, the existing cells are further divided. That is, the existing leaves of the tree are split and new leaves are formed. Essentially, each inserted hyperplane introduces a new level to $CellTree$. After insertion of the i-th hyperplane, each leaf of $CellTree$ corresponds to a cell in the arrangement induced by the i hyperplanes.

A general principle in our methodology is to push back as far as possible the expensive computational geometric operations, and to reduce their number to the minimum. Specifically, on the one hand, we concisely and efficiently represent the nodes of $CellTree$, without computing and storing their exact geometry (in Section 4.1). On the other hand, we eliminate *infeasible cells*, i.e., cells with zero extent, still *without* computing their geometry (in Section 4.2). Additionally, we eliminate unpromising cells, i.e., cells c with non-zero extent but $Rank(c)$ greater than k.

Furthermore, in Section 4.3 we make crucial observations on the hyperplane insertion operation, and propose optimizations that vastly improve the performance of CTA.

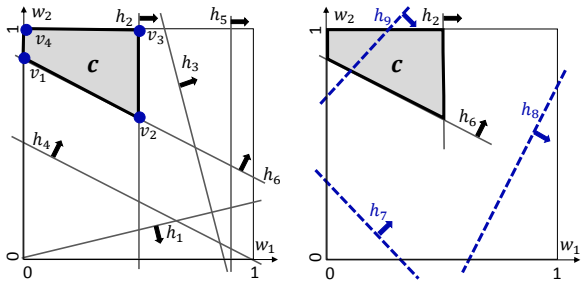
4.1 Cell Representation

Consider the arrangement of i hyperplanes. As described in Section 3.2, each cell is the intersection of i positive and negative halfspaces, and is thus a convex polytope in the (transformed) preference space. Take for example the arrangement of $i = 6$ hyperplanes in Figure 3(a). Cell c is defined as $h_1^- \cap h_2^- \cap h_3^- \cap h_4^+ \cap h_5^- \cap h_6^+$. If the exact geometry of c were to be computed and stored (as a convex polygon with vertices v_1, v_2, v_3, v_4), we would need $O(i^{\lfloor d'/2 \rfloor}) = O(n^{\lfloor d'/2 \rfloor})$ time [10], which is impractical given the large number of cells in $CellTree$.

To avoid expensive halfspace intersection, we represent a cell c implicitly by its set of defining halfspaces $c.\Psi$. For example, in Figure 3(a), we represent cell c by $c.\Psi = \{h_1^-, h_2^-, h_3^-, h_4^+, h_5^-, h_6^+\}$. The main challenge with this implicit representation is that we need a means to detect and eliminate infeasible cells, i.e., cells with zero extent. We address this challenge in Section 4.2.

[1]Although the transformed preference space is visualized for simplicity as a unit hyper-cube, it is actually just part of that cube, because for any weight vector it must be that $\sum_{i=1}^{d-1} w_i < 1$. E.g., in Figure 2(a) it is the part below (diagonal) line $w_1 + w_2 = 1$.

[2]In our context, to split a node means to create two leaves as its children.



(a) Arrangement of 6 hyperplanes (b) Hyperplane insertion

Figure 3: Cell representation and hyperplane insertion

Before we proceed to it, we stress that previous studies that work on the preference space for different problems, like [34] and [23], index the preference domain using a space partitioning method (a Quad-tree, specifically). That approach requires deriving the exact geometry of indexed cells, and may furthermore divide their extent into multiple leaves of the tree, thus replicating their information and wasting computations when re-encountering the same cell in different leaves.

4.2 Detecting Infeasible Cells Efficiently

An infeasible cell is one where the intersection of its defining halfspaces is empty, and it therefore does not appear in the arrangement and should be disregarded. Instead of performing actual halfspace intersection to detect infeasible cells, we use a much faster process. Specifically, we express the defining halfspaces of c as a system of inequalities. In the example of Figure 3(a), cell c involves the following constraints:

$$\begin{cases} h_1^- : S(\boldsymbol{r_1}) = \mathbf{r}_1 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_2^- : S(\boldsymbol{r_2}) = \mathbf{r}_2 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_3^- : S(\boldsymbol{r_3}) = \mathbf{r}_3 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_4^+ : S(\boldsymbol{r_4}) = \mathbf{r}_4 \cdot \mathbf{w} > S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_5^- : S(\boldsymbol{r_5}) = \mathbf{r}_5 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_6^+ : S(\boldsymbol{r_6}) = \mathbf{r}_6 \cdot \mathbf{w} > S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ \forall j \in [1, d-1], w_j \in (0,1); \sum_{j=1}^{d-1} w_j \leq 1 \end{cases} \quad (2)$$

We frame these inequalities as a linear programming (LP) problem with an arbitrary (linear) objective function that involves all weights $w_1, w_2, \cdots, w_{d-1}$, e.g., function $\sum_{j=1}^{d-1} w_j$. We solve the problem with an LP solver, such as lp_solve [1], and if it returns no result we infer that the cell is infeasible.

The time complexity of this feasibility test is linear to the number of hyperplanes i. Specifically, it takes $O(\alpha \cdot i)$ time, where $\alpha = \beta^{d'} \cdot d'!$ and β is a constant [5]. This is a major improvement compared to $O(i^{\lfloor d'/2 \rfloor})$ time required for halfspace intersection. However, we do not stop here. We further optimize the performance of the feasibility test based on a crucial observation.

The cost of the test depends on the number of halfspaces in $c.\Psi$. We reduce that cost by ignoring inconsequential halfspaces and removing their corresponding inequalities from the LP formulation. Consider again cell c in Figure 3(a), where $c.\Psi = \{h_1^-, h_2^-, h_3^-, h_4^+, h_5^-, h_6^+\}$. Although there are 6 defining halfspaces, only 2 of them determine its extent, i.e., h_2^- and h_6^+. We refer to these halfspaces as the *bounding* halfspaces. We may therefore ignore the inequalities for all the remaining (i.e., inconsequential) halfspaces and simplify the LP formulation to reflect only the bounding ones. The equivalent, but easier to solve, LP problem involves only the following inequalities:

$$\begin{cases} h_2^- : S(\boldsymbol{r_2}) = \mathbf{r}_2 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ h_6^+ : S(\boldsymbol{r_6}) = \mathbf{r}_6 \cdot \mathbf{w} > S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ \forall j \in [1, d-1], w_j \in (0,1); \sum_{j=1}^{d-1} w_j \leq 1 \end{cases} \quad (3)$$

To determine the bounding halfspaces is a tough problem, whose solution easily outweighs the gains of reducing the number of inequalities. Instead of computing the bounding halfspaces at the time of the feasibility test, we identify and rule out inconsequential halfspaces within our hyperplane insertion algorithm *without any additional computational cost*, as we discuss in Section 4.3.1.

An important remark is that all our kSPR algorithms, after identifying the cells that belong to the result, perform a finalization step to derive the exact geometry of each result cell by intersecting its defining halfspaces (ignoring the inconsequential ones). This is the only stage where we compute exact geometries. The derived kSPR regions are in the transformed preference space. If they are required in the original space, we may perform halfspace intersection in the original space instead; each halfspace corresponds to a record $\mathbf{r}$, and inequality $S(\mathbf{r}) > S(\mathbf{p})$ (or $S(\mathbf{r}) < S(\mathbf{p})$) can be mapped to a halfspace in either the original or the transformed space.

4.3 Hyperplane Insertion

The $CellTree$ is updated incrementally by inserting hyperplanes one by one. The efficiency of the insertion algorithm is essential to the performance of CTA. Thus, we elaborate on the insertion process and propose enhancements.

There are two alternatives in inserting a new hyperplane h_i. One is to directly insert h_i into the leaves of $CellTree$. Another is to perform insertion top-down. We choose the latter because (i) the number of leaves is very large (i.e., $O(i^{d'})$ [3]), (ii) determining containment at an internal node implies containment of the entire subtree rooted at it, and (iii) if an internal node has already a rank of k, we may already prune its entire subtree[3]. Therefore, the insertion process starts from the root of $CellTree$ and proceeds recursively to its children.

Each node corresponds to a region in the preference space. However, we do not explicitly compute or store that region. Instead, we use the representation technique in Section 4.1. At every node N of the tree (be it internal or leaf) we maintain a *cover set* that is initialized to be empty when the node is first created. Its purpose will become clear shortly. Consider the insertion algorithm for h_i when it runs on an internal node N of the tree. We check the following conditions, using the feasibility test in Section 4.2:

I. IF $N \cap h_i^- = \emptyset$: The node lies completely inside halfspace h_i^+. Add h_i^+ to the cover set of N.
II. ELSE IF $N \cap h_i^+ = \emptyset$: The node is completely inside halfspace h_i^-. Add h_i^- to the cover set of N.
III. ELSE: Hyperplane h_i cuts through node N. Recursively run the insertion algorithm on the children of N.

To exemplify, assume that node N corresponds to the gray region in Figure 3(b). The insertions of h_7, h_8, and h_9 fall under cases I, II, and III, respectively. Note that our feasibility test allows to determine each case without deriving the exact geometry of N.

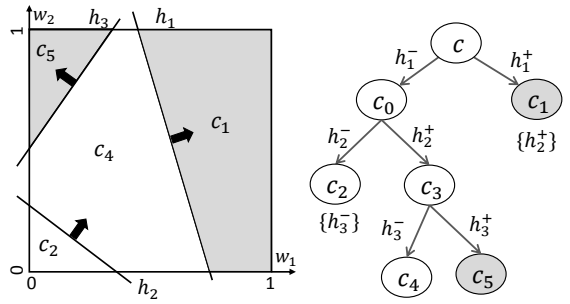
Returning to the insertion process, in cases I and II there is no need to invoke the insertion algorithm on the children of N, because its entire subtree is guaranteed to fall in h_i^+ and h_i^- respectively. We simply add the corresponding halfspace into the cover set of N to record that fact.

[3]The rank of a $CellTree$ node is defined similarly to that of a cell.

The insertion algorithm runs similarly on a leaf node c. The only difference is that in case III we split c, because hyperplane h_i cuts through it. That is, we create two children for c, and label the edges that point to them by h_i^- and h_i^+ respectively. Note that they are both guaranteed to be non-empty, so no feasibility test is required. Their cover sets are initialized as empty sets.

We highlight that the cover set of a node N (be it an internal or leaf node) does not include all the halfspaces that cover it, but a subset of them, i.e., those that were inserted after the node was created. The full set of halfspaces that cover N is the union of (i) its cover set, (ii) the cover sets of all its ancestor nodes, and (iii) all the halfspaces that label the edges of $CellTree$ along the path from the root to node N. By Lemma 1, if the number of positive halfspaces in that full set plus 1 exceeds k, we may safely eliminate N and its entire subtree (if any) because every cell under it is guaranteed to have a rank greater than k. Another situation where we eliminate N is when all the leaves in its subtree have been eliminated.

