τ-LevelIndex: **Towards Efficient Query Processing in Continuous Preference Space**

Jiahao Zhang[†], Bo Tang[‡§], Man Lung Yiu[†], Xiao Yan[‡§], Keming Li[‡§] *
[†]Department of Computing, Hong Kong Polytechnic University
[§] Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
[‡] Department of Computer Science and Engineering, Southern University of Science and Technology
{csjhzhang,csmlyiu}@comp.polyu.edu.hk,{tangb3@,yanx@,likm2020@mail.}sustech.edu.cn

ABSTRACT

Top-k related queries in continuous preference space (e.g., k-shortlist preference query kSPR, uncertain top-k query UTK, output-size specified utility-based query ORU) have numerous applications but are expensive to process. Existing algorithms process each query via specialized optimizations, which are difficult to generalize. In this work, we propose a novel and general index structure τ-LevelIndex, which can be used to process various queries in continuous preference space efficiently. We devise efficient approaches to build the τ-LevelIndex by fully exploiting the properties of continuous preference space. We conduct extensive experimental studies on both real- and synthetic- benchmarks. The results show that (i) our proposed index building approaches have low costs in terms of both space and time, and (ii) τ-LevelIndex significantly outperforms specialized solutions for processing a spectrum of queries in continuous preference space, and the speedup can be two to three orders of magnitude.

CCS CONCEPTS

• **Information systems → Top-k retrieval in databases**.

KEYWORDS

k-level index, preference space, top-k query processing

ACM Reference Format:
Jiahao Zhang, Bo Tang, Man Lung Yiu, Xiao Yan, Keming Li. 2022. τ-LevelIndex: Towards Efficient Query Processing in Continuous Preference Space. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526182

1 INTRODUCTION

The *preference model* is widely used in many applications, e.g., recommender system, multi-criteria decision making, and product ranking. Specifically, each option in the market has several attributes and the value of each attribute models the competitiveness of the option. For example, each hotel in the online portals (e.g, Booking [1] and TripAdvisor [6]) includes *value*, *facility*, and *cleanliness* attributes. The attribute values of *Crowne Plaza Hotel* in Booking are 0.83, 0.86, 0.89, respectively. We denote an option with d attributes as $\mathbf{r} = (r[1], r[2], \cdots, r[d])$ and all options form the dataset $\mathcal{D}$. A user chooses options based on her preference weight vector $\mathbf{w} = (w[1], w[2], \cdots, w[d])$, which specifies a numeric weight for each attribute. To compute the score of each option for the user, a linear scoring function (i.e., $\mathcal{S}_{\mathbf{w}}(\mathbf{r}) = \sum_{i=1}^{d} r[i]w[i]$) is used and the top-$k$ products with the highest scores are appealing to the user. In the past decades, the database community has conducted extensive studies on shortlisting product options for users, i.e., top-k query [22, 39], skyline query [11, 32], and hybrids of top-k and skyline queries [14, 28]. Rtree and its variants [10, 19] are widely used to accelerate the above query processing.

Besides finding top-k options for users, it is also important to consider user preferences from the perspective of product providers. For instance, a hotel manager may want to identify potential customers who rank his hotel as top-k, which should be targeted in advertising campaigns. Recently, many queries (e.g., *Monochromatic reverse top-k* [42], MaxRank [31], kSPR [37], *restricted skyline* [14], UTK [30], and ORU [28]) are proposed to explore the entire user preference space (i.e., the continuous space defined by preference weight $\mathbf{w}$) instead of discrete user preference weights in traditional top-k ranking queries (i.e., points in the preference space). These queries can help product providers analyze the competitiveness of their products in the market, identify target users, adjust the design of products to attract more users, etc. For example, the MaxRank query [31] reports the maximum rank a product can have among all possible users' preferences, which tells the product provider the market status of his product.

Many algorithms have been proposed to accelerate the query processing in continuous preference space, but they are still inadequate in two aspects. First, the query processing complexity is high even with state-of-the-art solutions as many expensive geometric operations are involved. For example, our experiments show that an ORU query can take more than 1000 seconds. Second, Existing algorithms develop specialized techniques for each query and there lacks a general index structure that accelerates a wide range of queries in continuous preference space, i.e., an analogue of Rtree for top-k ranking queries.

In this work, we devise a general index (τ-LevelIndex) for queries in continuous preference space, where τ is a user-specified parameter. A general index is favorable as it not only reduces query

[*]Dr. Bo Tang is the corresponding author.

processing complexity by avoiding expensive geometric operations but also amortizes the index construction and storage costs among queries (either of the same or different types).

Specifically, each option $\mathbf{r}$ in the dataset corresponds to a hyperplane $\mathcal{S}_{\mathbf{w}}(\mathbf{r})$ in continuous preference space, and the cell in τ-LevelIndex is defined by a set of halfspaces, which is formed by option hyperplanes. As we will elaborate in Section 4, many queries in continuous preference space can be processed by finding some cells in τ-LevelIndex or the options associated with certain cells. Constructing τ-LevelIndex can be converted to the famous *τ-level* problem [8] in computational geometry. However, the *τ-level* problem is only solved theoretically with time complexity $O(n^{\lfloor \frac{d}{2} \rfloor}\tau^{\lceil \frac{d}{2} \rceil})$ [9], where n is the number of options in the dataset $\mathcal{D}$, and d is the dimensionality of each option. The challenges of designing an effective index structure and efficient building procedure for τ-LevelIndex are:

- **Generality and query efficiency.** There are many types of queries in continuous preference space, e.g., kSPR, UTK, ORU. *How to design the structure of* τ-LevelIndex *such that it can support all these queries and process them efficiently?*
- **Index size.** As proved in [9], the total number of cells at level $\ell \in [1, \tau]$ of τ-LevelIndex is $O(n^{\lfloor \frac{d}{2} \rfloor}\ell^{\lceil \frac{d}{2} \rceil})$, and each cell is defined by the intersection of a set of halfspaces. Thus, τ-LevelIndex will be overwhelmingly large if cells are expressed explicitly. *How to define cells and arrange them in* τ-LevelIndex *to make the index size practical?*
- **Index building time.** Using [9] to build τ-LevelIndex has time complexity $O(n^{\lfloor \frac{d}{2} \rfloor}\tau^{\lceil \frac{d}{2} \rceil})$. Moreover, computing the cells in τ-LevelIndex requires expensive computational geometric operations (e.g., Intersect and Containment). *How to build* τ-LevelIndex *efficiently with a reasonable time cost?*

To overcome the above challenges, we first define the basic unit of τ-LevelIndex (i.e., *cell*) in a compact manner by implicitly exploiting its geometric properties, and connect two cells in adjacent levels of τ-LevelIndex if they have a parent-child relation. This structure not only reduces the index size but also enables τ-LevelIndex to process many queries (if not all) efficiently, as usually only a small number of cells need to be visited. We then propose three approaches (i.e., the UTK$_2$-adapted approach BSL, the insertion-based approach IBA and the partition-based approach PBA) to build τ-LevelIndex. BSL and IBA are simple but inefficient. However, they provide insights (i.e., avoid *redundant halfspaces in cells* and reduce *unnecessary cell insertion checking*) to devise novel and efficient PBA. We further utilize the dominance relations among options to improve the performance of PBA.

To the best of our knowledge, this is the first work in the database community that proposes a general index for a spectrum of queries in continuous preference space. Moreover, this work also proposes the first practical solution to the well-known *τ-level* problem (i.e., building τ-LevelIndex) in the computational geometry community. In addition to answering continuous preference space queries in database community, τ-LevelIndex may also find applications in computational geometry, e.g., simplex range searching [7], Voronoi diagram computing [23], graph drawing [15]. To sum up, the technical contributions of our work are as follows:

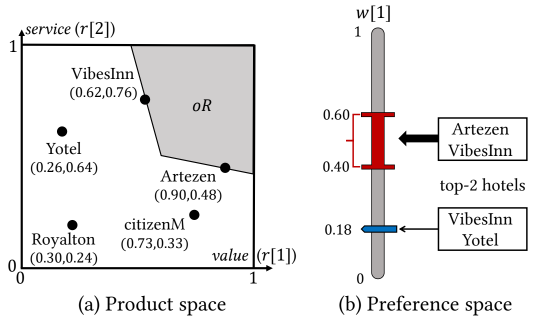


Figure 1: Discrete and continuous space

- We design a succinct structure for τ-LevelIndex with a novel implicit cell representation and formally define the τ-LevelIndex building problem in Section 3.
- For three representative queries (i.e., kSPR, UTK, and ORU) in continuous preference space, we show how to use τ-LevelIndex to process them efficiently in Section 4.
- We propose three building approaches for τ-LevelIndex with a suite of optimization techniques, which have low costs in both space and time, in Sections 5 and 6.
- We conduct extensive experiments on both synthetic- and real-datasets to validate the efficiency of our proposals for τ-LevelIndex building problem. Moreover, for three representative queries, we demonstrate the advantage of τ-LevelIndex in query processing efficiency over their state-of-the-art solutions in Section 7.

The remainder of this paper is organized as follows. Section 2 presents the preliminary and reviews the related work. Section 3 illustrates the τ-LevelIndex building problem. Section 4 shows how to process three representative queries via τ-LevelIndex. Sections 5 and 6 present our devised approaches for τ-LevelIndex building. Section 7 evaluates the effectiveness of our proposals by extensive experiments, and Section 8 concludes this work.

2 PRELIMINARY AND RELATED WORK

In this part, we introduce the preliminary of ranking queries in Section 2.1, then discuss the related work in Section 2.2.

2.1 Preliminary

Ranking queries have been extensively studied in the database community. Usually, it ranks products in the market by the users' preferences. Consider the option dataset $\mathcal{D} = \{\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_n\}$ in product space, it contains n options, which correspond to products such as hotels, laptops or phones in the market. Each product option $\mathbf{r} = (r[1], r[2], \cdots, r[d]) \in \mathcal{D}$ is a discrete point in the d-dimensional option space $\mathbb{R}^d$, see discrete black points in Figure 1(a), in which each dimension models an attribute of the product. The hotel dataset in Figure 1(a) has 5 hotels, each has two attributes *value* ($r[1]$) and *service* ($r[2]$). For example, the attributes *value* and *service* of *Artezen* are 0.90 and 0.48, respectively. The gray area in Figure 1(a) shows a continuous region *oR* in product space, we will elaborate it shortly. A user's preference for the attributes is captured

Table 1: Classification of ranking queries

Query type	Product space	Preference space	Relevant work
DC	Discrete	Continuous	[12, 14, 20, 29–31, 37, 40, 42, 49]
DD	Discrete	Discrete	[12, 18, 21, 26, 39–41, 43, 44]
CD	Continuous	Discrete	[36, 45–47]
CC	Continuous	Continuous	[17, 25, 38]

by weight vector $\mathbf{w} = (w[1], w[2], \cdots, w[d])$ in preference space. Without loss of generality [22], we assume the space of preference vector is $\{\mathbf{w} \in \mathbb{R}^d \mid \forall i \in [1,d]\ 0 \le w[i] \le 1, \text{and } \sum_{i=1}^{d} w[i] = 1\}$[1]. For example, Figure 1(b) illustrates the entire user preference space of $w[1]$ (i.e., the weight to the attribute *value*), which ranges from 0 to 1. Specifically, $w[1] = 0.18$ shows a user with weight vector $(w[1]=0.18, w[2]=1-w[1]=0.82)$, it is a discrete point in preference space.

The score of each product option $\mathbf{r} \in \mathcal{D}$ is calculated by the widely-used linear scoring function $\mathcal{S}_{\mathbf{w}}(\mathbf{r}) = \mathbf{r} \cdot \mathbf{w} = \sum_{i=1}^{d} r[i] \cdot w[i]$. For each user weight vector $\mathbf{w}$, the k highest scores of products are returned as the top-k result, e.g., the top-2 hotels of the user with $\mathbf{w} = (0.18, 0.82)$ are $\{$*VibesInn*, *Yotel*$\}$. The continuous preference region $[0.40, 0.60]$ shows a spectrum of users in preference space, i.e., these users who care *value* almost the same as *service*.

2.2 Related work

According to the properties of the product space and preference space (i.e., discrete versus continuous) in ranking queries, we categorize them into four types in Table 1, and briefly discuss each type of queries as follows.

Query type DC. In this work, we focus on the queries in this type. Specifically, it considers a set of discrete option points in product space, as 5 discrete hotels in Figure 1(a), and processes option-related queries in continuous user preference space, as the preference region $[0.40, 0.60]$ shown in Figure 1(b). The monochromatic reverse top-k query in [42] and why-not top-k query [20] are two classical problems in type DC. Skyline query [40] and onion-layer query [12] also belong to this type as they find qualified options in the entire preference space. Recently, many DC type queries (e.g., MaxRank [31], kSPR [37], and UTK [30]) are proposed, which we will discuss in more details in Section 4.