We provide an example in Figure 4, illustrating the arrangement in preference space and the structure of the tree. Assume that $k = 2$. Insertion of h_1 splits the root into two leaves, c_0 and c_1, for h_1^- and h_1^+ respectively. Next, consider the insertion of h_2. The insertion algorithm is invoked for the root's children, c_0 and c_1. For c_0, we determine that h_2 cuts through it (case III), thus, the leaf is split into two new ones, c_2 and c_3. For c_1, since $h_2^- \cap c_1 = \emptyset$ (case I), h_2^+ is included into c_1's cover set (the cover set is shown right below the node in Figure 4(b)). The rank of c_1 is 3 (i.e., greater than k), as it is already covered by h_1^+ (edge label) and h_2^+ (in its cover set), thus we eliminate it; we draw eliminated (i.e., pruned) nodes in gray. Consider now the insertion of h_3. The insertion algorithm is invoked for the root's only child c_0 (since c_1 was pruned), it determines that h_3 cuts through c_0 (case III), and is thus recursively invoked for its children c_2 and c_3. For c_2, $h_3^+ \cap c_2 = \emptyset$ (case II), hence h_3^- is included into its cover set. For c_3, the leaf is split into two new ones, c_4 and c_5 (case III). The rank of c_5 is already 3 (as it is covered by h_2^+ and h_3^+) and it is pruned.



(a) Arrangement (b) $CellTree$

Figure 4: $CellTree$ **example**

CTA terminates when either all leaves of $CellTree$ are eliminated or when all n hyperplanes have been inserted. In the first scenario, CTA reports an empty set as the kSPR result. In the latter, it reports each leaf (i.e., arrangement cell) whose rank is no larger than k. In Figure 4, assuming that all records in D have been mapped, the kSPR result comprises cells c_2, c_4 with rank 1 and 2, respectively. Algorithm 1 in Appendix E presents the pseudocode of CTA. Below we describe two important optimizations.

4.3.1 Eliminating Inconsequential Halfspaces

In Section 4.2 we explained that the feasibility check for a node can be accelerated by removing inconsequential halfspaces. Such halfspaces can be identified without any extra computations during the hyperplane insertion process. Specifically:

LEMMA 2. *Any halfspace that belongs to the cover set of a node or to the cover set of any of the node's ancestors is inconsequential.*

PROOF. Consider the insertion process for a hyperplane h on a node N (be it an internal or leaf node). In both cases I and II the hyperplane does not alter the existing shape/extent of N and therefore none of h^+ and h^- are bounding halfspaces for N or for any node in its subtree. □

In other words, the only halfspaces that could be bounding for a node N should appear as labels along the path from the root of $CellTree$ to N. These are the only record-induced halfspaces we include in the LP formulation for N. In Figure 4, for instance, only halfspaces h_1^-, h_2^- could be bounding for cell c_2, although it is covered by h_3^- too. Note that we have no guarantee that all the halfspaces that label the path are indeed bounding. E.g., in reality c_2 is only bounded by h_2^-, even though label h_1^- also appears along the path from the root. Even as such, i.e., by using a superset of the actual bounding halfspaces, our technique eliminates more than 96.5% of the defining halfspaces as inconsequential, thus offering one to two orders of magnitude speed-up to the feasibility test routine, as we show in the experiments.

4.3.2 Reducing the Number of Feasibility Tests

We employ a technique that utilizes the results of past feasibility tests in order to reduce the number of subsequent ones. Assume that during the insertion of a hyperplane, we perform a feasibility test on node N in order to check the conditions in case I or II, and that the LP solver reports that (the problem is feasible and that) the objective function is maximized at vector $\mathbf{w}^*$. Clearly, $\mathbf{w}^*$ falls in N. We record $\mathbf{w}^*$ for the very first feasible LP problem that was run on node N.

Consider now the subsequent insertion of another hyperplane h_i where the insertion algorithm needs to check conditions in cases I and II for N. In just $O(d)$ time we may determine whether $\mathbf{w}^*$ falls in h_i^- or h_i^+. If it falls in h_i^-, the condition in case I is guaranteed to be false (thus, saving the cost for the feasibility test $N \cap h_i^- = \emptyset$). Similarly, if $\mathbf{w}^*$ is in h_i^+, the condition in case II is surely false.

4.4 Complexity Analysis

LEMMA 3. *The time complexity of* CTA *is $O(\alpha \cdot n^d)$, where α is a constant depending only on the dimensionality d.*

PROOF. The majority of $CellTree$ nodes are in the leaf level, and the computational cost of CTA is dominated by execution of the insertion algorithm on the leaves. Consider the insertion of the i-th hyperplane. On the assumption that cells (leaves) which are not divided by h_i have already been dealt with by inclusion of h_i^- or h_i^+ in the cover set of an ancestor node, the cost is determined by the feasibility tests required for leaves in case III, i.e., those that h_i cuts through. By the zone theorem [14], the number of case III leaves is $O(i^{d'-1})$. For each of them, the cost of feasibility test is $O(\alpha \cdot i)$. Thus, the total cost for the insertion of h_i is $O(\alpha \cdot i^{d'})$. Since CTA inserts up to n hyperplanes, the overall time complexity is $\sum_{i=1}^{n} O(\alpha \cdot i^{d'}) \leq O(n \cdot \alpha \cdot n^{d'}) = O(\alpha \cdot n^d)$. □

5. PROGRESSIVE CTA

In this section we describe the *Progressive Cell Tree Approach* (P-CTA). This algorithm saves computations by (i) controlling the processing order of the records in D, i.e., the order in which their hyperplanes are inserted into $CellTree$, (ii) ignoring records that

cannot affect the kSPR result, and (iii) accelerating the insertion algorithm based on crucial observations.

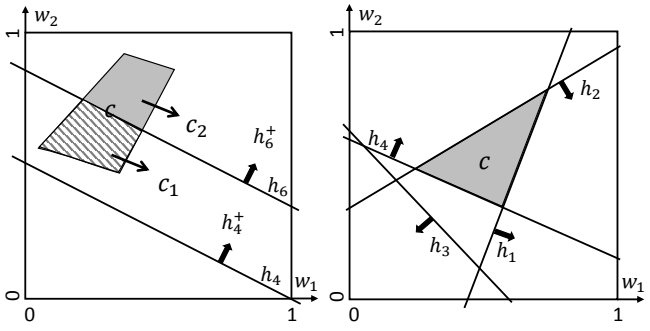
The basic CTA iteratively inserts hyperplanes into $CellTree$. During this process, it eliminates nodes (i.e., parts of the preference space) whose rank exceeds k. To achieve earlier pruning of unpromising nodes, and thus avoid unnecessary hyperplane insertions into them, P-CTA prioritizes the processing order of records so that those with higher pruning potential are processed first. Specifically, if the positive halfspace of record $\mathbf{r}_i$ covers that of $\mathbf{r}_j$ (i.e., $h_i^+ \succ h_j^+$), then $\mathbf{r}_i$ will increase the rank of more nodes, and should therefore be processed before $\mathbf{r}_j$. Determining containment among the different positive halfspaces, however, is too expensive to be practical. To avoid that cost but still effectively prioritize the processing order of the records, we use Lemma 4.

LEMMA 4. *If record $\mathbf{r}_i$ dominates record $\mathbf{r}_j$, then $h_i^+ \succ h_j^+$. Equivalently, it holds that $h_j^- \succ h_i^-$.*

PROOF. For any weight vector $\mathbf{w}$ in h_j^+ it holds that $S(\mathbf{r}_j) > S(\mathbf{p})$. Since $\mathbf{r}_i$ dominates $\mathbf{r}_j$, it also holds that $S(\mathbf{r}_i) > S(\mathbf{r}_j)$. Hence, $S(\mathbf{r}_i) > S(\mathbf{p})$, i.e., $\mathbf{w}$ must also be inside h_i^+, which proves that $h_i^+ \succ h_j^+$. In turn, $h_i^+ \succ h_j^+ \Leftrightarrow h_j^- \succ h_i^-$ □

Consider Figure 5(a) and assume that $\mathbf{r}_4$ dominates $\mathbf{r}_6$. Lemma 4 implies that h_4^+ covers h_6^+. If we needed to decide which one between $\mathbf{r}_4$ and $\mathbf{r}_6$ to process next, it should be $\mathbf{r}_4$. To exemplify, processing $\mathbf{r}_4$ would increase the rank of cell c in the figure (and thus it would expedite its possible pruning), whereas processing $\mathbf{r}_6$ would split the cell (i.e., it would prematurely grow the tree and, hence, increase the cost of subsequent operations on it). Based on Lemma 4, P-CTA establishes the following invariant.

INVARIANT 1. *A record will only be processed if all the records that dominate it have already been processed.*



(a) Dominance and containment (b) Defining the pivots of c

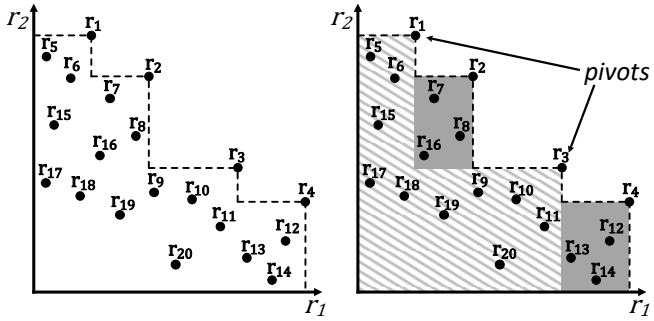
Figure 5: Ideas behind P-CTA ($d' = 2, k = 3$)

To uphold the invariant, P-CTA processes in a first batch (yet, still one by one) the records that belong to the skyline of D. The question now is which records should be processed next. Consider Figure 5(b) where the skyline records $\{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4\}$ have already been processed, and c is a promising cell in the resulting $CellTree$, i.e., it has $Rank(c) \leq k$. The set of defining/covering halfspaces for c is $c.\Psi = \{h_1^-, h_2^+, h_3^-, h_4^+\}$. We call *pivots* of c those processed records that contribute negative halfspaces to $c.\Psi$. In our example, the pivots of c are $\mathbf{r}_1$ and $\mathbf{r}_3$.

LEMMA 5. *Any unprocessed record $\mathbf{r}$ that is dominated by a pivot of c has no effect on the rank or extent of c.*

PROOF. Let $\mathbf{r}_j$ be a pivot of c that dominates $\mathbf{r}$. Lemma 4 suggests that $h^- \succ h_j^-$. Since $\mathbf{r}_j$ is pivot to c, it also holds that $h_j^- \succ c$. Thus, $h^- \succ c$, i.e., h does not cut through c and $\mathbf{r}$ does not affect the rank or the extent of c. □

To illustrate, Figure 6 continues the example of Figure 5(b) (where $d = 3$) but for ease of illustration it assumes 2-dimensional data. Figure 6(a) shows the skyline of D in data space. Figure 6(b) shows as striped the area dominated by the pivots of c, i.e., by $\mathbf{r}_1$ and $\mathbf{r}_3$. Lemma 5 suggests that records in the striped area have no effect on c. Hence, the only unprocessed records that could affect the rank and extent of c lie inside the gray regions. If these regions are empty, we can directly report c as part of the kSPR result. This is an important finding, because it enables major computation savings. Also, it renders P-CTA progressive, i.e., it allows the reporting of result regions before the algorithm terminates. That is a highly desirable property in preference-based querying [29].



(a) First batch (b) Second batch

Figure 6: Determining records to process next ($d = 2, k = 3$)

If the gray regions are not empty, and in order to uphold Invariant 1, we process next those among the gray region records that are not dominated by any unprocessed record, i.e., the second batch includes $\mathbf{r}_7, \mathbf{r}_8, \mathbf{r}_{12}$. Formally, these are the unprocessed records that belong to the skyline of D if we ignore non-pivot records ($\mathbf{r}_2, \mathbf{r}_4$).

In general, there are multiple promising cells. It is impractical to process individual batches of records for each of them. To determine a universal next batch to process, we compute the union of the non-pivot records for all promising cells, and recompute the skyline of D by ignoring the records that belong to that union. The unprocessed records in the new skyline form the next batch. In the example of Figure 6, assume that, in addition to c, there is another promising cell with pivots $\mathbf{r}_2$ and $\mathbf{r}_3$. In this case, the union of non-pivot records for the two cells is $\{\mathbf{r}_2, \mathbf{r}_4\} \cup \{\mathbf{r}_1, \mathbf{r}_4\} = \{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_4\}$. The recomputed skyline ignores that union, and comprises $\mathbf{r}_5, \mathbf{r}_6, \mathbf{r}_7, \mathbf{r}_8, \mathbf{r}_3, \mathbf{r}_{12}$. The unprocessed among them (i.e., all except $\mathbf{r}_3$) form the second batch. New batches of records are processed until all cells in $CellTree$ are either eliminated or reported.

The original skyline computation (for the first batch) as well as skyline recomputation (required for subsequent batches) can be performed using the index on D and the incremental *branch-and-bound skyline* (BBS) technique in [25]. Importantly, as new records are fetched and processed, we maintain in a dominance graph all the dominance relationships between processed records. This graph serves as a look-up structure, used to accelerate the insertion algorithm in Section 4.3. Specifically, when we insert hyperplane h_i into a node N (be it internal or leaf), we first look into the dominance graph to get the set of already processed records that dominate $\mathbf{r}_i$. If any of them contributes a negative halfspace to the cover set of N, we determine that h_i^- covers N (i.e., it is case II) and add h_i^- directly to the cover set of N, without any further checking. The reasoning behind this optimization is similar to

Lemma 5. Algorithm 2 in Appendix E summarizes the complete P-CTA algorithm. Lemma 6 is key for its complexity analysis.

LEMMA 6. *P-CTA will never process a record that is dominated by k or more other records in D.*

PROOF. Consider a record $\mathbf{r}$ that is dominated by a set D_r of other records, where $|D_r| \geq k$ ($|\cdot|$ denotes cardinality). For $\mathbf{r}$ to be processed, there must be at least one promising cell c such that $\mathbf{r}$ is not dominated by any of the cell's pivots. By Invariant 1, all records in D_r must have already been processed before $\mathbf{r}$. Therefore, each D_r record contributes a halfspace (positive or negative) in the $c.\Psi$ set. For $\mathbf{r}$ to be processed on behalf of c, none of the D_r records can be pivots, thus their contributing halfspaces are all positive. That is, $c.\Psi$ includes at least k positive halfspaces, i.e., $Rank(c) > k$, which contradicts the assumption that c is promising. □

COROLLARY 1. *The time complexity of P-CTA is $O(\alpha \cdot (k\frac{\log^{d-1} n}{d!})^d)$, where α is a constant depending only on the dimensionality d.*

PROOF. Assuming independent and uniformly distributed data records, the number of those that are dominated by none or fewer than k others is $O(k\frac{\log^{d-1} n}{d!})$ [17]. Since P-CTA processes a subset of these records, we derive its complexity by plugging that number into Lemma 3 instead of n. □

On a different note, Lemma 6 (with small modifications) suggests that a plausible kSPR solution is to compute all records in D that are dominated by none or fewer than k others, i.e., to compute what is commonly referred to as the k-skyband [25] of D, and feed them to CTA. As we show in Appendix B, the k-skyband is a large superset of the records processed by P-CTA, resulting in 4 to 9 times slower processing than P-CTA.

6. LOOK-AHEAD P-CTA

To further boost the performance of P-CTA, in this section we propose look-ahead techniques that enable (i) the early pruning of unpromising cells and (ii) the early detection of cells that belong to the kSPR result. We term the produced method *Look-ahead Progressive Cell Tree Approach* (LP-CTA).

6.1 Fundamental Idea

A cell c corresponds to a collection of weight vectors, which can produce a range of scores for the focal record $\mathbf{p}$. We denote the minimum and maximum possible score of $\mathbf{p}$ for any weight vector in c as $\underline{S}(\mathbf{p}, c)$ and $\overline{S}(\mathbf{p}, c)$, respectively. We can accurately compute $\underline{S}(\mathbf{p}, c)$ by solving an LP problem for minimizing $S(\mathbf{p})$ subject to the constraints that define cell c. To accelerate LP solving, we use Lemma 2 to remove inconsequential halfspaces (constraints), exactly as described in Section 4.3.1. Recall that c is in the transformed preference space (with axes $w_1, w_2, \cdots, w_{d-1}$), thus, the objective function $S(\mathbf{p})$ is expressed as $p_d + \sum_{i=1}^{d-1}(p_i - p_d)w_i$.

To demonstrate, if the cell at hand is cell c_2 in Figure 4, it is represented by halfspaces h_1^- and h_2^-, i.e., only those that appear as labels along the path from the root of $CellTree$ to leaf c_2. By combining the corresponding constraints and those that define the boundaries of the preference space, $\underline{S}(\mathbf{p}, c)$ is derived as the optimal value of the objective in the following LP problem:

$$\begin{aligned} \text{Minimize:} \quad & p_d + \textstyle\sum_{i=1}^{d-1}(p_i - p_d)w_i \\ \text{subject to:} \quad & h_1^- : S(\mathbf{r}_1) = \mathbf{r}_1 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ & h_2^- : S(\mathbf{r}_2) = \mathbf{r}_2 \cdot \mathbf{w} < S(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w} \\ & \forall j \in [1, d-1], w_j \in (0,1); \textstyle\sum_{j=1}^{d-1} w_j \leq 1 \end{aligned} \tag{4}$$

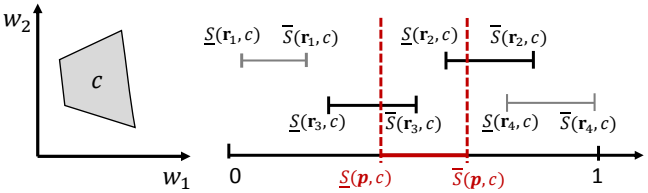
$\overline{S}(\mathbf{p}, c)$ is derived by solving an LP problem with the same constraints, but where the objective function $S(\mathbf{p})$ is maximized.

Similarly, we could compute the minimum and maximum score of any record $\mathbf{r} \in D$ for weight vectors in c (as $\underline{S}(\mathbf{r}, c)$ and $\overline{S}(\mathbf{r}, c)$), and derive a lower and upper bound for the rank of $\mathbf{p}$ in c as:

$$\begin{aligned} \underline{Rank}(c) &= 1 + \texttt{COUNT}\{\mathbf{r} \in D : \underline{S}(\mathbf{r}, c) > \overline{S}(\mathbf{p}, c)\} \\ \overline{Rank}(c) &= 1 + \texttt{COUNT}\{\mathbf{r} \in D : \overline{S}(\mathbf{r}, c) > \underline{S}(\mathbf{p}, c)\} \end{aligned} \tag{5}$$

To avoid confusion about the role of the lower and upper bound, we stress that the lower bound $\underline{Rank}(c)$ is the best rank that $\mathbf{p}$ could achieve in c, and $\overline{Rank}(c)$ is the worst, i.e., $\underline{Rank}(c) \leq \overline{Rank}(c)$. Note that the bounds are defined over all records in D and are, thus, irrelevant to which or how many records have been processed so far (i.e., mapped into hyperplanes and reflected in the $CellTree$).

Consider cell c in Figure 7(a) and assume that $D = \{\mathbf{r}_1, \cdots, \mathbf{r}_4\}$. In Figure 7(b) each data record $\mathbf{r}$ is mapped to a score interval $[\underline{S}(\mathbf{r}, c), \overline{S}(\mathbf{r}, c)]$ corresponding to the possible scores $\mathbf{r}$ could achieve in c. Focal record $\mathbf{p}$ is similarly mapped to $[\underline{S}(\mathbf{p}, c), \overline{S}(\mathbf{p}, c)]$. Based on these intervals, the best rank achievable by $\mathbf{p}$ in c is $\underline{Rank}(c) = 2$ and the worst is $\overline{Rank}(c) = 4$.



(a) Candidate cell (b) Score intervals

Figure 7: Deriving rank bounds for a cell

The rank bounds enable the early detection (i) of unpromising cells and (ii) of cells that definitely belong to the kSPR result. First, if $\underline{Rank}(c) > k$, we may safely prune c. Second, if $\overline{Rank}(c) \leq k$, we directly include c in the kSPR result[4]. In either scenario, c is ignored by subsequent traversals/operations in $CellTree$.

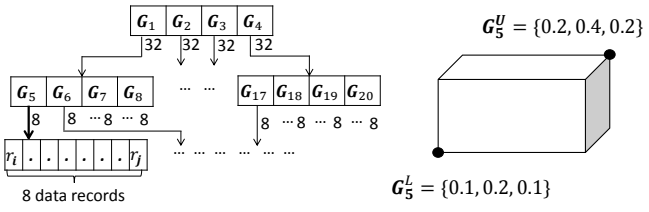
Clearly, for large datasets it is impractical to compute $\underline{S}(\mathbf{r}, c)$ and $\overline{S}(\mathbf{r}, c)$ for every $\mathbf{r} \in D$. In Section 6.2 we utilize the index on D to derive score bounds for entire groups of records, which in turn accelerate the computation of $\underline{Rank}(c)$ and $\overline{Rank}(c)$.

6.2 Group Bounds

Suppose that we organize D with an aggregate spatial index, such as the aggregate R-tree [24]. This is a regular R-tree where, in the internal nodes, each entry represents a group $\mathbf{G}$ of records and stores (i) the minimum bounding rectangle ($\mathbf{G}$.mbr) and (ii) the number of data records in its subtree ($\mathbf{G}$.num). Figure 8(a) illustrates an aggregate R-tree. Figure 8(b) shows the entry of group $\mathbf{G}_5$ in the data space ($d = 3$). The entry has $\mathbf{G}_5.\text{mbr} = ([0.1, 0.2], [0.2, 0.4], [0.1, 0.2])$ and $\mathbf{G}_5.\text{num} = 8$.