Query type DD. Queries in type DD work on a set of product option vectors (i.e., 5 discrete points in Figure 1(a)) and a set of user preference vectors (i.e., discrete point $w[1] = 0.18$ in preference space, see Figure 1(b)). Besides the traditional top-k query, reverse top-k query [41, 44] and top-m influential query [43] also belong to this type. Several techniques (e.g., indexing [39, 40], batching [18] and pre-computation [12, 21, 26]) have been proposed to accelerate these queries. Query type DD is inherently different from query type DC as the preference vectors are discrete. However, our τ-LevelIndex can also accelerate queries in type DD, as we will elaborate in Section 4.

Query type CD. Queries in this type are usually used for influence maximization applications, e.g., designing a new product such that it influences many users. Specifically, Tang et al. [36] proposed the m-impact region mIR problem, which identifies areas in product space, i.e., the continuous gray area *oR* in product space in Figure 1(a), such that the hotels lied in them attract at least m users in the market. The solution for mIR in [36] can be used to solve two other queries in type CD, i.e., influence-based cost optimization [46, 47] and improvement strategy making problem [45]. They are different from ours as we focus on discrete instead of continuous product space.

Query type CC. A representative of type CC is why-not reverse top-k query [17, 25], which explains why an option is not in the top-k result of a user. The query also suggests how to modify the option in continuous option space or the user preference vector in continuous preference space. E.g., how to modify *Yotel* or the preference region $[0.4, 0.6]$ in Figure 1(b) such that the option *Yotel* ranks top-2. Given a region in continuous preference space, the top-ranking region query (TopRR [38]) computes the region in continuous option space in which every option $\mathbf{r}$ belongs to the top-k set for every weight vector in the given preference region. For example, the gray region *oR* in Figure 1(a) includes all the top-k options for every weight $\mathbf{w}$ that lies in continuous preference region $[0.4, 0.6]$ in Figure 1(b). However, we work with discrete points instead of continuous areas in product space in this paper.

Other relevant queries. The k-hit query [33] models user preference vectors using a probability density function. Soliman et al. [35] find the most probable top-k result when user preference vectors are uniformly distributed in preference space. These works are different from type DC queries as they assume that the user preference follows a specific distribution while type DC queries assume that the user preference weight vector can locate in any position in continuous preference space.

3 LEVELINDEX BUILDING PROBLEM

Before we formulate our research problem, we first define a fundamental concept *halfspace* in continuous preference space.

Definition 1 (Halfspace). *Given two different options* $\mathbf{r}_i$ *and* $\mathbf{r}_j$ *in the option dataset* $\mathcal{D}$*, hyperplane* $\mathcal{H}_{i,j} : \mathcal{S}_{\mathbf{w}}(\mathbf{r}_i) = \mathcal{S}_{\mathbf{w}}(\mathbf{r}_j)$ *splits the whole continuous preference space into two halfspaces. The positive and negative halfspaces of* $\mathbf{r}_i$ *w.r.t.* $\mathbf{r}_j$ *are:* $\mathcal{H}^{+}_{i,j} = \{\mathbf{w} \in \mathbb{R}^{d-1} \mid \mathcal{S}_{\mathbf{w}}(\mathbf{r}_i) \ge \mathcal{S}_{\mathbf{w}}(\mathbf{r}_j)\}$ *and* $\mathcal{H}^{-}_{i,j} = \{\mathbf{w} \in \mathbb{R}^{d-1} \mid \mathcal{S}_{\mathbf{w}}(\mathbf{r}_i) \le \mathcal{S}_{\mathbf{w}}(\mathbf{r}_j)\}$*, respectively.*

Example. Consider *VibesInn*$(\mathbf{r}_1)$ and *Artezen*$(\mathbf{r}_2)$ in the hotel dataset in Figure 2(a). Hyerplane $\mathcal{H}_{1,2} : \mathcal{S}_{\mathbf{w}}(\mathbf{r}_1) = \mathcal{S}_{\mathbf{w}}(\mathbf{r}_2)$ splits the preference space [0,1] into two halfspaces, i.e., $\mathcal{H}^{+}_{1,2}$ and $\mathcal{H}^{-}_{1,2}$, as illustrated in Figure 2(b)[2]. The users in halfspace $\mathcal{H}^{+}_{1,2}$ prefer *VibesInn*$(\mathbf{r}_1)$ to *Artezen*$(\mathbf{r}_2)$ as the score of *VibesInn* $\mathcal{S}_{\mathbf{w}}(\mathbf{r}_1)$ is larger than or equal to the score of *Artezen* $\mathcal{S}_{\mathbf{w}}(\mathbf{r}_2)$. $\mathcal{H}^{+}_{1,2}$ is the same as $\mathcal{H}^{-}_{2,1}$ due to symmetry.

We then define rank-ℓ-th cell in preference space, which is the basic element in τ-LevelIndex.

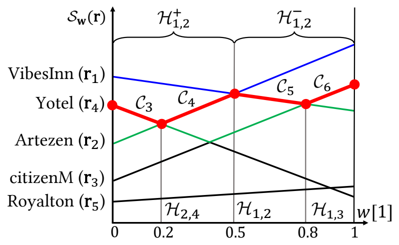
Definition 2 (Rank-ℓ-th Cell C). *Given an option dataset* $\mathcal{D}$*, we say a region* C *in preference space is a rank-*ℓ*-th cell iff every preference weight vector* $\mathbf{w}$ *in* C *satisfies (i) the top-*ℓ *option set is* $\mathcal{R}$*,*

[1]Note that as $\sum_{i=1}^{d} w[i] = 1$, we can determine a weight $\mathbf{w}$ using only $d-1$ parameters.

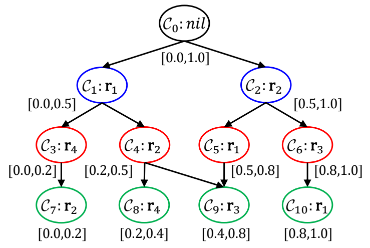
[2]For ease of presentation, we allow both halfspaces include the hyperplane $\mathcal{H}_{1,2}$.

Option	Hotel	Value	Service
$\mathbf{r}_1$	VibesInn	0.62	0.76
$\mathbf{r}_2$	Artezen	0.90	0.48
$\mathbf{r}_3$	citizenM	0.73	0.33
$\mathbf{r}_4$	Yotel	0.26	0.64
$\mathbf{r}_5$	Royalton	0.30	0.24

(a) Hotel dataset $\mathcal{D}$



(b) Score functions arrangement



(c) τ=3-LevelIndex

Figure 2: LevelIndex **example**

and (ii) the top-ℓ-th option is $\mathbf{r}_i$. With $\mathcal{R}$ and $\mathbf{r}_i$, the geometric region of the rank-ℓ-th cell C, denoted as $\mathrm{G}(C)$, can be expressed as

$$\mathrm{G}(C) = \Big(\bigcap_{\mathbf{r}_j \in \mathcal{R}-\{\mathbf{r}_i\}} \mathcal{H}^{-}_{i,j}\Big) \cap \Big(\bigcap_{\mathbf{r}_j \in \mathcal{D}-\mathcal{R}} \mathcal{H}^{+}_{i,j}\Big). \quad (1)$$

Example. Consider the rank-2nd cell C_4 in Figure 2(b). For every user whose preference weight vector $\mathbf{w}$ lies in C_4, it holds that (i) her top-2 hotels are $\mathcal{R} = \{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)\}$, and (ii) the top-2nd hotel is *Artezen*$(\mathbf{r}_2)$. The geometric region of C_4 (i.e., $w[1] \in [0.2, 0.5]$) in preference space is $\mathrm{G}(C_4) = \mathcal{H}^{-}_{2,1} \cap (\mathcal{H}^{+}_{2,3} \cap \mathcal{H}^{+}_{2,4} \cap \mathcal{H}^{+}_{2,5})$.

We next define level-ℓ arrangement in preference space.

Definition 3 (Level-ℓ Arrangement). *All rank-ℓ-th cells form the level-ℓ arrangement. It holds that the union of all rank-ℓ-th cells is the entire preference space.*

Example. Consider the example in Figure 2(b), level-2 arrangement consists of all rank-2nd cells, i.e., 4 red cells C_3, C_4, C_5 and C_6. The union of all rank-2nd cells' region is the entire preference space, i.e., $[0, 0.2] \cup [0.2, 0.5] \cup [0.5, 0.8] \cup [0.8, 1] = [0, 1]$. Interestingly, the union set of the rank-2nd options of these cells, i.e., $\{$*Yotel*$(\mathbf{r}_4)$, *Artezen*$(\mathbf{r}_2)$, *VibesInn*$(\mathbf{r}_1)$, *citizenM*$(\mathbf{r}_3)\}$, contains all options that can rank top-2nd for any user preference vector. It means that hotels not in the union set cannot rank the 2nd for any customer, e.g., *Royalton*$(\mathbf{r}_5)$ in Figure 2.

We formally specify the structure of τ-LevelIndex in Definition 4.

Definition 4. *(τ-LevelIndex) Given an integer $\tau \le |\mathcal{D}|$, the τ-*LevelIndex *is a directed acyclic graph (DAG), where the vertex set contains all rank-ℓ-th cells for $\ell \in [1, \tau]$. A rank-ℓ-th cell C_i stores its top-ℓ-th option $\mathbf{r}_\ell$, and it has a directed edge to a rank-$(\ell+1)$-th cell C_j iff the geometric region of C_i intersects with the geometric region of C_j, i.e., $\mathrm{G}(C_i) \cap \mathrm{G}(C_j) \neq \emptyset$.*

Example. Consider the 2-dimensional product dataset $\mathcal{D}$ in Figure 2(a). Its τ=3-LevelIndex is shown in Figure 2(c). We add a rank-0 cell C_0 that corresponds to the entire preference space (called the entry cell). For each cell C_i in τ-LevelIndex, the path length from C_0 to C_i is its level ℓ. As shown in Figure 2(c), the top-ℓ-th option is stored in each rank-ℓ-th cell C_i. We outline level-1, 2, 3 cells in blue, red, green, respectively. For any rank-ℓ-th cell C_i, its top-ℓ option set $\mathcal{R}$ can be obtained by traversing any path from the entry cell to C_i. Take rank-3rd cell C_9 for example, its top-3 hotel set is $\mathcal{R} = \{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)$, *citizenM*$(\mathbf{r}_3)\}$ (i.e., traversing the path $C_0 \to C_1 \to C_4 \to C_9$ or $C_0 \to C_2 \to C_5 \to C_9$). With the top-$\ell$ option set $\mathcal{R}$ of each cell C_i, we can obtain its geometric region using Definition 2. For the τ-LevelIndex example in Figure 2(c), we explicitly list the preference region of each cell, e.g., the preference region of C_4 is $[0.2, 0.5]$. All cells with the same rank ℓ cover the entire preference space and form the level-ℓ arrangement. The τ-LevelIndex can be utilized to answer many ranking queries in both type DD and DC (see Table 1) as we will show in Section 4. Thus, we define the τ-LevelIndex building problem in Problem 1.

Problem 1 (τ-LevelIndex building). *Given a product dataset $\mathcal{D}$ and an integer τ, the τ-*LevelIndex *building problem is to build the τ-*LevelIndex *efficiently in terms of both space and time consumption.*

4 QUERY PROCESSING WITH τ-LevelIndex

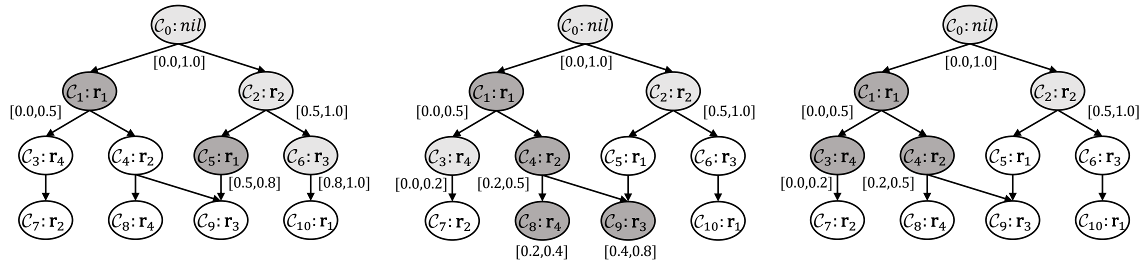
Before we present our solutions for the τ-LevelIndex building problem, we illustrate its usability in this section. Specifically, we first classify the type DC queries in Table 1 into three categories and show how to use τ-LevelIndex to process the representative query in each category. Then, we briefly discuss the generality and limitations of τ-LevelIndex.

I. Option analytical query in DC**.** The monochromatic reverse top-k query [42] returns the line segments in [0,1], in which the given option $\mathbf{r}$ (e.g., *VibesInn*$(\mathbf{r}_1)$ in Figure 2(a)) can be the top-k result. The maximum rank MaxRank query [31] returns the highest rank of an option $\mathbf{r}$ in the entire preference space. For example, the highest rank of *Yotel*$(\mathbf{r}_4)$ in the market. Tang et al. [37] solved a generalization of monochromatic reverse top-k query, i.e., k-shortlist preference region problem kSPR. We use kSPR query (see Problem 2) as an example to show how to process option analytical queries with τ-LevelIndex.