We call *min-corner* of $\mathbf{G}$.mbr its corner with the minimum coordinates and denote it as $\mathbf{G}^L$. Symmetrically, the *max-corner* $\mathbf{G}^U$ is the corner with the maximum coordinates. In the case of $\mathbf{G}_5$ above, $\mathbf{G}_5^L = (0.1, 0.2, 0.1)$ and $\mathbf{G}_5^U = (0.2, 0.4, 0.2)$. The min-corner and max-corner of an R-tree entry $\mathbf{G}$ can be used to derive score bounds for any record under it. Specifically, due to the increasing monotonicity of the scoring function $S(\cdot)$ (Equation 1), for any

[4]Note that in CTA and P-CTA, cell c could be split during the insertion of new hyperplanes, only to eventually include all the produced parts (as separate cells) into the kSPR result.



(a) Aggregate R-tree (b) MBR of entry $\mathbf{G}_5$

Figure 8: Computing group bounds

record $\mathbf{r}$ in the subtree of $\mathbf{G}$, and for any weight vector, it holds that $S(\mathbf{G}^L) \leq S(\mathbf{r}) \leq S(\mathbf{G}^U)$. Thus, for weight vectors in a specific cell c, it holds that $\underline{S}(\mathbf{G}^L, c) \leq S(\mathbf{r}) \leq \overline{S}(\mathbf{G}^U, c)$; the group bounds $\underline{S}(\mathbf{G}^L, c)$ and $\overline{S}(\mathbf{G}^U, c)$ are derived by solving an LP problem for each, as in Section 6.1. In particular, for the former we minimize the objective function $S(\mathbf{G}^L)$ and for the latter we maximize objective function $S(\mathbf{G}^U)$, both subject to the constraints imposed by cell c and the boundaries of the preference space.

Having defined the group bounds, we can now utilize the aggregate R-tree on D to derive the rank bounds for a cell c. We initialize $\underline{Rank}(c) = \overline{Rank}(c) = 1$, and traverse the index in a top-down fashion, starting from the root. When we examine a non-leaf entry $\mathbf{G}$, we compare the group bounds of $\mathbf{G}$ with the score bounds of the focal record $\mathbf{p}$. If $\overline{S}(\mathbf{G}^U, c) < \underline{S}(\mathbf{p}, c)$, we ignore the subtree of $\mathbf{G}$. If $\underline{S}(\mathbf{G}^L, c) > \overline{S}(\mathbf{p}, c)$, we increment $\underline{Rank}(c)$ and $\overline{Rank}(c)$ by $\mathbf{G}$.num and ignore the subtree of $\mathbf{G}$. If interval $[\underline{S}(\mathbf{p}, c), \overline{S}(\mathbf{p}, c)]$ completely covers interval $[\underline{S}(\mathbf{G}^L, c), \overline{S}(\mathbf{G}^U, c)]$, we increment $\overline{Rank}(c)$ by $\mathbf{G}$.num and ignore the subtree of $\mathbf{G}$, because even if we go deeper in that subtree, the score intervals of all underlying records are guaranteed to be completely covered by the score interval of $\mathbf{p}$. In all other cases, we visit the node pointed by $\mathbf{G}$ and perform the same process on its own entries recursively. When the traversal encounters records, we apply the same reasoning, but we use $\underline{S}(\mathbf{r}, c)$ and $\overline{S}(\mathbf{r}, c)$ instead of the group bounds.

While effective (in the early pruning and reporting of cells), the above process computes $\underline{S}(\cdot, c)$ and $\overline{S}(\cdot, c)$ bounds for many entries and records in the aggregate R-tree, each requiring an expensive call to the LP solver. This is a major issue, especially considering that we need to compute rank bounds for a large number of different cells. In Section 6.3 we reduce the number of calls to the LP solver, without sacrificing effectiveness.

6.3 Fast Bounds

We propose bounds that are faster to compute than $\underline{S}(\cdot, c)$ and $\overline{S}(\cdot, c)$ presented previously. These bounds are looser, and hence applied in tandem with, yet before we resort to expensive $\underline{S}(\cdot, c)$ and $\overline{S}(\cdot, c)$ computation, in a filter-and-refine fashion.

For a cell c, we can compute its *min-vector* $\mathbf{w}^L$ as a weight vector (in the original, d-dimensional preference space) such that the score of any record $\mathbf{r}$ according to $\mathbf{w}^L$ is no larger than the score of $\mathbf{r}$ according to any weight vector in c. The *max-vector* $\mathbf{w}^U$ of c plays the symmetric role of awarding to any record $\mathbf{r}$ a score no smaller than any weight vector in c.

We derive the min-vector of c as follows. First, we compute the minimum possible value for each of $w_1, w_2, \cdots, w_{d-1}$ by solving an LP problem for each of them, subject to the constraints that define c. Then, we compute the minimum value for w_d by solving a similar LP problem with objective $w_d = 1 - \sum_{i=1}^{d-1} w_i$. The minimum w_i values derived by these (d in total) LP problems comprise the min-vector $\mathbf{w}^L$. The score of any record in a group $\mathbf{G}$ is lower bounded by the score of the min-corner $\mathbf{G}^L$ according to $\mathbf{w}^L$. We call the latter the fast lower bound and denote it as $\underline{S}^{fast}(\mathbf{G}, c)$.

The max-vector $\mathbf{w}^U$ of c is computed by solving the same LP problems as for $\mathbf{w}^L$, but here the objectives are maximized. For a group $\mathbf{G}$, the fast upper bound $\overline{S}^{fast}(\mathbf{G}, c)$ is derived as the score of $\mathbf{G}^U$ according to $\mathbf{w}^U$. Fast bounds can be similarly computed for data records, using the same vectors $\mathbf{w}^L$ and $\mathbf{w}^U$.

With the fast bounds, we can accelerate the rank bound computation in Section 6.2 as follows. For any entry $\mathbf{G}$ considered during the traversal of the data index, we first apply the fast bounds to obtain the score interval $[\underline{S}^{fast}(\mathbf{G}, c), \overline{S}^{fast}(\mathbf{G}, c)]$. If this score interval does not overlap with (or is completely covered by) the score interval of $\mathbf{p}$, we avoid computing the expensive group bounds $\underline{S}(\mathbf{G}^L, c)$ and $\overline{S}(\mathbf{G}^U, c)$. When we encounter data records, we use the fast bounds similarly as filtering step, and only resort to the expensive $\underline{S}(\mathbf{r}, c)$ and $\overline{S}(\mathbf{r}, c)$ if the fast filtering step is inconclusive.

Importantly, we call the new bounds fast, because vectors $\mathbf{w}^L$ and $\mathbf{w}^U$ are computed once per cell c, and are reused to derive score intervals for any entry or record encountered in the data index during the rank bound computation for c. With $\mathbf{w}^L$ and $\mathbf{w}^U$ at hand, each $\underline{S}^{fast}(\cdot, c)$ and $\overline{S}^{fast}(\cdot, c)$ value can be derived in $O(d)$ time. Juxtapose this to the need for two LP calls for each and every encountered entry or record in Section 6.2, at the cost of $O(\alpha \cdot i)$ per LP call (where i is the number of constraints that define c). Our evaluation in Section 7 demonstrates that the use of fast bounds as filters reduces the running time of LP-CTA by up to 64%.

6.4 Putting it All Together

We have described and optimized a process to derive rank bounds for a given cell c. The question now is which cells to compute these bounds for. A possible strategy is to apply them to both created cells whenever a leaf of $CellTree$ is split. An alternative is to compute the rank bounds of newly created cells, i.e., new leaves in $CellTree$, after processing an entire batch of records (recall that LP-CTA, just like P-CTA, fetches the new records to process in batches). We found empirically that the second strategy leads consistently to faster processing. Algorithm 3 in Appendix E summarizes LP-CTA.

The time complexity of LP-CTA is the same as P-CTA in Corollary 1. The reason is that, in the worst case, the look-ahead techniques will fail to quickly prune or report any cells in $CellTree$. At the same time, the overall time complexity is dominated by the operations required within the $CellTree$ (as opposed to look-ahead computations). Despite the common asymptotic complexity of the two algorithms, in practice LP-CTA is two times to an order of magnitude faster than P-CTA, as we show in the experiments.

7. EXPERIMENTAL EVALUATION

In this section we present our empirical findings. In Section 7.1 we describe the experimental setting. In Section 7.2 we conduct a case study on a real dataset to demonstrate the applicability of the kSPR problem. In Section 7.3 we evaluate the performance of our solutions on real and synthetic datasets. Finally, in Section 7.4 we investigate the effectiveness of independent optimizations.

7.1 Experimental Setting

We use the standard synthetic benchmarks for preference-based queries [7], namely *Independent* (IND), *Correlated* (COR), and *Anti-correlated* (ANTI), which represent typical data distributions in multi-criteria decision making applications. We also experiment with real datasets HOTEL, HOUSE, and NBA, which are commonly used in the rank-aware processing literature. HOTEL includes 418K 4-dimensional records that correspond to hotels. HOUSE comprises 315K 6-dimensional records, each correspond-

ing to an American family and its spendings in 6 types of expenses. NBA contains 22K records of statistics for NBA players, where 8 attributes are suitable for rank-aware processing [18]. Table 1 provides details on the size, attributes, and source of the real datasets.

Dataset	d	n	Attributes	Source
HOTEL	4	418,843	No. of stars, Price, No. of rooms, No. of facilities	hotels-base.com
HOUSE	6	315,265	Gas, Electricity, Water, Heating, Insurance, Property tax	ipums.org
NBA	8	21,960	Games, Rebounds, Assists, Steals, Blocks, Turnovers, Personal fouls, Points	basketball-reference.com

Table 1: Real dataset information

We index each dataset with an aggregate R-tree[5]. Data and index are kept in memory, although in Appendix A we also present results for the scenario where they are stored on the disk. The main performance factor is execution time, but for most experiments we also include side metrics that offer insight into the problem and methods. Table 2 provides the value ranges for each tested parameter, indicating their default values in bold. To isolate the effect of a parameter, we vary it while keeping the remaining ones to their defaults. Each plotted value corresponds to the average of measurements observed across 1000 queries (focal records) randomly selected from the corresponding dataset. The response times reported for our algorithms include the finalization step of deriving the exact geometry of each result cell by halfspace intersection (using the standard qhull library [2]).

Parameter	Tested and default values
Dataset cardinality (n)	100K, **500K**, 1M, 5M, 10M
Dimensionality (d)	2, 3, **4** , 5, 6, 7
Value k	10, **30**, 50, 70, 90

Table 2: Experiment parameters, tested values, and defaults

By default, CTA, P-CTA, and LP-CTA are equipped with all applicable optimizations. All methods were implemented in C++, using lp_solve [1] as LP solver. The experiments run on a PC with Intel i7-4770 3.40GHz CPU, 8GB DDR3 RAM, and 256GB SSD.