Problem 2 (kSPR query [37]). *Given a dataset $\mathcal{D}$, a focal option $\mathbf{r}$ and an integer k,* kSPR *query returns all regions in preference space in which $\mathbf{r}$ ranks top-k among $\mathcal{D}$.*

Using τ-LevelIndex, kSPR$(k, \mathbf{r})$ can be solved by traversing all paths from the entry cell C_0 until reaching the k-th level or the queried option $\mathbf{r}$, whichever sooner. Take kSPR$(2,$ *VibesInn*$(\mathbf{r}_1))$ for example, it finds the preference regions where *VibesInn*$(\mathbf{r}_1)$ is always one of the top-2 hotels. As shown in Figure 3(a), it returns C_1 and C_5 as the kSPR result regions. The search process only visits 5 cells (in gray) on τ-LevelIndex, i.e., C_0, C_1, C_2, C_5, and C_6. Compared to the specialized solution in [37], processing kSPR using τ-LevelIndex avoids (i) building the *CellTree* from scratch, which has extremely high time complexity[3], and (ii) incurring expensive

[3]The time complexity is $O(\alpha \cdot (k\frac{\log^{d-1} n}{d!})^d)$, where α is a constant that depends on the dimensionality d [37].


$\mathcal{C}_0$: nil
[0.0,1.0]
$\mathcal{C}_1$: $\mathbf{r}_1$
$\mathcal{C}_2$: $\mathbf{r}_2$
[0.0,0.5]
[0.5,1.0]
$\mathcal{C}_3$: $\mathbf{r}_4$
$\mathcal{C}_4$: $\mathbf{r}_2$
$\mathcal{C}_5$: $\mathbf{r}_1$
$\mathcal{C}_6$: $\mathbf{r}_3$
[0.5,0.8]
[0.8,1.0]
[0.0,0.2]
[0.2,0.5]
$\mathcal{C}_7$: $\mathbf{r}_2$
$\mathcal{C}_8$: $\mathbf{r}_4$
$\mathcal{C}_9$: $\mathbf{r}_3$
$\mathcal{C}_{10}$: $\mathbf{r}_1$
[0.2,0.4]
[0.4,0.8]

(a) kSPR($k=2, \mathbf{r}=\mathbf{r}_1$) (b) UTK($k=3, c=[0.35, 0.45]$) (c) ORU($k=2$, $w[1]=0.3, m=3$)

Figure 3: Representative queries on LevelIndex

function calls for feasibility test (in lp_solve library), which is a core subroutine in the optimized solution (i.e., LP-CTA) for kSPR [37].

II. Preference region query in DC. Ciaccia and Martinenghi [14] proposed the *restricted skyline* query. Specifically, given a family of scoring functions (e.g., L_p norms) and a set of linear conditions on the weights (i.e., a continuous region in preference space), the *restricted skyline* query returns all options in the dataset which could rank top-1st for every weight that satisfies the linear conditions. The uncertain top-k UTK query [30] generalizes the *restricted skyline* query, i.e., k can be an arbitrary value. We use UTK query (including its two variants) as an example to show how τ-LevelIndex processes preference region queries.

Problem 3 (UTK query [30]). *Given a dataset $\mathcal{D}$, a query region c in preference space and an integer k,* UTK *query reports (i) all options that can rank top-k among $\mathcal{D}$ for any possible preference weight $\mathbf{w}$ in c, and (ii) a partitioning of c where each partition is associated with its top-k result set.*

Using the τ-LevelIndex, UTK(k, c) can be solved by identifying all level-k cells which intersect with c, and then collecting the top-k result set of these cells. Suppose data analysts want to find the top-3 hotels when the users' weight on *value* ranges from 0.35 to 0.45. It can be answered by UTK query, i.e., UTK$(3, c=[0.35, 0.45])$ in Figure 3(b), the search process checks C_1 and C_2 on level-1, and skips C_2 as it does not intersect with c. Similarly, c intersects with C_4 on level-2, and terminates after checking C_8 and C_9 on level-3. All the gray cells in Figure 3(b) are visited when processing UTK. The top option set of a cell C can be obtained by collecting the top-ℓ-th products of the visited qualified cells in τ-LevelIndex. For example, the result of UTK$(3, c=[0.35, 0.45])$ is $\{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)$, *citizenM*$(\mathbf{r}_3)$, *Yotel*$(\mathbf{r}_4)\}$, as the hotels shown in the deep gray cells of Figure 3(b). The partitioning of C is $\mathrm{G}(C_8)\cap c$ and $\mathrm{G}(C_9)\cap c$, which top-3 result sets are $\{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)$, *Yotel*$(\mathbf{r}_4)\}$ and $\{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)$, *citizenM*$(\mathbf{r}_3)\}$, respectively.

III. Option and preference region hybrid query in DC. Given a user preference vector $\mathbf{w}$, the immutable queries [29, 49] return a preference region around $\mathbf{w}$ where the top-k result is the same. Specifically, [29] returns the area which only varies in one dimension, but [49] returns the area which varies in all d dimensions. The ORU [28] solved a generalized version of both local [29] and global [49] immutable region queries. Thus, we use ORU query to show how τ-LevelIndex can be applied to process option and preference region hybrid queries.

Table 2: ORU **illustration for Figure 3(c)**

Iteration	Min-heap	Result set $\mathcal{R}$
1st	$\langle(\underline{C_0}, 0)\rangle$	$\{\emptyset\}$
2nd	$\langle(\underline{C_1}, 0), (C_2, 0.2)\rangle$	$\{$*VibesInn*$(\mathbf{r}_1)\}$
3rd	$\langle(\underline{C_4}, 0), (C_3, 0.1), (C_2, 0.2)\rangle$	$\{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)\}$
4th	$\langle(\underline{C_3}, 0.1), (C_2, 0.2)\rangle$	$\{$*VibesInn*$(\mathbf{r}_1)$, *Artezen*$(\mathbf{r}_2)$, *Yotel*$(\mathbf{r}_4)\}$

Problem 4 (ORU query). *Given a dataset $\mathcal{D}$, an integer k, a query weight $\mathbf{w}$ and the requested result size m,* ORU *reports a sized-m result set, in which each option belongs to the top-k result for at least one reference vector $\mathbf{w}'$ within the minimum expansion distance ρ from $\mathbf{w}$, i.e., $\|\mathbf{w}' - \mathbf{w}\| \leq \rho$.*

Using τ-LevelIndex, ORU$(k, \mathbf{w}, m)$ can be solved by collecting the top option sets of level-ℓ cells in ascending order of their distances[4] to $\mathbf{w}$ until the number of products reaches m. In particular, we first push the root cell C_0 into a min-heap such that the head of the heap has the minimum distance to $\mathbf{w}$. In each iteration, we merge the top-ℓ-th option of the cell at heap head into the result set if $\ell \leq k$, then pop the head and push all its connecting cells in level-$(\ell+1)$ iff $(\ell+1) \leq k$. The procedure terminates when the size of result set reaches m. Suppose data analysts want to find 3 hotels, each of which can be the top-2 result for at least one user around w[1]=0.3, as ORU$(k = 2, w[1] = 0.3, m = 3)$ shown in Figure 3(c). Table 2 shows the heap and result set during the search process, where the value associated with each cell is its minimum distance to $\mathbf{w}$. Compared to the solution in [28], processing ORU query with τ-LevelIndex significantly reduces time complexity as it avoids expensive geometric computations.

Discussion of τ-LevelIndex**.** For the above examples, we can see that τ-LevelIndex is a general index for many queries in continuous preference space. This is because these queries usually are finding a specified preference cell or a set of options which are associated with specified preference cells in τ-LevelIndex, and τ-LevelIndex readily stores all cells and top option sets. Thus, answers to the queries can be obtained by simply looking up the required cells and top option sets in τ-LevelIndex. For example, the why-not top-k query [20] and skyline query [40] also can be processed efficiently with τ-LevelIndex. In particular, the option set in each level arrangement in τ-LevelIndex is tighter than the result set of the corresponding skyline or onion-layer queries. For the why-not top-k query with input option $\mathbf{r}$ and user weight $\mathbf{w}$, we can find the

[4]In general cases, a cell C is a convex set in $\mathbb{R}^{d-1}$ and a weight is a point in $\mathbb{R}^{d-1}$. Thus, the distance between a weight $\mathbf{w}$ and a cell C is defined as $\|\mathbf{w} - \mathbf{w}'\|$, where $\mathbf{w}'$ is the projection of $\mathbf{w}$ in C.

cells in τ-LevelIndex where $\mathbf{r}$ is in the top-k result at first, and then explain why $\mathbf{r}$ is not in the top-k result of user $\mathbf{w}$ by checking these cells. In addition to the queries in type DC, other queries also can be processed on τ-LevelIndex. For example, the top-k query can be directly answered by checking the cells which contain the user weight vector $\mathbf{w}$ from level 1 to k in τ-LevelIndex, we will investigate its performance in Section 7.3. Besides processing preference space queries, τ-LevelIndex has many applications in computational geometry problems, e.g., solving τ-*level* [34] and τ-*set* [13] problems, and the vertices of cells can be used to assist graph drawing [15, 27].

As we stated in Section 2, τ-LevelIndex cannot be used to answer the queries in types CC and CD. The reason is the queries (e.g., mIR query in [36] and why-not reverse top-k query in [17, 25]) in both types require the continuous product space either as input or as output, but τ-level index is a partition of continuous preference space by discrete options in product space, see Figures 2(b) and (c).

5 OUR BASIC APPROACHES

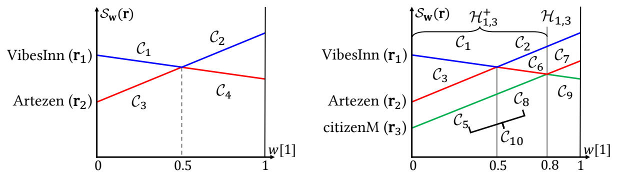
Even though τ-LevelIndex can be used to answer a spectrum of queries in continuous preference space, it is only solved theoretically with expensive time complexity [9]. In this section, we devise two basic approaches to build τ-LevelIndex (see Problem 1) in practical. In particular, we propose the first basic approach (i.e., UTK$_2$-adapted approach BSL) for τ-LevelIndex building problem in Section 5.1, then devise the second basic approach (i.e., the insertion-based approach IBA) in Section 5.2.

5.1 UTK$_2$-adapted Approach: BSL

For a given region c in preference space, Mouratidis and Tang [30] devised an algorithm (called UTK$_2$) that partitions c into (smaller) sub-regions in which preference vectors have the same top-ℓ option set. UTK$_2$ can be adapted to build τ-LevelIndex using the following procedures: (i) for each $\ell \in [1, \tau]$, use the entire preference space as the query region c and incur UTK$_2$ to obtain its partitioning, which returns all cells in level-ℓ arrangement; (ii) for each cell in level ℓ, conduct intersection checking for every cell in level-$(\ell+1)$ to add the directed edges. The UTK$_2$-adapted approach BSL is correct but it incurs high complexity, we will verify it in Section 7. The reasons are (i) the cost of UTK$_2$ is high (see Lemma 5 in [30]), and (ii) numerous intersection testings are required to connect cells. Even though the UTK$_2$-adapted approach BSL is expected to be inefficient, to the best of our knowledge, it is the first implemented solution for the τ-LevelIndex building problem in both database and computational geometric communities.

5.2 Insertion-based Approach: IBA

The insertion-based approach IBA builds τ-LevelIndex by inserting options one by one. To insert an option $\mathbf{r}_j$, we check cells in the current τ-LevelIndex in a top-down manner. For a rank-ℓ-th cell C, we only store its top-ℓ-th option, say $\mathbf{r}_\ell$. Its top-ℓ result set $\mathcal{R}$ could be derived by visiting the nodes on the path from C_0 to C. When checking cell C for the inserted option $\mathbf{r}_j$, IBA tests whether $\mathbf{r}_j$ ranks higher than or equal to $\mathbf{r}_\ell$ for any preference vector $\mathbf{w}$ in C. This is equivalent to determining the relation between the geometric region of cell C (i.e., $\mathrm{G}(C)$) and halfspaces of $\mathcal{H}_{\ell,j}$ (i.e., $\mathcal{H}^+_{\ell,j}$ and $\mathcal{H}^-_{\ell,j}$), which boils down to the following three cases.