7.2 Case Study

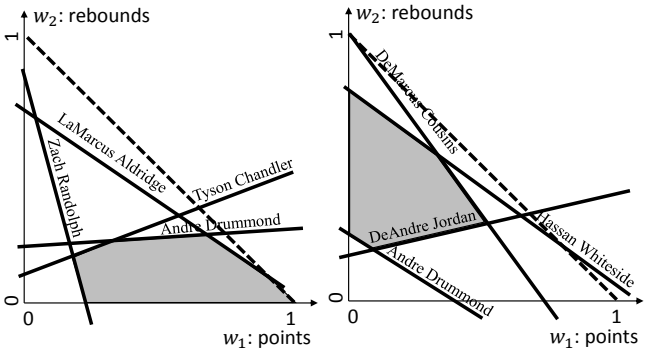
To demonstrate the usefulness of kSPR, we conduct a case study in the context of competitive sports. We use two fractions of the NBA dataset, namely the 2014-2015 and the 2015-2016 season statistics, and three attributes (points, rebounds, and assists). Imagine that a manager is looking to market a player whose contract has expired. Our query could help the manager figure how to frame the player's characteristics to make him appear the most competitive.

In this case study, we set $k = 3$ and use as focal record **p** player *Dwight Howard*, who plays the 'Center' position. Figures 9(a) and 9(b) visualize the kSPR regions of **p** in the 2014-2015 and 2015-2016 seasons, respectively. Note that the (transformed) preference space is triangular, as indicated by the dashed diagonal line[6].

[5]We present the index construction cost in Appendix D.

[6]In the original preference space there is a third dimension, w_3 (weight for assists). The values of w_3 for the kSPR regions in Figure 9 are average, meaning that assists are not as decisive a factor as points and rebounds in *Dwight Howard*'s case. Recall that we can produce the kSPR regions either in the transformed or in the original preference space, as explained at the end of Section 4.2.

Weight w_1 corresponds to the points dimension, representing the significance of the attack capabilities of the players. Weight w_2 pertains to the rebounds dimension, reflecting the significance of their defense abilities. In the 2014-2015 season (Figure 9(a)) *Dwight Howard* is most competitive, i.e., ranks among the top-3 players, when w_1 is high and w_2 is low. Hence, to make him stand out from the competition, it is advisable for his manager to stress his solid attack capabilities. In 2015-2016 (Figure 9(b)) the kSPR region corresponds to preferences with low w_1 and high w_2 values. This suggests that in order to effectively market *Dwight Howard* this year, his manager should put more emphasis on his defense skills.

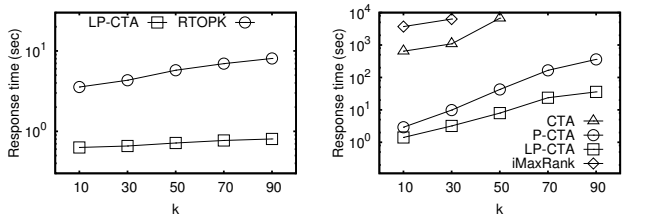


(a) 2014-2015 season (b) 2015-2016 season

Figure 9: kSPR result for *Dwight Howard* (NBA, $k = 3$)

7.3 Performance Evaluation

In Section 2 we mentioned that Vlachou et al. [31] describe a monochromatic reverse top-k method for 2-dimensional data, which essentially solves the kSPR problem for $d = 2$. We denote that method as RTOPK. We implemented it in C++, since in the original paper Java was used. RTOPK only considers data records that are neither dominated by nor dominate the focal record **p** (the same reasoning as in Section 3.1 applies). In Figure 10(a) we compare it with LP-CTA in the special case of 2-dimensional records, using IND data and varying k. LP-CTA is an order of magnitude faster than RTOPK. The reason is that the latter needs to compare **p** with every data record that does not dominate nor is dominated by **p**, and compute a switching value for each of them. In contrast, LP-CTA only considers a small subset of the dataset. E.g., for $k = 30$, RTOPK considers around 600 times more records than LP-CTA. Since RTOPK does not extend to higher dimensions ($d > 2$), we ignore it in the following experiments.



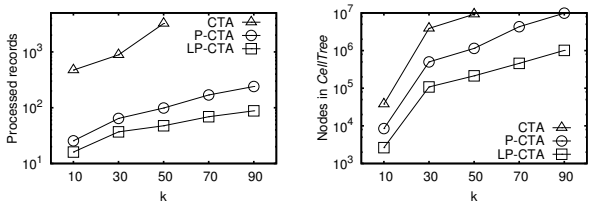
(a) RTOPK [31] (IND, $d = 2$) (b) iMaxRank [23] (IND, $d = 4$)

Figure 10: Comparison with adaptations of previous work

In Section 2 we also mentioned that the incremental version of the maximum rank query [23] can be adapted to solve the kSPR problem. That is, after computing the best rank k^* achievable by **p** (and the corresponding regions in preference space), to in-

crementally probe the algorithm and report the regions for ranks $k^* + 1, k^* + 2$, etc. until k. We denote this method as iMaxRank. We used directly the implementation of [23], which relies on the same data indices and key libraries (e.g., qhull) as our kSPR methods. We compare them in Figure 10(b), where we use IND data and vary k, while setting the remaining parameters to their defaults. iMaxRank is three orders of magnitude slower than P-CTA and LP-CTA, and 6 times slower than our basic approach, CTA. It takes almost 2 hours for $k = 30$ and fails to terminate in reasonable time for $k > 30$. The primary reason for its poor performance is that it executes numerous computational geometric operations, and in particular expensive halfspace intersections. Moreover, it indexes the preference space with a Quad-tree, which leads to a clumsy partitioning, and results in each halfspace cutting through (and thus being intersected with the contents of) many Quad-tree leaves. Due to its unsuitability for the kSPR problem and its inability to scale, we exclude it from subsequent experiments.

Turning to the performance of our kSPR methods, Figure 10(b) offers an indicative comparison. CTA exceeds 2 hours for $k > 50$. On the contrary, P-CTA and LP-CTA scale well, with LP-CTA terminating in a few seconds. Between them, LP-CTA is the clear winner, being from 2 to 10 times faster. To investigate further, for the same experiment, in Figure 11 we plot the number of processed records (equivalently, the number of hyperplanes inserted into $CellTree$) and the number of nodes in $CellTree$ upon termination. The prioritized processing and quick cell reporting in P-CTA leads to 13 to 32 times fewer processed records than CTA, and to 8 times smaller $CellTree$ structure. On top of that, the look-ahead techniques in LP-CTA achieve an additional reduction of up to 3 times in the number of processed records, and up to 9 times in $CellTree$ nodes. We note that the strong point of LP-CTA lies in the early pruning and reporting of cells, which is only indirectly reflected in the numbers of processed records and $CellTree$ nodes.

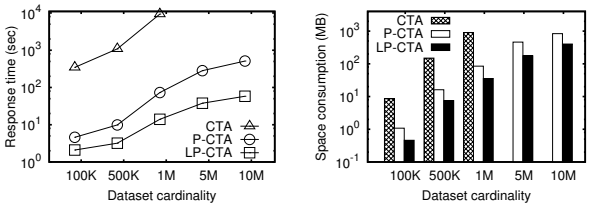


(a) Processed records (b) Nodes in $CellTree$

Figure 11: Effect of k (IND)

In Figure 12 we compare our methods when varying the dataset cardinality n from 100K to 10M. In Figure 12(a) we plot the running time. LP-CTA is clearly ahead of competition, and its gap from the runner-up (P-CTA) widens with n (e.g., it is 9 times faster for $n = 10$M), demonstrating its superior scalability. Figure 12(b) presents the space requirements for the same experiment. These are dominated by the size of $CellTree$ (discussed in the previous experiment in the context of Figure 11(b)), thus, LP-CTA has the smallest overhead, while CTA the largest. Importantly, even for $n = 10$M, LP-CTA requires just 403MB, which is well within the capacity of commodity computers (and it is also the largest space requirement we observed for LP-CTA across all experiments).

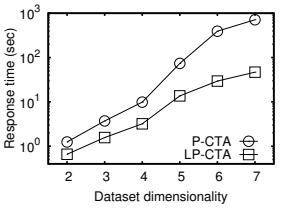
In Figure 13 we vary d from 2 to 7, and present the response time of our two best methods (in Figure 13(a)), and the number of regions in the kSPR result (in Figure 13(b)). The number of kSPR regions, which has to do with the nature of the problem, increases



(a) Response time (b) Space consumption

Figure 12: Effect of n (IND)

quickly with d, hence leading to a similar increase in response time. The rising number of regions is because as dimensionality grows, the records become score-wise less distinguishable [23], thus making it more likely for the focal record to be among the top-k records in more parts of the preference space.



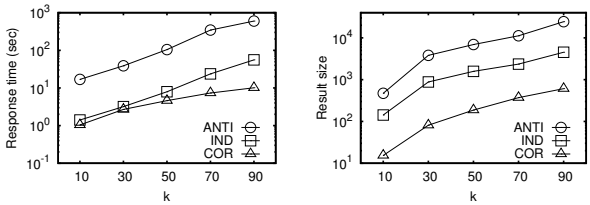
(a) Response time

Dimensionality	Result size
2	18.67
3	432.69
4	879.36
5	3205.74
6	5181.21
7	8326.42

(b) Result size

Figure 13: Effect of d (IND)

Next, we focus on our best method, LP-CTA, and study the effect of data distribution. In Figure 14 we show the response time of LP-CTA and the result size for IND, COR, and ANTI, while varying k. Performance is best for COR and worst for ANTI. This is expected, since in COR records have the highest likelihood to dominate one another [7], thus producing fewer possible top-k results and, in turn, fewer kSPR regions. In contrast, in ANTI records tend to not dominate one another, hence allowing for a multitude of possible top-k results, and therefore more regions in preference space where $\mathbf{p}$ may enter the top-k list.

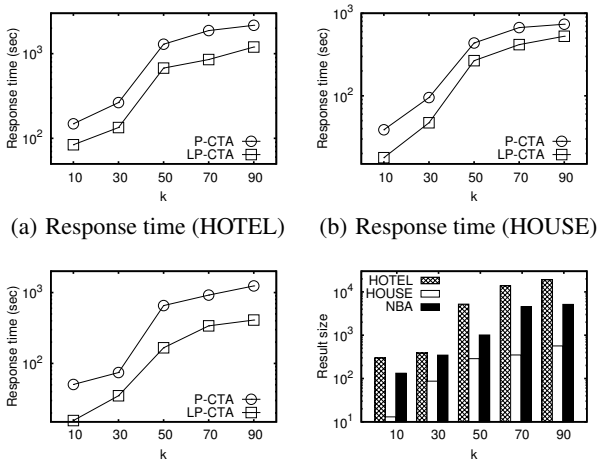


(a) Response time of LP-CTA (b) Result size

Figure 14: Effect of data distribution

In Figure 15 we try real datasets. Figures 15(a), (b), (c) compare P-CTA and LP-CTA for HOTEL, HOUSE, and NBA, when varying k. Figure 15(d) shows the respective numbers of kSPR regions for all three datasets. The relative performance of our methods is the same as in previous experiments. Comparing across datasets, the response times in NBA are similar to those in HOUSE. On the one hand, NBA contains 14 times fewer records than HOUSE, but on the other, its kSPR result includes an order of magnitude more

regions. These factors are canceling out each other's effect, leading to similar running times in the two datasets. The algorithms perform the slowest for HOTEL, which is the largest real dataset and where the number of result regions is significantly higher.



(a) Response time (HOTEL) (b) Response time (HOUSE)

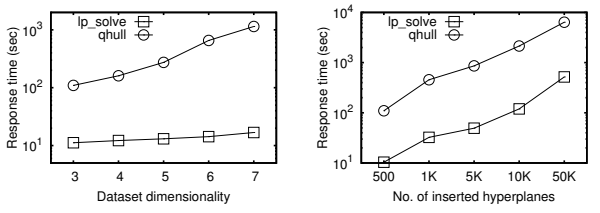
(c) Response time (NBA) (d) Result size

Figure 15: Experiments on real datasets

7.4 Effectiveness of Optimizations

Here we evaluate individual optimizations within our methods. First, we compare our LP-based feasibility test with straightforward halfspace intersection. Then, we assess the benefits derived from the elimination of inconsequential halfspaces. Finally, we compare the efficacy of the different look-ahead bounds.

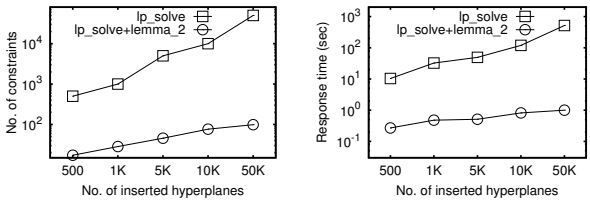
In Figure 16 we pick a number m of IND records that are not dominated nor dominate $\mathbf{p}$, and insert them into $CellTree$. We do not prune any node based on its rank, so as to derive the full arrangement of the m hyperplanes. We then randomly pick 100 leaves of $CellTree$ and perform two different feasibility tests to them: (i) our LP-based test from Section 4.2 (using lp_solve) and (ii) computing the exact geometry of the respective cells via halfspace intersection (using the qhull library). In Figures 16(a) and 16(b) we report the techniques' total response time (for all 100 leaves) by varying the dimensionality d and the number of hyperplanes m. Our LP-based test is 10 to 68 times faster than halfspace intersection. The gap widens with d, because the cost of geometric operations involved in halfspace intersection explodes with d.



(a) Effect of d (b) Effect of m

Figure 16: Effectiveness of LP-based feasibility test

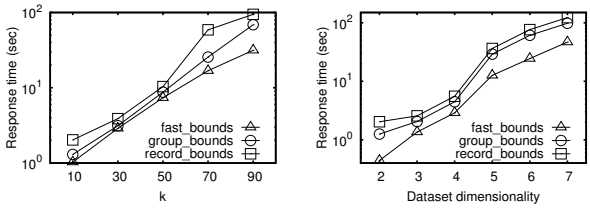
In Figure 17 we investigate the effectiveness of eliminating inconsequential halfspaces by the technique described in Section 4.3.1. As in the previous experiment, we insert m halfspaces into $CellTree$ and perform feasibility tests on 100 randomly chosen leaves by calling the LP solver (i) on the entire set of defining halfspaces (represented as lp_solve in the charts) and (ii) only on the halfspaces that are not deemed inconsequential by Lemma 2 (represented as lp_solve+lemma_2). We plot the average number of constraints and the total running time of the two approaches while varying m. On the average, lp_solve+lemma_2 processes only 17 constraints for $m = 500$, and 98 for $m = 50$K, corresponding to 3.5% and 0.2% of the total number of defining halfspaces. This results in 32 to 517 times faster feasibility testing, and confirms the effectiveness of our halfspace elimination technique.



(a) No. of constraints (b) Response time

Figure 17: Effectiveness of Lemma 2

Finally, in Figure 18 we compare three versions of LP-CTA. The first (record_bounds) derives the rank bounds of considered cells based on per-record bounds, as presented in Section 6.1. The second (group_bounds) utilizes the data index and employs group bounds, as in Section 6.2. The third (fast_bounds) additionally utilizes the fast bounds in Section 6.3 for quick filtering, as in our full-fledged LP-CTA implementation. In Figures 18(a) and 18(b) we vary k and d, respectively, in our default setting. Our group bounds offer savings of 19% to 56% (compared to plain record bounds). On top of that, the fast bounds offer additional 16% to 64% savings (compared to the group bounds version).



(a) Effect of k (b) Effect of d

Figure 18: Effectiveness of group and fast bounds in LP-CTA

8. CONCLUSION

In this paper we introduce the *k-Shortlist Preference Region* problem (kSPR), which identifies all the regions in preference space where a given option (focal record) ranks among the top-k available alternatives. The problem finds application in market impact analysis and targeted advertising, among others. We develop a suite of techniques and data structures (e.g., $CellTree$, LP-based testing, look-ahead techniques) for efficiently solving this problem. We present a case study on the NBA dataset to demonstrate the usefulness of kSPR, and we verify the efficiency and practicality of our methodology with experiments on benchmark datasets. An interesting direction for future work is approximate kSPR algorithms, with accuracy guarantees, for the purpose of faster processing.

9. ACKNOWLEDGEMENTS

Man Lung Yiu and Bo Tang were supported by grant GRF 152196/16E from the Hong Kong RGC. Kyriakos Mouratidis was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

10. REFERENCES

[1] lp_solve. http://lpsolve.sourceforge.net/5.5/.

[2] qhull. http://www.qhull.org.

[3] P. K. Agarwal and M. Sharir. Arrangements and their applications. *Handbook of computational geometry*, pages 49–119, 2000.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] M. D. Berg, O. Cheong, M. V. Kreveld, and M. Overmars. *Computational geometry: algorithms and applications*. Springer, 2008.

[6] A. Blum, J. C. Jackson, T. Sandholm, and M. Zinkevich. Preference elicitation and query learning. *Journal of Machine Learning Research*, 5:649–667, 2004.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[8] Y. Cai, Y. Tang, and N. Mamoulis. Maximizing a record's standing in a relation. *IEEE Trans. Knowl. Data Eng.*, 27(9):2401–2414, 2015.

[9] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

[10] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993.

[11] M. A. Cheema, Z. Shen, X. Lin, and W. Zhang. A unified framework for efficiently processing ranking related queries. In *EDBT*, pages 427–438, 2014.

[12] W. Chu and Z. Ghahramani. Preference learning with gaussian processes. In *ICML*, pages 137–144, 2005.

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

[14] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Comput.*, 22(2):418–429, 1993.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[16] Y. Gao, Q. Liu, G. Chen, B. Zheng, and L. Zhou. Answering why-not questions on reverse top-k queries. *PVLDB*, 8(7):738–749, 2015.

[17] P. Godfrey. Skyline cardinality for relational processing. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 78–97. Springer, 2004.

[18] Z. He and E. Lo. Answering why-not questions on top-k queries. *IEEE Trans. Knowl. Data Eng.*, 26(6):1300–1315, 2014.

[19] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

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

[21] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[22] K. Mouratidis and H. Pang. Computing immutable regions for subspace top-k queries. In *PVLDB*, pages 73–84, 2013.

[23] K. Mouratidis, J. Zhang, and H. Pang. Maximum rank query. *PVLDB*, 8(12):1554–1565, 2015.

[24] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.

[25] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[26] P. Peng and R. C. Wong. k-hit query: Top-k query with probabilistic utility function. In *SIGMOD*, pages 577–592, 2015.

[27] A. M. Rashid, G. Karypis, and J. Riedl. Learning preferences of new users in recommender systems: an information theoretic approach. *SIGKDD Explorations*, 10(2):90–100, 2008.

[28] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *SIGMOD*, pages 805–816, 2011.

[29] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[30] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376, 2010.

[31] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1215–1229, 2011.

[32] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Identifying the most influential data objects with reverse top-k queries. *PVLDB*, 3(1):364–372, 2010.

[33] A. Vlachou, C. Doulkeridis, K. Norvag, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD*, pages 481–492, 2013.

[34] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-*k* queries. In *SIGMOD*, pages 397–408, 2012.

[35] J. Zhang, K. Mouratidis, and H. Pang. Global immutable region computation. In *SIGMOD*, pages 1151–1162, 2014.

[36] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *PVLDB*, 7(10):785–796, 2014.

APPENDIX

A. DISK-BASED SCENARIO

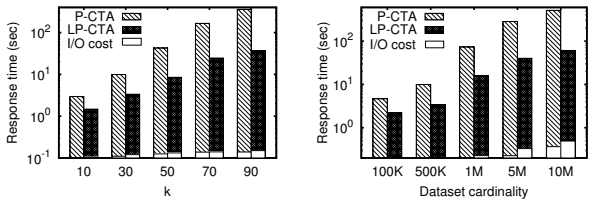
The experiments in Section 7 focus on the setting where data and index are kept in primary storage, because their combined size easily fits in the memory of commodity computers. Here we provide representative results for the scenario where data and index are stored on the disk, thus introducing I/O cost to the algorithms' overall response time. Our methods extend directly to that setting, because they utilize data indices (R-trees) that are readily applicable to disk-resident data.

We focus on our two best methods, P-CTA and LP-CTA. Figures 19(a), 19(b), and 19(c) present the total response time, accounting for both CPU and I/O time, using IND data and varying k, n, and d, respectively. Figure 19(d) shows the total response time for the real datasets using the default values for all parameters. The white part in each bar corresponds to the I/O time, and the patterned part to CPU time. The overall length of the bar (i.e., total response time) is in logarithmic scale, but its white and patterned parts are proportional to their contribution to the total response time. Note that in our system a random page read (on SSD) takes 0.2ms.

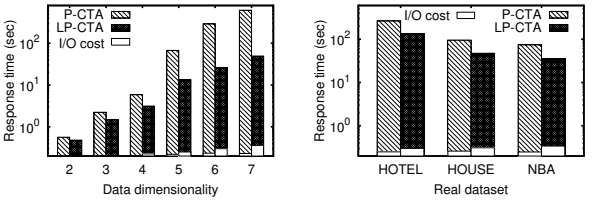
Although LP-CTA processes fewer records than P-CTA (as we demonstrated in Section 7), its look-ahead techniques rely on bounds derived by accessing the data index for every $CellTree$ leaf it considers. As a result, its I/O cost is larger than P-CTA. However, its superior CPU time (see Figures 10(b), 12(a), 13(a), and 15 for reference to the respective subfigures in Figure 19) renders LP-CTA several times faster in terms of total time, especially when scale is at its largest. For instance, for $k = 90$, its total response time is 9.8 times shorter than P-CTA. For $n = 10$M its total response time is 8.4 times shorter, and for $d = 7$ it is 12.4 times shorter. We conclude that LP-CTA is far superior to P-CTA in the disk-based scenario too.