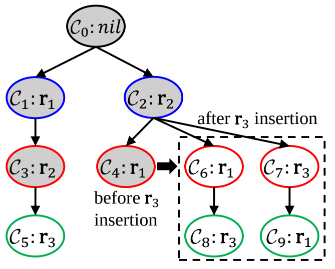
(a) Before $\mathbf{r}_3$ insertion (b) After $\mathbf{r}_3$ insertion

Figure 4: Option insertion for 3-LevelIndex **example**

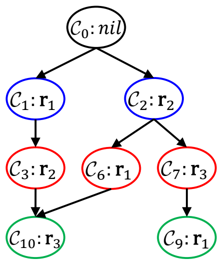
- Case I **If** $(\mathcal{H}^+_{\ell,j} \supseteq \mathrm{G}(C))$: it means that $\mathbf{r}_\ell$ always ranks higher than $\mathbf{r}_j$ in cell C. For example, as shown in Figure 4(b), $\mathcal{H}^+_{1,3} = \{\mathbf{w} | \mathcal{S}_\mathbf{w}(\mathbf{r}_1) \geq \mathcal{S}_\mathbf{w}(\mathbf{r}_3)\} \supseteq \mathrm{G}(C_1)$, it means *VibesInn*($\mathbf{r}_1$) always has higher scores than *citizenM*($\mathbf{r}_3$) for every user in preference region C_1. Thus, it checks $\mathbf{r}_j$ with the children of cell C in τ-LevelIndex recursively if C is an internal node. Otherwise, i.e, C is a leaf node, a new rank-$(\ell+1)$-th cell C' (with top-$(\ell+1)$-th option being $\mathbf{r}_j$) is created and set as a child of cell C when $\ell + 1 \leq \tau$.
- Case II **Else if** $(\mathcal{H}^-_{\ell,j} \supseteq \mathrm{G}(C))$: it means that for every weight vector $\mathbf{w} \in C$, the score of $\mathbf{r}_j$ is always larger than $\mathbf{r}_\ell$. Thus, the rank-ℓ-th option in cell C should be updated to $\mathbf{r}_j$. It inserts a new node C' (with the rank-ℓ-th option being $\mathbf{r}_j$) and sets cell C as a child of the newly inserted cell C' (which makes C as a rank-$(\ell+1)$-th cell). It is symmetric to Case I.
- Case III **Else:** it means that the hyperplane $\mathcal{H}_{\ell,j}$ splits cell C into two parts. For example, $\mathcal{H}_{1,3}$ (i.e., $w[1] = 0.8$) splits cell C_4 in Figure 4(b). For the part where the rank of $\mathbf{r}_\ell$ is higher than $\mathbf{r}_j$, it checks $\mathbf{r}_j$ with the children of cell C recursively as Case I. For the other part, where $\mathbf{r}_\ell$ ranks lower than $\mathbf{r}_j$, it creates a new rank-ℓ-th node C' (with the rank-ℓ-th option being $\mathbf{r}_j$), and sets cell C as a child of the newly inserted cell C' as Case II.

Take Figure 4 as an example, Figure 4(a) shows the two level arrangements with two hotels *VibesInn*($\mathbf{r}_1$) and *Artezen*($\mathbf{r}_2$). After inserting *citizenM*($\mathbf{r}_3$), the three level arrangements are shown in Figure 4(b). We will elaborate the details of inserting *citizenM*($\mathbf{r}_3$) to build a 3-LevelIndex in Figure 5, which corresponds to the level arrangements in Figure 4(b). The cells in 3-LevelIndex before inserting *citizenM*($\mathbf{r}_3$) are shown in Figure 5(a) in gray color, i.e., C_0 to C_4. Consider the insertion of *citizenM*($\mathbf{r}_3$), it first checks *citizenM*($\mathbf{r}_3$) with cell C_1 (with top-1st option *VibesInn*($\mathbf{r}_1$)). Since the score of *VibesInn*($\mathbf{r}_1$) is always larger than *citizenM*($\mathbf{r}_3$) in C_1, see $\mathcal{S}_\mathbf{w}(\mathbf{r}_1)$ and $\mathcal{S}_\mathbf{w}(\mathbf{r}_3)$ in Figure 4(b), Case I holds. It then checks *citizenM*($\mathbf{r}_3$) with the child cell of C_1, i.e., C_3 with top-2nd option *Artezen*($\mathbf{r}_2$), *citizenM*($\mathbf{r}_3$) ranks lower than *Artezen*($\mathbf{r}_2$) for every preference weight vector $\mathbf{w}$ in cell C_3, thus, a new node C_5 with top-3rd option *citizenM*($\mathbf{r}_3$) is created and set as a child of cell C_3 in 3-LevelIndex.

Now we consider inserting *citizenM*($\mathbf{r}_3$) into cell C_2. As Case I holds for cell C_2, its checks *citizenM*($\mathbf{r}_3$) with C_2's child cell C_4. For cell C_4 with top-2nd option *VibesInn*($\mathbf{r}_1$), hyperplane $\mathcal{H}_{1,3}$ splits it into two parts, i.e., C_6 and C_7. It shows the insertion procedure of Case III. For the part in C_4 where the rank of *VibesInn*($\mathbf{r}_1$) is higher than *citizenM*($\mathbf{r}_3$), i.e., C_6 with top-2nd option *VibesInn*($\mathbf{r}_1$), a new node C_8 with top-3rd option *citizenM*($\mathbf{r}_3$) is added as the child of cell C_6. For the other part in C_4 where the rank of *VibesInn*($\mathbf{r}_1$) is lower than *citizenM*($\mathbf{r}_3$), i.e., C_7 with top-2nd option *citizenM*($\mathbf{r}_3$),



(a) Inserting *citizenM*($\mathbf{r}_3$)



(b) 3-LevelIndex

Figure 5: Insertion-based approach example

it creates a new cell C_9 with top-3rd option *VibesInn*($\mathbf{r}_1$) and sets C_9 as the child of cell C_7.

The pseudocode in Algorithm 1 summarizes the insertion-based approach IBA. At the beginning, it initializes the entry cell C_0 of τ-LevelIndex by setting its rank-ℓ-th option $\mathbf{r}_\ell$ as *nil*. It inserts each option $\mathbf{r}_j \in \mathcal{D}$ into τ-LevelIndex one by one, as shown in Line 2 to 5 in Aglorithm 1. The key subroutine in the insertion-based approach IBA is Insert, its inputs are cell C, inserting option $\mathbf{r}_j$ and the top-ℓ result set $\mathcal{R}$ of cell C. Given a cell C, with the top-ℓ result set $\mathcal{R}$ and its rank-ℓ-th option $\mathbf{r}_\ell$, the exact preference region of cell C, i.e., G(C), can be obtained by Definition 2. Then, we employ lp_Solve[5] to check the relationship between cell C and hyperplane $\mathcal{H}_{\ell,j}$, which is formed by C's rank-ℓ-th option $\mathbf{r}_\ell$ and inserting option $\mathbf{r}_j$. Line 8 to 23 corresponds to the three cases we have elaborated.

After inserting each option $\mathbf{r}_j$, it incurs a Merge function to improve the efficiency of subsequent insertions. Specifically, the Merge function merges rank-ℓ-th cells that share the same top-ℓ result set and top-ℓ-th option. For example, consider rank-3-th cells C_5 and C_8 in Figure 5(a), the top-3 result sets and top-3rd option of both cells are $\mathcal{R} = \{$*VibesInn*($\mathbf{r}_1$), *Artezen*($\mathbf{r}_2$), *citizenM*($\mathbf{r}_3$)$\}$ and *citizenM*($\mathbf{r}_3$), respectively. Thus, C_5 and C_8 are merged into a cell C_{10}, as shown in Figure 5(b). During the insertion process, cells with higher rank than τ will be eliminated, as it will not be a part of the result τ-LevelIndex. This can be achieved by checking the cardinality of top-ℓ result set $\mathcal{R}$ when incurring the Insert function on cell C. We omit it in Algorithm 1 for the ease of presentation. Before analyzing the time complexity of IBA, we introduce two optimizations to improve its efficiency.

Option filtering. Given a dataset $\mathcal{D}$, the τ-LevelIndex construction problem only needs to consider options in the first τ layers of skylines. The reason is that the other options cannot rank top-τ for any user weight vector in preference space, and thus can be omitted when building τ-LevelIndex. For example, option *Royalton*($\mathbf{r}_5$) in Figure 2(a) will not contribute to the 3-LevelIndex in Figure 2(c) and can be ignored. Thus, we compute the τ-skyband [32] option set of the dataset $\mathcal{D}$, and use it as the input dataset to build τ-LevelIndex.

Insertion ordering. An intuitive solution is to randomly assign an insertion order for options. However, insertion ordering can have a significant influence on the performance of IBA. The reason is obvious—with a poor insertion ordering, many redundant cells (which will not appear in the result τ-LevelIndex) will be created. Our second optimization for IBA is to insert options according to their skyline layers, i.e., options at lower (e.g., the first) skyline layers are inserted before options at higher (e.g., the second) skyline layers. We investigate the effect of the insertion ordering by the experiments shown in Figure 11, Section 7.

[5]http://lpsolve.sourceforge.net/5.5/

Algorithm 1 IBA(Dataset $\mathcal{D}$,value τ)

```
1:  Initialize $\tau$-LevelIndex entry cell $C_0$ with $\mathbf{r}_\ell \leftarrow nil$
2:  for each option $\mathbf{r}_j \in \mathcal{D}$ do
3:      Result set $\mathcal{R} \leftarrow \emptyset$
4:      Insert($C_0, \mathbf{r}_j, \mathcal{R}$)
5:      Merge and Eliminate cells in $\tau$-LevelIndex
6:  Return $\tau$-LevelIndex

    Routine Insert(cell $C$, option $\mathbf{r}_j$, result set $\mathcal{R}$):
7:  $C$.IsVisited ← True
8:  if $\mathcal{H}^+_{\ell,j} \supseteq$ G($C$) then                          ▷ Case I
9:      if $C$ is an internal node then
10:         for each child $c$ of $C$ and $c$.IsVisited is False do
11:             Insert($c, \mathbf{r}_j, \mathcal{R} \cup \{\mathbf{r}_\ell\}$)
12:     else
13:         Initialize cell $C'$ by setting $\mathbf{r}_\ell \leftarrow \mathbf{r}_j$
14:         Set $C'$ as the child of $C$
15: else if $\mathcal{H}^-_{\ell,j} \supseteq$ G($C$) then                     ▷ Case II
16:     Clone $C$ to $C'$, and set $\mathbf{r}_\ell$ of $C'$ as $\mathbf{r}_j$
17:     Set $C$ as the child of $C'$
18: else                                                 ▷ Case III
    // the part $\mathcal{S}_\mathbf{w}(\mathbf{r}_\ell) \le \mathcal{S}_\mathbf{w}(\mathbf{r}_j)$ in cell $C$
19:     Clone $C$ to $C'$ and set $\mathbf{r}_\ell$ of $C'$ as $\mathbf{r}_j$
20:     Clone $C$ to a new cell $C''$
21:     Set $C''$ as the child of $C'$
    // the part $\mathcal{S}_\mathbf{w}(\mathbf{r}_\ell) \ge \mathcal{S}_\mathbf{w}(\mathbf{r}_j)$ in cell $C$
22:     for each child $c$ of $C$ and $c$.IsVisited is False do
23:         Insert($c, \mathbf{r}_j, \mathcal{R} \cup \{\mathbf{r}_\ell\}$)
```

Complexity analysis. We analyze the time complexity of IBA in Algorithm 1 in Lemma 1.

Lemma 1. *Given a d-dimensional dataset $\mathcal{D}$, the time complexity of using the insertion-based approach* IBA *to construct τ*-LevelIndex *is $O(m^{d-1})$, where m is the cardinality of the τ-skyband set of $\mathcal{D}$.*

Proof. The computational cost of IBA is roughly equal to the total number of cells in τ-LevelIndex. Algorithm 1 inserts all products in the τ-skyband of dataset $\mathcal{D}$, i.e., m options in total, into a d-1 dimensional preference space. Thus, the number of cells in τ-LevelIndex is $O(m^{d-1})$ according to the zone theorem in [16]. □

6 PARTITION-BASED APPROACH: PBA

In this section, we devise a partition-based approach (PBA) to build τ-LevelIndex incrementally (i.e., level by level). PBA outperforms IBA by (i) employing a partition-based method to compute the cells in τ-LevelIndex, instead of inserting options one by one as in IBA, and (ii) ignoring the unqualified options when partitioning each cell in τ-LevelIndex.

6.1 Observations from IBA

We elaborate two observations from IBA, which inspire us to design PBA to reduce the index building time.

Observation I: Redundant cell representation. For each cell C in τ-LevelIndex, its exact geometric region $\mathrm{G}(C)$ is defined by the top-ℓ result set $\mathcal{R}$ implicitly. For example, consider the rank-1st cell C_1 (with top-1 result set $\mathcal{R} = \{VibesInn(\mathbf{r}_1)\}$) in Figure 4(b). According to Definition 2, the geometric region of C_1 is $\mathrm{G}(C_1) = \mathcal{H}^+_{1,2} \cap \mathcal{H}^+_{1,3}$. However, the representation of C_1 can be simplified to: $\mathrm{G}(C_1) = \mathcal{H}^+_{1,2}$. The reason is that the score of $Artezen(\mathbf{r}_2)$ is larger than the score of $citizenM(\mathbf{r}_3)$ in the entire preference space, i.e., $\forall w[1] \in [0, 1], \mathcal{S}_{\mathbf{w}}(\mathbf{r}_2) \geq \mathcal{S}_{\mathbf{w}}(\mathbf{r}_3)$, as shown in Figure 4(b). Thus, it holds $\mathcal{S}_{\mathbf{w}}(\mathbf{r}_1) \geq \mathcal{S}_{\mathbf{w}}(\mathbf{r}_2) \geq \mathcal{S}_{\mathbf{w}}(\mathbf{r}_3)$ for any weight vector $\mathbf{w} \in \mathcal{H}^+_{1,2}$ (i.e., $w[1] \in [0, 0.5]$). Thus, $\mathcal{H}^+_{1,3}$ can be safely omitted in $\mathrm{G}(C_1)$.