B. P-CTA VS. K-SKYBAND APPROACH

At the end of Section 5, we mentioned that Lemma 6 implies that we could solve the kSPR problem if we run CTA on the k-skyband of D. At that point we stated that the resulting approach is impractical because the k-skyband includes numerous records, and that P-CTA processes a small subset of them. Here we substantiate



(a) Effect of k (IND) (b) Effect of n (IND)



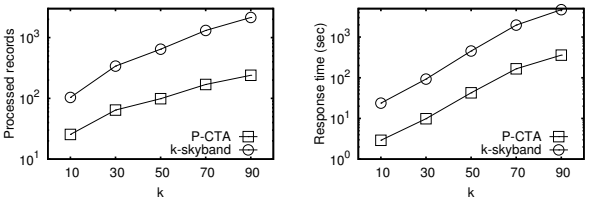
(c) Effect of d (IND) (d) Real datasets ($k = 30$)

Figure 19: Performance in disk-based scenario

that fact by means of an experiment.

In Figure 20 we compare the two approaches on IND data by varying k and keeping the remaining parameters to their defaults (note that the setup is the same as in Figures 10(b) and 11, so reference to those charts is possible too). Figure 20(a) presents the number of processed records in the two methods, and Figure 20(b) presents their running time. The size of the k-skyband is an order of magnitude larger than the number of records processed by P-CTA. The difference in the number of processed records renders P-CTA 4 to 9 times faster. These results illustrate the solid pruning ability of P-CTA. They also demonstrate that Lemma 6 is not the only reason behind P-CTA's efficiency, but optimizations like the pivot-based pruning and the direct cell reporting, which are enabled by Lemma 5, are particularly effective.

We stress that the purpose of this experiment is to offer insight into the strengths of P-CTA in particular, and to bring out its differences from the k-skyband approach, which is why our best approach, LP-CTA, is omitted from Figure 20. Its number of processed records and running time were already presented in Figures 11(a) and 10(b), respectively. For completeness, we mention that it is 16 to 131 times faster than the k-skyband approach.



(a) Processed records (b) Response time

Figure 20: P-CTA vs. k-skyband approach (IND)

C. TRANSFORMED VS. ORIGINAL SPACE

In Section 3.2 we explained that in our approach processing takes place in the transformed preference space, with dimensionality reduced by 1 compared to the original preference space. We mentioned that this lowers the cost of key operations, such as LP solving, because it depends on d. Here, we investigate this deeper, apply our methodology to the original space, and compare its versions for the two spaces.

In the original space, equality $S(\mathbf{r}) = S(\mathbf{p})$ for any $\mathbf{r} \in D$ corresponds to a hyperplane that passes through the origin. This means that the cells of arrangement Γ (and thus the kSPR regions too) are polyhedral cones. In two dimensions, the cells look like the wedges in Figure 1(b). In three dimensions, they look like the cone shown in Figure 21. In terms of algorithm design, the techniques in Sections 4 and 5 apply directly to the original space. The look-ahead techniques in Section 6, however, require redesign (while some of them do not apply at all, as we explain shortly).

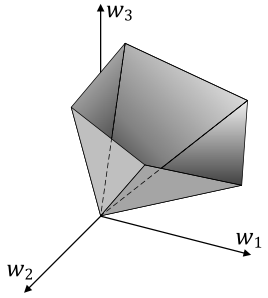


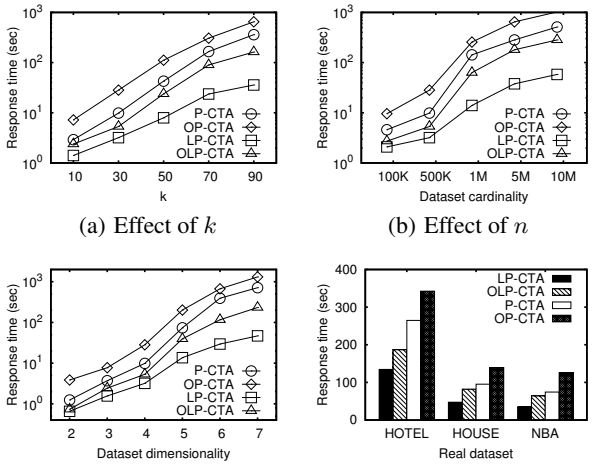
Figure 21: Cell in original preference space for $d = 3$

Take Section 6.1 as an example. Since every cell includes the origin, $\underline{S}(\mathbf{p}, c)$ is always 0. Similarly, for any record $\mathbf{r} \in D$, $\underline{S}(\mathbf{r}, c)$ is 0 too. Visually, this means that all the score intervals in Figure 7(b), including that of $\mathbf{p}$, start from 0. That fact yields[7] $\underline{Rank}(c) = 1$ and $\overline{Rank}(c) = 1+n$, i.e., renders the rank bounds useless. To circumvent this problem, our LP formulations for each $\mathbf{r} \in D$ should use $S(\mathbf{r}) - S(\mathbf{p})$ as the optimization function. Specifically, for every $\mathbf{r} \in D$, we solve an LP problem where we minimize optimization function $S(\mathbf{r}) - S(\mathbf{p})$ within the cell at hand. If the minimal value of the function is positive, it means that $S(\mathbf{r}) > S(\mathbf{p})$ *anywhere in the cell*, hence, we increment both $\underline{Rank}(c)$ and $\overline{Rank}(c)$ by 1. Else, we solve another LP to maximize the same function. If the maximal value is positive, it means that $S(\mathbf{r}) > S(\mathbf{p})$ *in a part of the cell*, thus, we only increment $\overline{Rank}(c)$. Following the same lines, we can adapt (to the original space) the group bounds in Section 6.2 too.

The aforementioned trick, however, does not apply to the fast bounds in Section 6.3. Specifically, the fact that every cell includes the origin means that the min-vector $\mathbf{w}^L$ will always be the origin, yielding $\underline{S}^{fast}(\mathbf{G}, c) = 0$ for every group $\mathbf{G}$ and every cell c. Therefore, the fast bounds cannot be used in the original space.

In Figure 22 we compare P-CTA and LP-CTA with their original-space counterparts, denoted as OP-CTA and OLP-CTA, respectively. We vary k, n, d using IND data, and also present results using the real datasets in the default setting. In all cases, OP-CTA and OLP-CTA are slower than their transformed-space versions. In particular, OP-CTA is 30% to 3.5 times slower than P-CTA. Similarly, OLP-CTA is 30% to 5 times slower than LP-CTA. The reason why the difference between the versions of LP-CTA is larger (than the difference between the versions of P-CTA), is because some look-ahead techniques in LP-CTA do not apply to the original space at all (namely, the fast bounds).

[7] From the definition of the rank bounds, we have $\underline{Rank}(c) = 1 + \texttt{COUNT}\{\mathbf{r} \in D : \underline{S}(\mathbf{r}, c) > \overline{S}(\mathbf{p}, c)\} = 1$, since $\underline{S}(\mathbf{r}, c) = 0$ for every $\mathbf{r}$. Also, $\overline{Rank}(c) = 1 + \texttt{COUNT}\{\mathbf{r} \in D : \overline{S}(\mathbf{r}, c) > \underline{S}(\mathbf{p}, c)\} = 1 + n$, since $\underline{S}(\mathbf{p}, c) = 0$.



(a) Effect of k (b) Effect of n

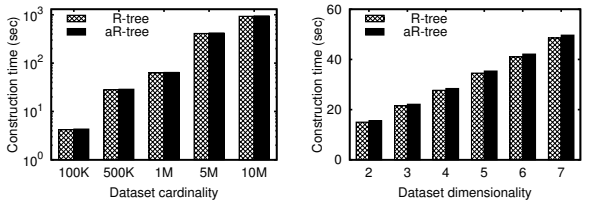
(c) Effect of d (d) Real datasets

Figure 22: Processing in transformed vs. original space

D. INDEX CONSTRUCTION COST

The (aggregate) R-tree index on D is a "build-once, use-many" general purpose structure, to be used for multiple kSPR queries, possibly also for different query types. For completeness, we present the one-off index construction cost, as well as the response time for P-CTA and LP-CTA when that cost is amortized over the 1000-query workloads we use in our experiments.

In Figure 23 we show the index construction cost when we vary n and d in the default IND dataset. Although in our implementation we use the same index for all methods (aggregate R-tree), we also present results for the plain R-tree (since CTA and P-CTA do not require an aggregate index). Note that the indices are built and kept in main memory. In Figure 24 we amortize the index construction cost over the 1000 queries executed in each experiment, and present the resulting response time of P-CTA and LP-CTA for different n and d, using IND data and the default $k = 30$. The performance of the algorithms does not differ much from the respective figures in Section 7, i.e., Figures 12(a) and 13(a). For instance, in the default setting, the amortization increases the response time of P-CTA by 0.29% and that of LP-CTA by 0.9%.



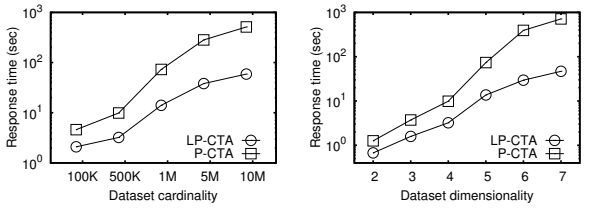
(a) Effect of n (b) Effect of d

Figure 23: Index construction time (IND)

E. PSEUDOCODES

Algorithm 1 summarizes CTA, as described in Section 4. The various halfspace sets involved are the following:

- $N.C$ denotes the cover set of a node N.
- Ψ_B is the set of halfspaces that label edges along the path from the root of $CellTree$ to node N.



(a) Effect of n (b) Effect of d

Figure 24: Amortized response time (IND)

- Ψ_S includes the halfspaces that define the boundaries of the preference space, i.e., it corresponds to constraints $\forall j \in [1, d'], w_j \in (0, 1); \sum_{j=1}^{d'} w_j \leq 1$.
- Ψ_C is the union of the cover sets of all the ancestors of N in $CellTree$.