Given a rank-ℓ-th cell C and the inserted option $\mathbf{r}_j$, IBA tests the relationship between cell C and hyperplane $\mathcal{H}_{\ell,j}$, then classifies the situation into three cases (see Lines 8, 15, and 18 in Algorithm 1). The number of halfspaces in C determines the cost of the relationship test. Specifically, the halfspace intersection cost is $O(n^{\lfloor d/2 \rfloor})$ when there are n halfspaces. Therefore, our first question is *how to define a cell C concisely*?

Observation II: Unnecessary cell checking. Consider the τ=2-LevelIndex in Figure 4(a) with $citizenM(\mathbf{r}_3)$ being the inserted option. In IBA, $citizenM(\mathbf{r}_3)$ checks every cell in 2-LevelIndex, i.e., the gray cells (C_1, C_2, C_3 and C_4) in Figure 5(a). However, $citizenM(\mathbf{r}_3)$ does not change the level-1 arrangement in 3-LevelIndex, i.e., the two rank-1st cells C_1 and C_2 are not affected, as shown in Figure 5(b). But IBA needs to check $citizenM(\mathbf{r}_3)$ with cells C_1 and C_2 by conducting geometric relationship tests, which obviously wastes computation cost. Therefore, our second question is *how to avoid unnecessary cell checking*?

In addition to the above inefficiencies, IBA processes each option in the dataset $\mathcal{D}$ one by one. After each iteration, it returns τ-LevelIndex with the options that have been inserted so far. Thus, it cannot build the τ-LevelIndex incrementally, i.e., building the τ-LevelIndex upon the $(\tau-1)$-LevelIndex of the input dataset $\mathcal{D}$.

6.2 PBA **Algorithm**

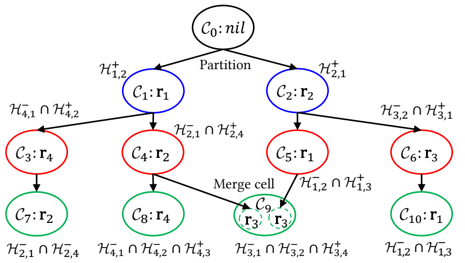
Our partition-based approach PBA is designed to solve the two inefficiencies of IBA. In particular, PBA avoids redundant halfspaces in each rank-ℓ-th cell C by observing a crucial property among the halfspaces, which we will present shortly. Moreover, to resolve the unnecessary checking issue in IBA and build τ-LevelIndex incrementally, PBA only processes the options that can rank top-(ℓ+1)-th for any possible weight vector $\mathbf{w}$ in each rank-ℓ-th cell C.

Simplified cell representation. Before presenting the partition-based approach PBA, we show an alternative method to define the geometric region of each cell C in τ-LevelIndex in Definition 5.

Definition 5. *Given a dataset $\mathcal{D}$, suppose that cell set Γ includes all rank-$(\ell-1)$-th cells with the same top-$(\ell-1)$ option set $\mathcal{R}_\Gamma$*[6]. *Let $\mathcal{P}$ be the top-ℓ-th option set for any possible weight vector $\mathbf{w}$ that lies in the geometric region of Γ. For the rank-ℓ-th cell C in level-ℓ arrangement of τ-*LevelIndex*, whose top-ℓ-th option is $\mathbf{r}_i \in \mathcal{P}$ and top-ℓ result set is $\mathcal{R} = \mathcal{R}_\Gamma \cup \{\mathbf{r}_i\}$, we define C's geometric region in preference space as follows:*

$$\mathrm{G}'(C) = \Big(\bigcap_{\mathbf{r}_j \in \mathcal{R}-\{\mathbf{r}_i\}} \mathcal{H}^-_{i,j}\Big) \cap \Big(\bigcap_{\mathbf{r}_j \in \mathcal{P}-\{\mathbf{r}_i\}} \mathcal{H}^+_{i,j}\Big). \tag{2}$$

[6]The result set $\mathcal{R}_\Gamma$ is order insensitive.



$\mathcal{C}_0$: *nil*
Partition
$\mathcal{H}^+_{1,2}$
$\mathcal{H}^+_{2,1}$
$\mathcal{C}_1$: $\mathbf{r}_1$
$\mathcal{C}_2$: $\mathbf{r}_2$
$\mathcal{H}^-_{4,1} \cap \mathcal{H}^+_{4,2}$
$\mathcal{H}^-_{2,1} \cap \mathcal{H}^+_{2,4}$
$\mathcal{H}^-_{3,2} \cap \mathcal{H}^+_{3,1}$
$\mathcal{C}_3$: $\mathbf{r}_4$
$\mathcal{C}_4$: $\mathbf{r}_2$
$\mathcal{C}_5$: $\mathbf{r}_1$
$\mathcal{C}_6$: $\mathbf{r}_3$
Merge cell
$\mathcal{H}^-_{1,2} \cap \mathcal{H}^+_{1,3}$
$\mathcal{C}_7$: $\mathbf{r}_2$
$\mathcal{C}_8$: $\mathbf{r}_4$
$\mathcal{C}_9$
$\mathbf{r}_3$
$\mathbf{r}_3$
$\mathcal{C}_{10}$: $\mathbf{r}_1$
$\mathcal{H}^-_{2,1} \cap \mathcal{H}^-_{2,4}$
$\mathcal{H}^-_{4,1} \cap \mathcal{H}^-_{4,2} \cap \mathcal{H}^+_{4,3}$
$\mathcal{H}^-_{3,1} \cap \mathcal{H}^-_{3,2} \cap \mathcal{H}^+_{3,4}$
$\mathcal{H}^-_{1,2} \cap \mathcal{H}^-_{1,3}$

Figure 6: Partition-based approach PBA **example**

Lemma 2 serves as a basic building brick of the partition-based approach PBA.

Lemma 2. *The geometric region of rank-ℓ-th cell C in Definition 2 is equivalent to the geometric region in Definition 5, i.e.,* $\mathrm{G}(C) = \mathrm{G}'(C)$.

Proof. According to Definition 2, the geometric region of cell C is: $\mathrm{G}(C) = \big(\cap_{\forall \mathbf{r}_j \in \mathcal{R}-\{\mathbf{r}_i\}} \mathcal{H}^-_{i,j}\big) \cap \big(\cap_{\forall \mathbf{r}_j \in \mathcal{D}-\mathcal{R}} \mathcal{H}^+_{i,j}\big)$. We first define the geometric region of Γ via Definition 2 as follows: $\mathrm{G}(\Gamma) = \bigcup_{\forall C_j \in \Gamma} \mathrm{G}(C_j)$. Since $\mathcal{P}$ includes all possible top-ℓ-th options for any weight $\mathbf{w}$ in $\mathrm{G}(\Gamma)$, we have $\forall \mathbf{r}_i \in \mathcal{P}$ and $\forall \mathbf{r}_j \in \mathcal{D} - \{\mathcal{R} \cup \mathcal{P}\}, \mathcal{S}_{\mathbf{w}}(\mathbf{r}_i) > \mathcal{S}_{\mathbf{w}}(\mathbf{r}_j)$. It means that $\mathcal{H}^+_{i,j}$ holds for any weight $\mathbf{w}$ in $\mathrm{G}(\Gamma)$. Next, it holds that $\mathrm{G}(\Gamma) \supseteq \mathrm{G}(C)$ as the top-ℓ result set of rank-ℓ-th cell C is $\mathcal{R} = \mathcal{R}_\Gamma \cup \{\mathbf{r}_i\}$. Then, we conclude that $\forall \mathbf{r}_j \in \mathcal{D} - \{\mathcal{R} \cup \mathcal{P}\}, \mathcal{H}^+_{i,j} \supseteq \mathrm{G}(\Gamma) \supseteq \mathrm{G}(C)$. It means that $\forall \mathbf{r}_j \in \mathcal{D} - \{\mathcal{R} \cup \mathcal{P}\}$, halfspace $\mathcal{H}^+_{i,j}$ covers $\mathrm{G}(C)$. Thus, $\mathcal{H}^+_{i,j}$ can be removed from $\mathrm{G}(C)$ as it does not bound the geometric region of C in preference space. After removing all these halfspaces, it holds that $\mathrm{G}(C) = \big(\cap_{\forall \mathbf{r}_j \in \mathcal{R}-\{\mathbf{r}_i\}} \mathcal{H}^-_{i,j}\big) \cap \big(\cap_{\forall \mathbf{r}_j \in \mathcal{P}-\{\mathbf{r}_i\}} \mathcal{H}^+_{i,j}\big) = \mathrm{G}'(C)$. □

Example. Given the hotel dataset in Figure 2(a), suppose we want to build its 3-LevelIndex. Consider the entry cell C_0 with its top-1st hotel set $\mathcal{P} = \{VibesInn(\mathbf{r}_1), Artezen(\mathbf{r}_2)\}$. According to Lemma 2, the geometric region of C_1 (with top-1st option $VibesInn(\mathbf{r}_1)$) is $\mathrm{G}'(C_1) = \mathcal{H}^+_{1,2}$ (in Definition 5), as shown in Figure 6, which is equivalent to $\mathrm{G}(C_1) = \mathcal{H}^+_{1,2} \cap \mathcal{H}^+_{1,3} \cap \mathcal{H}^+_{1,4} \cap \mathcal{H}^+_{1,5}$ in Definition 2 because $citizenM(\mathbf{r}_3)$, $Yotel(\mathbf{r}_4)$ and $Royalton(\mathbf{r}_5)$ cannot rank top-1st for any weight $\mathbf{w}$ in the entire preference space C_0, as their scores shown in Figure 2(b).

Corollary 1. *Given a rank-ℓ-th cell C, let $\mathcal{P}$ be the rank top-$(\ell+1)$-th option set for any weight $\mathbf{w}$ that lies inside C. It holds that for every child cell of C, its top-$(\ell+1)$-th option is in $\mathcal{P}$.*

Proof. The proof is trivial as $\mathcal{P}$ includes all possible top-(ℓ+1)-th options for any weight $\mathbf{w}$ in the geometric region of cell C. □

Partition-based approach. With Lemma 2 and Corollary 1, the main idea of the partition-based approach PBA is as follows. Given a rank-ℓ-th cell C, we first compute the set of possible top-$(\ell+1)$-th options $\mathcal{P}$ for any weight $\mathbf{w}$ in the geometric region of C, then create C's rank-$(\ell+1)$-th child cells by verifying the feasibility of each option in $\mathcal{P}$. Finally, we obtain the geometry region of all rank-$(\ell+1)$-th cells in level-$(\ell+1)$ arrangement of τ-LevelIndex with Equation (2) and Lemma 2.

Take the rank-1st cell C_1 in Figure 6 as an example, its possible top-2nd option set $\mathcal{P} = \{Artezen(\mathbf{r}_2), Yotel(\mathbf{r}_4)\}$, and each option leads to a rank-2nd cell. Applying Equation (2), the rank-2nd cells of C_1 are C_3 and C_4 with geometric regions $\mathcal{H}^-_{4,1} \cap \mathcal{H}^+_{4,2}$, and $\mathcal{H}^-_{2,1} \cap \mathcal{H}^+_{2,4}$, respectively, as shown in Figure 6.