Algorithm 1 CTA(dataset D, focal record **p**, value k)

```
1: Initialize CellTree root N with cover set N.C ← ∅
2: for each r_i ∈ D do
3:     Map r_i to hyperplane h_i
4:     N ← root of CellTree
5:     Insert(N, h_i)
6:     if the root N is eliminated then
7:         Return empty set
8: Return each CellTree leaf c with Rank(c) ≤ k

   Routine Insert(node N, hyperplane h_i):
9: if N is eliminated then
10:    Return
11: N_l, N_r ← left and right child of N
12: if Rank(N) > k or both N_r and N_l are eliminated then
13:    Eliminate N, eliminate its subtree (if any), and return
14: Ψ_B ← halfspaces that label edges along the path from root to N
15: Ψ_S ← space boundaries: ∀j ∈ [1, d'], w_j ∈ (0, 1); Σ_{j=1}^{d'} w_j ≤ 1
16: if h_i^- ∩ Ψ_B ∩ Ψ_S = ∅ then                          ▷ Case I
17:    N.C ← N.C ∪ h_i^+
18: else if h_i^+ ∩ Ψ_B ∩ Ψ_S = ∅ then                     ▷ Case II
19:    N.C ← N.C ∪ h_i^-
20: else                                                  ▷ Case III
21:    if N is a leaf then
22:        Split N into two children with empty cover sets
23:        Label edge to left / right child with h_i^- / h_i^+ respectively
24:    else
25:        Insert(N_l, h_i)
26:        Insert(N_r, h_i)

   Routine Rank(node N):
27: Ψ_C ← union of cover sets of all ancestors of N
28: Ψ_B ← halfspaces that label edges along the path from root to N
29: Return no. of positive halfspaces in N.C ∪ Ψ_C ∪ Ψ_B plus 1
```

By Lemma 2, sets $N.C$ and Ψ_C are inconsequential, i.e., sets Ψ_B and Ψ_S alone suffice to determine the extent of node N. Therefore, the conditions in lines 16 and 18 only consider Ψ_B, Ψ_S to define the extent of N (thus applying the optimization in Section 4.3.1 to accelerate feasibility tests).

The full set of halfspaces that define/cover node N is the union $N.C \cup \Psi_C \cup \Psi_B$. Hence, by Lemma 1 the rank of N is equal to the number of positive halfspaces in that union, plus 1 (see line 29).

A final note about Algorithm 1 concerns lines 12–13. Node N and its entire subtree are pruned when $Rank(N)$ exceeds k. N is also pruned when both its children (if any) have been eliminated. The later case propagates bottom-up the elimination of children to

their parents, so that no explicit check for un-pruned leaves is required in the subtree of N in order to prune it. For instance, if the condition in line 6 is true, it means that all leaves in $CellTree$ have been eliminated.

Algorithm 2 summarizes P-CTA. Set $\mathcal{PR}$ holds the already processed records, i.e., those whose corresponding hyperplanes have already been inserted into $CellTree$. Set $\mathcal{S}$ holds the next batch of records to process. As explained in Section 5, Lemma 5 guarantees that if the condition in line 16 is false, the rank and extent of c cannot be altered by any unprocessed record, and since it is a promising cell, i.e., $Rank(c) \leq k$, it can already be included in the kSPR result T and be ignored in subsequent processing (line 19). Promising cells that cannot be reported directly, have their non-pivot records collected into set $\mathcal{NP}$ (line 17). The records in $\mathcal{NP}$ are removed from the current skyline $\mathcal{SL}$ in line 20, and the remaining part of $\mathcal{SL}$ is updated by unprocessed records using the incremental BBS algorithm.

Algorithm 2 P-CTA(dataset D, focal record $\mathbf{p}$, value k)

1: Initialize $CellTree$ root N with cover set $N.C \leftarrow \emptyset$; result set $T \leftarrow \emptyset$
2: Initialize processed record set $\mathcal{PR} \leftarrow \emptyset$; skyline set $\mathcal{SL} \leftarrow \emptyset$
3: Incremental-BBS($\mathcal{SL}$, D)
4: $DG \leftarrow$ initialize dominance graph with a node for each record in $\mathcal{SL}$
5: Initialize $\mathcal{S} \leftarrow \mathcal{SL}$ ▷ First batch to process
6: **while** TRUE **do**
7: **for** each $\mathbf{r}_i \in \mathcal{S}$ **do**
8: Map $\mathbf{r}_i$ to hyperplane h_i
9: $D_r \leftarrow \mathbf{r}_i$'s ancestors in DG ▷ Records that dominate $\mathbf{r}_i$
10: $\mathbf{optInsert}(N, h_i, D_r)$
11: **if** the root N is eliminated **then** return T
12: $\mathcal{PR} \leftarrow \mathcal{PR} \cup \mathcal{S}$
13: Initialize union of non-pivots $\mathcal{NP} \leftarrow \emptyset$
14: **for** each leaf c in $CellTree$ with $Rank(c) \leq k$ **do**
15: $\mathcal{NP}_c \leftarrow$ non-pivot records of c
16: **if** $\exists \mathbf{r} \in D - \mathcal{PR}$ such that $\mathbf{r}$ is not dominated by any pivot of c and $\mathbf{r}$ is dominated by a record in $\mathcal{NP}_c$ **then**
17: $\mathcal{NP} \leftarrow \mathcal{NP} \cup \mathcal{NP}_c$
18: **else**
19: $T \leftarrow T \cup c$; remove c from $CellTree$
20: Update $\mathcal{SL}$ by Incremental-BBS($\mathcal{SL} - \mathcal{NP}$, $D - \mathcal{PR}$)
21: Set next batch to process $\mathcal{S} \leftarrow$ the unprocessed records in $\mathcal{SL}$
22: Update DG with $\mathcal{S}$ and with their dominance relationships
23: Return T

Routine optInsert(node N, hyperplane h_i, dominating records D_r):
24: **if** N is eliminated **then**
25: Return
26: $N_l, N_r \leftarrow$ left and right child of N
27: **if** $Rank(N) > k$ or both N_r and N_l are eliminated **then**
28: Eliminate N, eliminate its subtree (if any), and return
29: $\Psi \leftarrow$ the full halfspace set of N
30: **if** $\exists\, h_j^- \in \Psi$ where $\mathbf{r}_j \in D_r$ **then**
31: $N.C \leftarrow N.C \cup h_i^-$
32: **else**
33: Same as lines 14–24 in **Routine Insert** in Algorithm 1
34: $\mathbf{optInsert}(N_l, h_i, D_r)$
35: $\mathbf{optInsert}(N_r, h_i, D_r)$

Algorithm 3 summarizes LP-CTA. Its main part relies on Algorithm 2, with the addition of the rank bound computation and the subsequent quick pruning or reporting of cells in lines 8 and 10. The key procedure is **UpdateRank** that utilizes the aggregate R-tree on D to derive the rank bounds for the cell at hand. During the traversal of the R-tree, we first apply fast bounds to the encountered entries $\mathbf{G}$ and only if comparison to the score interval of $\mathbf{p}$ is inconclusive, do we compute the tighter group bounds $\underline{S}(\mathbf{G}^L, c), \overline{S}(\mathbf{G}^U, c)$ in line 23. If the comparison is again inconclusive, we execute procedure **UpdateRank** recursively on the child entries of node $\mathbf{G}$ in lines 25–26. Similarly, for every encountered record $\mathbf{r}$, if the fast bounds fail to determine the relative score order between $\mathbf{r}$ and $\mathbf{p}$, we compute the more accurate $\underline{S}(\mathbf{r}, c), \overline{S}(\mathbf{r}, c)$ in line 30. Recall that each $\underline{S}(\cdot, c)$ or $\overline{S}(\cdot, c)$ computation (be it for a group or individual record) requires the solution of an LP problem.

Algorithm 3 LP-CTA(dataset D, focal record $\mathbf{p}$, value k)

1: Same as lines 1–16 in Algorithm 2
2: $\underline{Rank}(c) \leftarrow 1$; $\overline{Rank}(c) \leftarrow 1$
3: $\mathbf{w}^L, \mathbf{w}^U \leftarrow$ min-vector and max-vector of cell c
4: $\underline{S}(\mathbf{p}, c), \overline{S}(\mathbf{p}, c) \leftarrow \mathbf{p}$'s score bound in cell c
5: **for** each entry $\mathbf{G}$ in the R-tree root with $\underline{Rank}(c) \leq k$ **do**
6: $\mathbf{UpdateRank}(\mathbf{G}, c)$
7: **if** $\underline{Rank}(c) > k$ **then**
8: Remove c from $CellTree$
9: **else if** $\overline{Rank}(c) \leq k$ **then**
10: $T \leftarrow T \cup c$; remove c from $CellTree$
11: **else**
12: Same as lines 17–23 in Algorithm 2

Routine UpdateRank(R-tree entry $\mathbf{G}$, cell c):
13: **if** $\mathbf{G}$ is an internal node **then**
14: $\underline{S}^{fast}(\mathbf{G}, c) \leftarrow \mathbf{w}^L \cdot \mathbf{G}^L$; $\overline{S}^{fast}(\mathbf{G}, c) \leftarrow \mathbf{w}^U \cdot \mathbf{G}^U$
15: **if** $\underline{S}^{fast}(\mathbf{G}, c) > \overline{S}(\mathbf{p}, c)$ **then**
16: $\underline{Rank}(c) \leftarrow \underline{Rank}(c) + \mathbf{G}.\mathsf{num}$
17: $\overline{Rank}(c) \leftarrow \overline{Rank}(c) + \mathbf{G}.\mathsf{num}$
18: **else if** $\underline{S}(\mathbf{p}, c) \leq \underline{S}^{fast}(\mathbf{G}, c)$ and $\overline{S}^{fast}(\mathbf{G}, c) \leq \overline{S}(\mathbf{p}, c)$ **then**
19: $\overline{Rank}(c) \leftarrow \overline{Rank}(c) + \mathbf{G}.\mathsf{num}$
20: **else if** $\overline{S}^{fast}(\mathbf{G}, c) < \underline{S}(\mathbf{p}, c)$ **then**
21: Return
22: **else**
23: $\underline{S}(\mathbf{G}^L, c), \overline{S}(\mathbf{G}^U, c) \leftarrow$ tight group bounds for $\mathbf{G}$ in cell c
24: Same as lines 15–22 by replacing the fast with the tight bounds
25: **for** each child entry $\mathbf{G}_i$ inside node $\mathbf{G}$ with $\underline{Rank}(c) \leq k$ **do**
26: $\mathbf{UpdateRank}(\mathbf{G}_i, c)$
27: **else** ▷ $\mathbf{G}$ is an R-tree leaf containing records
28: **for** each record $\mathbf{r}$ in $\mathbf{G}$ **do**
29: Same as lines 15–22 by using fast bounds for $\mathbf{r}$
30: $\underline{S}(\mathbf{r}, c), \overline{S}(\mathbf{r}, c) \leftarrow$ tight bounds for $\mathbf{r}$ in cell c
31: **if** $\underline{S}(\mathbf{r}, c) > \overline{S}(\mathbf{p}, c)$ **then**
32: $\underline{Rank}(c) \leftarrow \underline{Rank}(c) + 1$
33: $\overline{Rank}(c) \leftarrow \overline{Rank}(c) + 1$
34: **else if** $\overline{S}(\mathbf{r}, c) < \underline{S}(\mathbf{p}, c)$ **then**
35: Continue
36: **else**
37: $\overline{Rank}(c) \leftarrow \overline{Rank}(c) + 1$