Algorithm 2 PBA(Dataset $\mathcal{D}$,value τ)

```
1:  Initialize $\tau$-LevelIndex entry cell $C_0$ with $C_0.\mathbf{r}_\ell \leftarrow nil$
2:  Initialize top-$\ell$ set $C_0.\mathcal{R} \leftarrow \emptyset$, bounding option set $C_0.\mathrm{B} \leftarrow \emptyset$
3:  for each $\ell$ from 0 to $\tau - 1$ do
4:      for each rank-$\ell$-th cell $C$ in $\tau$-LevelIndex do
5:          $\mathcal{P} \leftarrow$ ComputeP($C, \mathcal{D}$)                ▷ Section 6.3
6:          Partition($C, \mathcal{P}$)
7:      Merge cells at level-$(\ell+1)$ arrangement in $\tau$-LevelIndex
8:  Return $\tau$-LevelIndex

    Routine Partition(cell $C$, top-$(\ell+1)$-th set $\mathcal{P}$):
9:  for each $\mathbf{r}_i \in \mathcal{P}$ do
    // Verify whether $\mathbf{r}_i$ can rank top-$(\ell+1)$-th in $C$
10:     if $\left(\cap_{\mathbf{r}_j \in \mathcal{P}-\{\mathbf{r}_i\}} \mathcal{H}^+_{i,j}\right) \cap \mathrm{G}'(C) \neq \emptyset$ then
11:         Clone $C$ to $C'$, $C'.\mathbf{r}_\ell \leftarrow \mathbf{r}_i$
12:         $C'.\mathcal{R} \leftarrow C.\mathcal{R} \cup \{\mathbf{r}_i\}$, $C'.\mathrm{B} \leftarrow \mathcal{P} - \{\mathbf{r}_i\}$
13:         Set $C'$ as the child of $C$
```

Algorithm 2 shows the pseudocode of the partition-based approach PBA. For each cell C, $C.\mathcal{R}$ and $C.\mathbf{r}_\ell$ store its top-ℓ result set and top-ℓ-th option, respectively. Moreover, it employs C.B to store the options that bound the geometric region of C. Considering level ℓ in τ-LevelIndex, PBA processes each rank-ℓ-th cell C individually (see Line 4). At Line 5, it computes the top-$(\ell+1)$ option set $\mathcal{P}$ for any weight vector in C. Then it incurs the Partition function to compute C's rank-$(\ell+1)$-th child cells in Line 6. The Merge function is called to merge the rank-$(\ell+1)$-th cells that share the same top-$(\ell+1)$ result set and top-$(\ell+1)$-th option, which is the same as Merge in the insertion-based approach (see Line 5 in Algorithm 1). In addition, for each rank-$(\ell+1)$-th cell C, its C.B unions all bounding options of each merged cells.

For each cell C at level ℓ, it includes two key functions (i.e., ComputeP and Partition) to compute its child cells at level $(\ell+1)$. The routine ComputeP (in Line 5) computes the top-$(\ell+1)$-th option set $\mathcal{P}$ for every user preference weight in cell C, we will elaborate it in Section 6.3. The routine Partition (in Line 6) shows the details to compute the rank-$(\ell+1)$-th child cells of C with the top-$(\ell+1)$-th option set $\mathcal{P}$. For each candidate top-$(\ell+1)$-th option $\mathbf{r}_i$, PBA tests its corresponding exact preference region by applying Equation (2) at Line 10. If it is feasible, PBA creates C's rank-$(\ell+1)$-th child cell C', sets its top-$(\ell+1)$-th option as $\mathbf{r}_i$, includes $\mathbf{r}_i$ into the top-$(\ell+1)$ result set, and let the options in $\mathcal{P}$ (except $\mathbf{r}_i$) be the bounding option set in C'.B.

Example. Consider the example in Figure 6, the entry cell C_0 is divided into the rank-1st cells C_1 and C_2 when $\ell = 0$ in Algorithm 2. For each rank-1st cell, we first compute its corresponding top-2nd option set $\mathcal{P}$ via the ComputeP function, and then obtain its rank-2nd cells by the Partition function. Thus, we have that C_1's rank-2nd child cells are C_3 and C_4, and C_2's rank-2nd child cells are C_5 and C_6, which form the level-2 arrangement in 3-LevelIndex. With the same partition procedure, PBA computes the cells in level-3 arrangement. After applying the Merge function, the rank-3rd cell C_9 in 3-LevelIndex is merged from the rank-3rd child cells of C_4 and C_5, as both cells have the same top-3 result set $\mathcal{R} = \{VibesInn(\mathbf{r}_1), Artezen(\mathbf{r}_2), citizenM(\mathbf{r}_3)\}$ and the same top-3rd option $citizenM(\mathbf{r}_3)$.

Complexity Analysis. In Lemma 3, we analyze the time cost of PBA, which outperforms IBA since τ is much smaller than m.

Lemma 3. *Given a d-dimensional dataset $\mathcal{D}$, the time complexity of the partition-based approach* PBA *in Algorithm 2 to build* τ-LevelIndex *is $O(m^{\lfloor d/2 \rfloor} \tau^{\lceil d/2 \rceil + 1})$, where m is the size of the τ-skyband set of $\mathcal{D}$.*

Proof. The main time cost of PBA is computing the cells in each level of τ-LevelIndex. For $\forall \ell \in [1, \tau]$, use m_ℓ to denote the number of options that could be the top-ℓ-th option in the preference space, it holds $m_\ell \leq m$. The number of cells in each level is $O(m_\ell^{\lfloor d/2 \rfloor} \ell^{\lceil d/2 \rceil})$ [8], and thus the total number of cells in τ-LevelIndex is: $O(\sum_{\ell=1}^{\tau} m_\ell^{\lfloor d/2 \rfloor} \ell^{\lceil d/2 \rceil}) \leq O(m^{\lfloor d/2 \rfloor} \cdot \tau \cdot \tau^{\lceil d/2 \rceil}) = O(m^{\lfloor d/2 \rfloor} \tau^{\lceil d/2 \rceil + 1})$. □

Index updating. For a new arriving option $\mathbf{r}$, IBA inserts it into the τ-LevelIndex accordingly. For other updates (e.g., option deletion, adding/removing option attributes), we suggest users constructing τ-LevelIndex via PBA approach from scratch as PBA is very efficient.

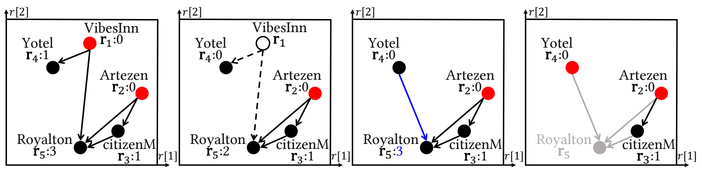
6.3 Fast Candidate Set Computation

As we discussed in Section 6.2, computing the top-$(\ell+1)$-th option set $\mathcal{P}$ of rank-ℓ-th cell C is a key subroutine in the partition-based approach PBA. In this section, we improve the performance of PBA by utilizing the dominance relation among the options to compute the top-$(\ell+1)$-th option set $\mathcal{P}$ for each cell C in Algorithm 2.

Given the hotel dataset in Figure 2(a) and $\tau = 3$, we elaborate the core idea to accelerate $\mathcal{P}$ computation in PBA by a running example in Figure 7. Inspired by [24, 30, 50], we employ a directed acyclic graph (DAG) to maintain the pair-wise dominance relationship among options. We first construct the DAG of the option dataset by testing the dominance relationship among every pair of options. For example, $VibesInn(\mathbf{r}_1)$ dominates $Yotel(\mathbf{r}_4)$, thus, there is an arrow from $VibesInn(\mathbf{r}_1)$ to $Yotel(\mathbf{r}_4)$ in the DAG, as shown in Figure 7(a). For each option in DAG, we attach the number of its dominating options with it. For example, the number of dominators of $Royalton(\mathbf{r}_5)$ in Figure 7(a) is 3, i.e., $VibesInn(\mathbf{r}_1)$, $Artezen(\mathbf{r}_2)$ and $citizenM(\mathbf{r}_3)$ dominate it.

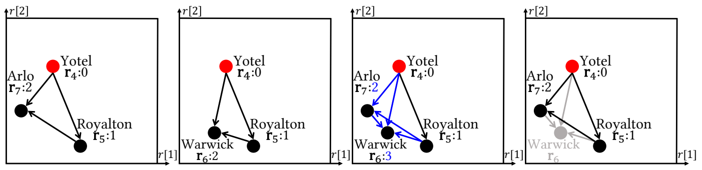
Observation 1. *Given the entry cell C_0 with its dominance graph* DG_0, *option $\mathbf{r}$ is in the top-1st option set $\mathcal{P}$ for any weight $\mathbf{w}$ in C_0 iff the in-degree of $\mathbf{r}$ is 0 in* DG_0.

Take the hotel dataset in Figure 7(a) as an example, the top-1st option set is $\mathcal{P} = \{VibesInn(\mathbf{r}_1), Artezen(\mathbf{r}_2)\}$ for any weight $\mathbf{w}$ in the entry cell C_0. Obviously, the above observation also holds for



(a) DG_0 (b) Inheriting DG_1 (c) Updating DG_1 (d) Pruning

Figure 7: Maintaining dominance graph for cell partition



(a) DG_i (b) DG_j (c) Merging (d) Pruning

Figure 8: Maintaining dominance graph for cell merging

each rank-ℓ-th cell C_i and its corresponding dominance graph DG_i. The reason is that the other options in DG_i cannot be the top-(ℓ+1)-th option for any weight $\mathbf{w}$ in C_i as each of them is dominated by at least one option whose in-degree is 0.

Maintain dominance graph for cell partition. Lemma 4 summarizes the dominance relationship among the options in cell C and its child cell C'.

Lemma 4. *Suppose the dominance graph of cell C is* DG*, the dominance relationship among the options in* DG *also holds for every weight* $\mathbf{w}$ *in C's child cell C'.*

Proof. The proof is straight forward as any weight $\mathbf{w}$ in C' is also in C. □

Returning to the running example, cell C_1 in Figure 6 is a rank-1st child cell of C_0 with top-1st option *VibesInn*($\mathbf{r}_1$). With Lemma 4, the dominance relationship in DG_0 in Figure 7(a) also holds in the dominance graph DG_1 in Figure 7(b). *VibesInn*($\mathbf{r}_1$) and its edges are removed from DG_1 as it is the top-1st option in C_1, as the circle and dotted arrows shown in Figure 7(b). Although *Yotel*($\mathbf{r}_4$) does not dominate *Royalton*($\mathbf{r}_5$) in cell C_0, *Yotel*($\mathbf{r}_4$) dominates *Royalton*($\mathbf{r}_5$) in C_0's child cell C_1. Thus, we update the dominance relationship among the options for C_1, as the added blue edge shown in Figure 7(c). The number of dominators of each option in DG_1 can be also updated accordingly. Since C_1 is in level-1 arrangement, it only needs to obtain the cells in next $\tau-1 = 2$ levels. Thus, options whose number of dominators exceeds 2 can be pruned as they cannot be top-τ options for any weight $\mathbf{w}$ in C_1. For example, *Royalton*($\mathbf{r}_5$) is pruned as the number of dominating options is 3 in C_1, see the gray cell in Figure 7(d). Next, the rank-1st cell C_1 will partition into rank-2nd cell with its top-2nd option set $\mathcal{P} = \{Yotel(\mathbf{r}_4), Artezen(\mathbf{r}_2)\}$, as the in-degrees of *Yotel*($\mathbf{r}_4$) and *Artezen*($\mathbf{r}_2$) are 0 in its dominance graph DG_1, see the red points in Figure 7(d).

Maintain dominance graph for cell merge. To guarantee the correctness of PBA, we also need to merge the dominance graphs when PBA incurs Merge function to merge cells at the same level. For example, rank-3rd cell C_9 in Figure 6 is merged from two cells C_i and C_j, which are split from C_4 and C_5, respectively. We illustrate the dominance graph merging procedure by building τ = 5-LevelIndex in Figure 8. Suppose the dominance graphs of C_i and C_j are shown in Figures 8(a) and (b), respectively. We include two extra hotels *Warwick*($\mathbf{r}_6$) and *Arlo*($\mathbf{r}_7$) in this example for clear presentation. To compute the dominance graph of the merged new cell C_9, we union the node set and intersect the edge set of DG_i and DG_j. For example, the union node set is $\{Yotel(\mathbf{r}_4), Royalton(\mathbf{r}_5), Warwick(\mathbf{r}_6), Arlo(\mathbf{r}_7)\}$ and the intersection edge set is the black arrow shown in Figure 8(c). Next, we test the pair-wise dominance relationship among the options w.r.t. the merged cell C_9, the blue arrows shown in Figure 8(c) are added. The unqualified option in C_9 is pruned by verifying the number of its dominators in DG_9. For example, the gray option *Warwick*($\mathbf{r}_6$) in Figure 8(d) is pruned as it has more than two dominators, thus it cannot be the top-5-th result for any weight $\mathbf{w}$ in rank-3rd cell C_9.

In summary, we utilize the dominance graph to maintain the domination relationship among the options in each cell, which avoids expensive *r-skyband* function call for each cell by exploiting the inheritance of the dominance graphs among parent-child cells. It improves the performance of ComputeP function in PBA significantly as (i) the top-(ℓ+1)-th option set $\mathcal{P}$ for each cell can be obtained by collecting all options with in-degree 0 in its dominance graph directly; and (ii) the unqualified candidate options in each cell can be pruned via the number of its dominators in the dominance graph efficiently.

7 EXPERIMENTAL EVALUATION

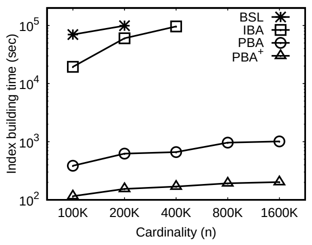
In this section, we conduct extensive experiments for performance evaluation. We introduce the experimental settings in Section 7.1, evaluate different approaches to build τ-LevelIndex in Section 7.2. In Section 7.3, we investigate the performance of processing three representative queries (i.e., kSPR, UTK and ORU) by using τ-LevelIndex.

7.1 Experimental Settings

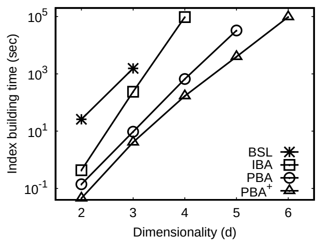
Datasets. We use both synthetic- and real- datasets for the experiments. The synthetic datasets are generated following a standard benchmark for preference-based queries [11]. In particular, the attributes of an option can be configured to follow different distributions, i.e., independent (IND), positively correlated (COR), negatively correlated (or anti-correlated, ANTI). We use the synthetic datasets to explore the influence of different factors (e.g., dataset cardinality, data dimensionality, and data distribution). We also include 3 real datasets that are widely used by related work [36, 38]. Specifically, HOTEL includes 419K options, and each option is a hotel with 4 attributes, i.e., no. of stars, no. of rooms, no. of facilities and price [2]. Each house in HOUSE has 6 attributes including gas, electricity, water, heating, insurance and property tax [3], and there are a total of 315K options. NBA includes statistics of 21.9K NBA players, and models an NBA player with 8 professional metrics, i.e., games, rebounds, assists, steals, blocks, turnovers, personal fouls and points [4].

Approaches for index construction. We compare the following approaches for the τ-LevelIndex building problem. We implemented all our approaches by C++ and posted the source code at [5] for reproducibility.

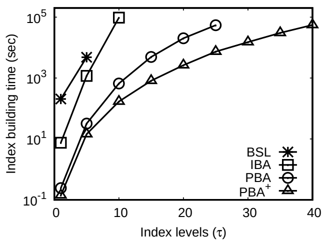
- BSL, UTK_2-adapted baseline approach in Section 5.1.



(a) effect of cardinality n

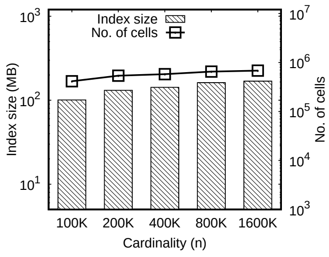


(b) effect of dimensionality d

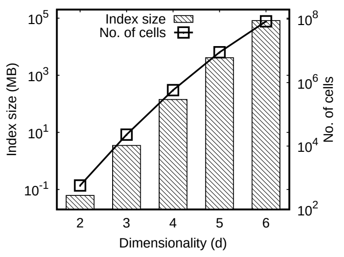


(c) effect of τ

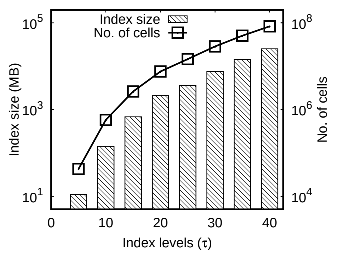
Figure 9: τ-LevelIndex **construction time evaluation, data distribution:** IND



(a) effect of cardinality n



(b) effect of dimensionality d



(c) effect of τ

Figure 10: τ-LevelIndex **index size evaluation, data distribution:** IND

Table 3: Parameter settings for experiments

Parameter	Value
Cardinality n	100K, 200K, **400K**, 800K, 1600K
Dimensionality d	2, 3, **4**, 5, 6
Value τ	1, 5, **10**, 15, 20, 30, 40
Distribution	COR, **IND**, ANTI

- IBA, the insertion-based approach in Algorithm 1.
- PBA, basic partition-based approach which employs *r-skyband* to compute the candidate option set $\mathcal{P}$, see Algorithm 2.
- PBA$^+$, advanced partition-based approach using the fast candidate set computation techniques in Section 6.3.

Query processing. For processing kSPR, UTK and ORU queries, we compare τ-LevelIndex with their state-of-the-art solutions, i.e., look-ahead progressive cell tree approach LP-CTA [37] for kSPR query, joint arrangement approach JAA [30] for UTK query and the state-of-the-art approach, denoted as ORU [36], for ORU query. For these solutions, we used the C++ source code obtained from the authors of the original papers. Please note that all state-of-the-art solutions for the above queries employed Rtree or its variants (e.g., aggregate Rtree) as index to shortlist the candidate options.

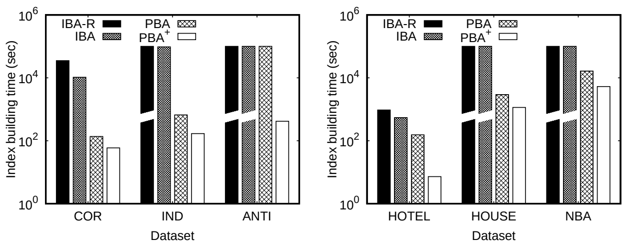
All experiments are conducted in single thread mode on a machine equipped with Intel Gold-5122 3.60 GHZ CPU and 128 GB RAM, running on Ubuntu 18.04. For the index building approaches, we compare their memory consumption and running time. Table 3 lists the tested parameters and the default settings are marked in bold. Before building τ-LevelIndex, we adopt k-skyband [32] and k-onionlayer [12] to filter the dataset $\mathcal{D}$. For query processing, we compare the query processing time. The filtered datasets and indices are stored in RAM for both index building and query processing. To conduct the evaluation in reasonable time, we terminate an experiment if it cannot finish in 10^5 seconds (about 28 hours) and omit its result.

7.2 τ-LevelIndex Building Evaluation

Figure 9 reports the influence of dataset cardinality n, data dimensionality d, and the number of levels in τ-LevelIndex on the index building time of the 4 index building approaches. The results show that for all solutions, index building time increases slowly with dataset cardinality but much faster with dimensionality and index levels. This is because different factors have different effects on the number of cells in τ-LevelIndex as illustrated in Figure 10, and the time cost of all solutions are roughly proportional to the number of cells. Figure 9 also shows that PBA$^+$ runs significantly faster than BSL, IBA and PBA across all cases, and the speedup can be 2-3 orders of magnitude. Moreover, there is a trend that the speedup of PBA$^+$ increases when the problem is more difficult (i.e., when n, τ, d increase). We omit the UTK$_2$-adapted approach BSL for the rest of the paper as it is unacceptably slow.

Next, we report the size of τ-LevelIndex and the number of cells in τ-LevelIndex in Figure 10 on IND, by varying dataset cardinality n, data dimensionality d, and the number of levels τ for PBA$^+$. The results show that the index size is affordable in all cases. In particular, the index size is only 142.26MB under the default setting (i.e., $n = 400K$, $d = 4$ and $\tau = 10$). This is because we represent each cell in τ-LevelIndex implicitly, which reduces the space cost significantly compared with explicit representation, e.g., halfspace-based method, vertex-based method in the literature [31, 37, 48]. As illustrated in Figure 10(a), increasing dataset cardinality only causes a sub-linear increase in the number of cells. This is because only some of the options can rank top-τ and are used to define cells. In contrast, as shown in Figures 10(b) and (c), the number of cells grows super-linearly with dimensionality d and the number of levels τ. This explains why the index building time grows faster with d and τ than with n in Figure 9.

We then experimented the index building approaches on different data distributions and real datasets in Figure 11. Additionally,



(a) synthetic datasets (b) real datasets

Figure 11: Index construction time for different datasets

Table 4: Effectiveness analysis on PBA$^+$ (IND, τ=10)

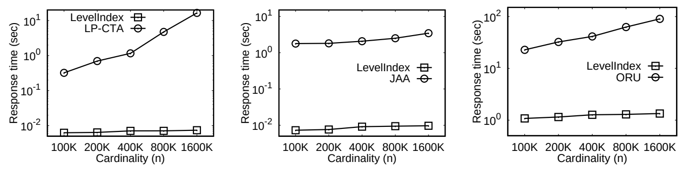
Level	Post-filter candidates	Actual candidates	Hyperplanes for IBA	Hyperplanes for PBA$^+$
3	7.33	5.93	3.68K	37.9
6	4.57	4.10	3.68K	17.2
9	4.04	3.71	3.68K	13.9

we also evaluate the performance of IBA-R, which inserts options randomly. Figures 11(a) and (b) show that PBA$^+$ consistently outperforms PBA, IBA and IBA-R. First, index building time increases from COR to IND and is the largest for ANTI as ANTI produces more cells in τ-LevelIndex than COR and IND. Second, the building time cost is longer for NBA than for HOTEL and HOUSE because the dimensionality of NBA (i.e, 8d) is larger than HOTEL (with 4d) and HOUSE (with 6d), which also leads to more cells and more complex intersection of halfspaces. The broken bars denote that index building procedures are terminated after reaching the time limit, i.e., 10^5 seconds. Last but not least, since IBA used the suggested insertion order we proposed in Section 5.2, which avoids creating many unnecessary cells in τ-LevelIndex, IBA always outperforms IBA-R in all cases, as illustrated in Figure 11.

The advantage of the partition-based approaches (i.e., PBA and PBA$^+$) over the insertion-based approach (IBA) is due to candidate set filtering as reported in Table 4. *Post-filter candidates* is the average number of candidate options for each cell after ComputeP while *Actual candidates* is the average number of options actually used to partition each cell. The results show that most of the post-filter candidates are useful, which means that PBA and PBA$^+$ can avoid checking many unnecessary options for cell partitioning. In contrast, IBA needs to check all options that pass the initial filtering with τ-skyband. The last two columns of Table 4 report the average number of hyperplanes in the representation of each cell in different levels for IBA and PBA$^+$ (same as PBA). The results show that PBA$^+$ uses significantly fewer hyperplanes to represent each cell than IBA. The number of hyperplanes is constant for different levels in IBA because every option in the τ-skyband will contribute a halfspace in each cell in τ-LevelIndex according to Definition 2. In contrast, the number of hyperplanes decreases with levels for PBA$^+$ because fewer options can enter the top-ℓ option set *for a specific cell* as the cells become smaller when the level increases. Thus, compared with IBA, PBA$^+$ significantly reduces the number of hyperplanes in each cell , e.g., from 3.68K to 13.9.

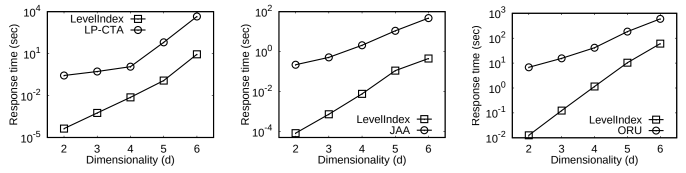
7.3 Query Processing on τ-LevelIndex

In this section, we compare the response time of the queries when processed with τ-LevelIndex and when using their state-of-the-art



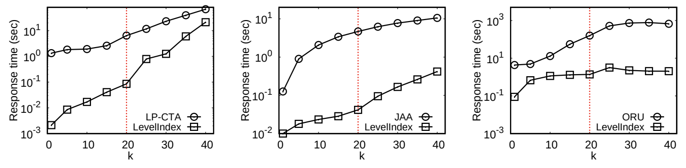
(a) kSPR (b) UTK, σ = 1% (c) ORU, m = 50

Figure 12: Effect of n**, with** τ=20, k = 10, **on** IND



(a) kSPR (b) UTK, σ = 1% (c) ORU, m = 50

Figure 13: Effect of d**, with** τ=20, k = 10, **on** IND



(a) kSPR (b) UTK, σ = 1% (c) ORU, m = 50

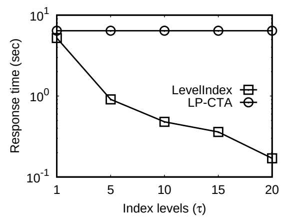
Figure 14: Effect of k**, with** τ=20-LevelIndex, **on** IND

Table 5: Average visited cells for each query (IND)

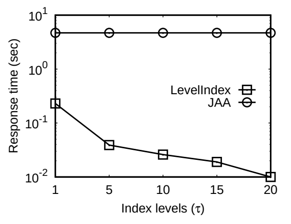
n	100K	200K	400K	800K	1600K
kSPR	4.22K	4.59K	4.73K	5.09K	5.36K
UTK	491	570	453	481	501
ORU	8.66K	8.98K	7.59K	7.03K	6.91K
d	2	3	4	5	6
kSPR	62	656	5.36K	42.4K	328K
UTK	40	132	453	7.28K	75.5K
ORU	511	2.47K	7.59K	22.7K	62.6K

solutions in the literature. For all experiments, we build τ = 20-LevelIndex for the given dataset using PBA$^+$. Figures 12 and 13 illustrate the response time of three representative queries by varying dataset cardinality n and dimensionality d, respectively. The results show that LevelIndex consistently outperforms the state-of-the-art solutions by a large margin for all three queries (i.e., kSPR, UTK and ORU) and under different configurations. In particular, with k = 10 and n = 1600K, τ-LevelIndex speeds up state-of-the-art solutions by 2,361x, 366x and 71x for kSPR, UTK and ORU, respectively. The superior performance of τ-LevelIndex can be explained by the fact that it pre-computes cells on level-1 to level-τ and can answer queries with simple lookup. For τ-LevelIndex, the response time is longer for ORU than for kSPR and UTK because ORU computes the distances from the point to convex sets during lookup while kSPR and UTK compute simpler set containment and intersection tests.

Figure 12 and Figure 13 also show that the query processing time of τ-LevelIndex stays stable when increasing n but grows quickly when increasing d. This can be explained by the different effects of n and d on the number of visited cells during query processing as the query processing cost is proportional to the number of visited cells for τ-LevelIndex. The first four rows of Table 5 show that the number of visited cells does not change much with n for all three

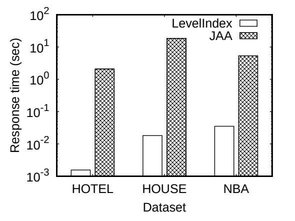


(a) kSPR

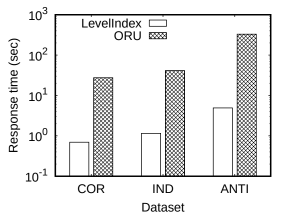


(b) UTK, $\sigma = 1\%$

Figure 15: Effect of τ, with $k = 20$, on IND



(a) UTK on real datasets



(b) ORU on synthetic datasets

Figure 16: Query processing time for different datasets

queries while the number of visited cells increases super-linearly with d as reported in the last four rows of Table 5. In particular, for kSPR, visited cells increase with n because a larger n results in more cells and consequently there are more cells that contain the query option in their top option set. For ORU, visited cells decrease with n because the cells have more diverse top option sets under a larger n, and thus fewer cells are required to collect m options.

In Figure 14, we report the response time when changing k. Note that we use a LevelIndex with $\tau = 20$, which means that when $k > 20$, in addition to looking up the index, τ-LevelIndex also needs to conduct computation to obtain the query results. Figure 14 shows that τ-LevelIndex consistently outperforms the state-of-the-art solutions no matter the index has sufficient levels or not. The reason is that τ-LevelIndex already computes the level-τ cells, on which basis computing the level-k cells (with $k \geq \tau$) is cheaper than from scratch as in the existing solution. Besides, we can also observe a quick increase in response time when τ-LevelIndex switches from pure lookup to lookup-based computation, i.e., for the k values larger than 20, as marked by the dotted lines in Figure 14.

To evaluate the query processing time by varying the values of τ in LevelIndex, we process kSPR and UTK queries with $k = 20$ with different τ values in Figures 15(a) and (b), respectively. As expected, the processing time of both queries drops with the rising of τ value, i.e., from 1 to 20. As a guideline, we recommend to set the parameter τ as the maximum value of k in users' queries. Alternatively, PBA$^+$ can be used to build the τ-LevelIndex index incrementally (see Section 6.2), until users are satisfied with the response time. Thus, users could set a smaller τ first, then expand it on demand.

In Figures 16(a) and (b), we test the performance of processing UTK and ORU queries with 10-LevelIndex on the real and synthetic datasets, respectively. The results show that τ-LevelIndex consistently outperforms the state-of-the-art solutions under different data distributions, i.e., using τ-LevelIndex to process UTK query is 2-3 orders of magnitude faster than the state-of-the-art solution JAA on the three real datasets. For ORU query, the speedup of τ-LevelIndex over state-of-the-art solution are 39.5, 35.9, and 67.1 times on COR, IND and ANTI, respectively.

Table 6: No. of queries to amortize index construction cost

Levels (τ)	10 (k=10 in queries)			20 (k=20 in queries)		
Dataset	HOTEL	HOUSE	NBA	HOTEL	HOUSE	NBA
kSPR	5	19	167	8	40	288
UTK	4	114	1.12K	3	2.26K	2.51K
ORU	1	11	117	1	29	44

To justify the benefits of using τ-LevelIndex, we consider two frameworks, i.e., one that processes queries with the state-of-the-art solutions, and one that builds τ-LevelIndex before query processing. The total time cost of the state-of-the-art solutions consists of only query response time, but the cumulative time cost of τ-LevelIndex includes both index construction time and query processing time. We keep feeding queries and report the minimal number of queries such that τ-LevelIndex solution achieves smaller time cost than state-of-the-art solutions on the real datasets. Specifically, queries with $k = 10$ and 20 are processed on $\tau = 10$ and 20-LevelIndex, respectively. As shown in Table 6, the cost of constructing τ-LevelIndex can be amortized with a moderate number of queries, especially for the more expensive ORU query. However, for UTK query on NBA dataset, it requires running almost 1K to 2K queries to amortize the construction cost as (i) the default query region $\sigma = 1\%$ is very small so that the baselines run faster for UTK queries (i.e., only 13.7s for $k = 10$) than for kSPR/ORU queries (e.g., 105.7s for kSPR queries with the same k), and (ii) index building takes a longer time for NBA with $d = 8$ than for HOTEL ($d = 4$) and HOUSE ($d = 6$).

Finally, to demonstrate the generality of τ-LevelIndex, we investigate the performance of top-k query in DD-type with our τ-LevelIndex based approach. We compare it with the branch-and-bound approach BRS [39]. Both approaches are very fast (e.g., less than 80ms when $k = 20$). But, it is worth pointing out that the performance gain of our LevelIndex-based approach becomes significant when k becomes large. For example, LevelIndex-based approach outperforms BRS by 26.31% when k is 20.

8 CONCLUSION

In this work, we proposed τ-LevelIndex, which can be used to process a spectrum of queries in continuous preference space. We proposed three approaches with several optimization techniques to build τ-LevelIndex efficiently. In particular, we represent each cell in an implicit manner to reduce index size, and devise a fast candidate set computation method to partition the geometric region of each cell. We first conducted extensive experiments to demonstrate the superiority of our proposals in terms of index building time and index size. We then investigated the performance of three queries with τ-LevelIndex. Our solution is faster than their state-of-the-art solutions in the literature by 2 to 3 orders of magnitude. For future work, we plan to enhance τ-LevelIndex to handle dynamic updates of the options for online applications.

ACKNOWLEDGMENTS

This work was supported by grant GRF 152050/19E from the Hong Kong RGC, and the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and a research gift from Huawei.

REFERENCES

[1] 2022. *Booking.* https://www.booking.com/

[2] 2022. *Hotel dataset.* www.hotels-base.com/

[3] 2022. *House dataset.* www.ipums.org/

[4] 2022. *NBA dataset.* www.basketball-reference.com/

[5] 2022. τ-LevelIndex *implementation.* https://github.com/cupermanrose/tau-LevelIndex/

[6] 2022. *TripAdvisor.* https://www.tripadvisor.com/

[7] Pankaj K Agarwal. 2017. Simplex range searching and its variants: A review. *A Journey Through Discrete Mathematics* (2017), 1–30.

[8] Pankaj K Agarwal, Boris Aronov, Timothy M Chan, and Micha Sharir. 1998. On Levels in Arrangements of Lines, Segments, Planes, and Triangles. *Discrete & Computational Geometry* 19, 3 (1998), 315–331.

[9] Pankaj K Agarwal, Mark De Berg, Jiri Matousek, and Otfried Schwarzkopf. 1998. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM journal on computing* 27, 3 (1998), 654–667.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD.* 322–331.

[11] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. 2001. The skyline operator. In *ICDE.* 421–430.

[12] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. 2000. The onion technique: Indexing for linear optimization queries. In *SIGMOD.* 391–402.

[13] Bernard Chazelle and Franco P Preparata. 1986. Halfspace range search: an algorithmic application of k-sets. *Discrete & Computational Geometry* 1, 1 (1986), 83–93.

[14] Paolo Ciaccia and Davide Martinenghi. 2017. Reconciling skyline and ranking queries. *PVLDB* 10, 11 (2017), 1454–1465.

[15] Vida Dujmoviū and Stefan Langerman. 2011. A center transversal theorem for hyperplanes and applications to graph drawing. In *SoCG.* 117–124.

[16] Herbert Edelsbrunner, Joseph O'Rourke, and Raimund Seidel. 1986. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.* 15, 2 (1986), 341–363.

[17] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, and Linlin Zhou. 2015. Answering why-not questions on reverse top-k queries. *PVLDB* 8, 7 (2015), 738–749.

[18] Shen Ge, Nikos Mamoulis, David W Cheung, et al. 2012. Efficient all top-k computation-a unified solution for all top-k, reverse top-k and top-m influential queries. *TKDE* 25, 5 (2012), 1015–1027.

[19] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD.* 47–57.

[20] Zhian He and Eric Lo. 2012. Answering why-not questions on top-k queries. *TKDE* 26, 6 (2012), 1300–1315.

[21] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD* 30, 2 (2001), 259–270.

[22] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *Comput. Surveys* 40, 4 (2008), 1–58.

[23] Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. 2020. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. *Discrete & Computational Geometry* 64, 3 (2020), 838–904.

[24] Jinfei Liu, Li Xiong, Jian Pei, Jun Luo, and Haoyu Zhang. 2015. Finding pareto optimal groups: Group-based skyline. *PVLDB* 8, 13 (2015), 2086–2097.

[25] Qing Liu, Yunjun Gao, Gang Chen, Baihua Zheng, and Linlin Zhou. 2016. Answering why-not and why questions on reverse top-k queries. *The VLDB Journal* 25, 6 (2016), 867–892.

[26] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. 2007. Efficient top-k aggregation of ranked inputs. *TODS* 32, 3 (2007), 19.

[27] Tamara Mchedlidze, Marcel Radermacher, and Ignaz Rutter. 2017. Aligned drawings of planar graphs. In *International Symposium on Graph Drawing and Network Visualization.* 3–16.

[28] Kyriakos Mouratidis, Keming Li, and Bo Tang. 2021. Marrying Top-k with Skyline Queries: Relaxing the Preference Input while Producing Output of Controllable Size. In *SIGMOD.* 1317–1330.

[29] Kyriakos Mouratidis and Hwee Hwa Pang. 2013. Computing immutable regions for subspace top-k queries. *PVLDB* 6, 2 (2013), 73–84.

[30] Kyriakos Mouratidis and Bo Tang. 2018. Exact processing of uncertain top-k queries in multi-criteria settings. *PVLDB* 11, 8 (2018), 866–879.

[31] Kyriakos Mouratidis, Jilian Zhang, and Hwee Hwa PANG. 2015. Maximum rank query. *PVLDB* 8, 12 (2015), 1554–1565.

[32] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *TODS* 30, 1 (2005), 41–82.

[33] Peng Peng and Raymong Chi-Wing Wong. 2015. k-hit query: Top-k query with probabilistic utility function. In *SIGMOD.* 577–592.

[34] Jörg-Rüdiger Sack and Jorge Urrutia. 1999. *Handbook of computational geometry.* Elsevier.

[35] Mohamed A Soliman, Ihab F Ilyas, Davide Martinenghi, and Marco Tagliasacchi. 2011. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *SIGMOD.* 805–816.

[36] Bo Tang, Kyriakos Mouratidis, and Mingji Han. 2021. On m-Impact Regions and Standing Top-k Influence Problems. In *SIGMOD.* 1784–1796.

[37] Bo Tang, Kyriakos Mouratidis, and Man Lung Yiu. 2017. Determining the impact regions of competing options in preference space. In *SIGMOD.* 805–820.

[38] Bo Tang, Kyriakos Mouratidis, Man Lung Yiu, and Zhenyu Chen. 2019. Creating top ranking options in the continuous option and preference space. *PVLDB* 12, 10 (2019), 1181–1194.

[39] Yufei Tao, Vagelis Hristidis, Dimitris Papadias, and Yannis Papakonstantinou. 2007. Branch-and-bound processing of ranked queries. *Information Systems* 32, 3 (2007), 424–445.

[40] Yufei Tao, Xiaokui Xiao, and Jian Pei. 2007. Efficient skyline and top-k retrieval in subspaces. *TKDE* 19, 8 (2007), 1072–1088.

[41] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. 2010. Reverse top-k queries. In *ICDE.* 365–376.

[42] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Norvag. 2011. Monochromatic and bichromatic reverse top-k queries. *TKDE* 23, 8 (2011), 1215–1229.

[43] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. 2010. Identifying the most influential data objects with reverse top-k queries. *PVLDB* 3, 1-2 (2010), 364–372.

[44] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. 2013. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD.* 481–492.

[45] Guolei Yang and Ying Cai. 2017. Querying Improvement Strategies. In *EDBT.* 294–305.

[46] Jianye Yang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2016. Influence based cost optimization on user preference. In *ICDE.* 709–720.

[47] Jianye Yang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2019. Cost optimization based on influence and user preference. *Knowledge and Information Systems* 61, 2 (2019), 695–732.

[48] Albert Yu, Pankaj K Agarwal, and Jun Yang. 2012. Processing a large number of continuous preference top-k queries. In *SIGMOD.* 397–408.

[49] Jilian Zhang, Kyriakos Mouratidis, and HweeHwa Pang. 2014. Global immutable region computation. In *SIGMOD.* 1151–1162.

[50] Lei Zou and Lei Chen. 2010. Pareto-based dominant graph: An efficient indexing structure to answer top-k queries. *TKDE* 23, 5 (2010), 727–741.