

A Framework for Top-k Queries with Constrained Preferences

Kyriakos Mouratidis · Nikolaos Chaloulakos · Bo Tang

Received: date / Accepted: date

Abstract Traditional rank-aware processing assumes a dataset that contains available options to cover a specific need (e.g., restaurants, hotels, etc) and users who browse that dataset via top- k queries with linear scoring functions, i.e., by ranking the options according to the weighted sum of their attributes, for a set of given weights. In practice, however, user preferences (weights) may only be estimated with bounded accuracy, or may be inherently imprecise due to the inability of a human user to specify exact weight values with absolute accuracy. Motivated by this, we define the *constrained-preference top-k* (CT) query. Given an approximate description of the weight values, CT reports all options that may belong to the top- k set. Our CT algorithm assumes that the dataset is indexed with a general-purpose index (e.g., an R-tree) and delivers efficient processing, be it when data and index are in memory, or on the disk. Furthermore, we delve deeper into the special and highly practical case of CT for top-record sets (i.e., $k = 1$), termed CT^1 , and devise a specialized method for it. Our CT^1 algorithm offers *node-access optimality*, i.e., a guarantee to access the minimum number of index nodes. This translates to optimal I/O cost (in the disk-based scenario) and to significant computation savings (which is relevant in both the disk-based and the memory-based scenarios).

K. Mouratidis
School of Computing and Information Systems, Singapore Management University, Singapore
E-mail: kyriakos@smu.edu.sg

N. Chaloulakos
School of Computing and Information Systems, Singapore Management University, Singapore
E-mail: nikolaos.c.2025@phdcs.smu.edu.sg

B. Tang
Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
E-mail: tangb3@sustech.edu.cn

1 Introduction

In the present reality of mobile devices and permanent connectivity, consumers have numerous alternatives at their fingertips for their every need. Examples include the search for a restaurant, hotel, or apartment, but also the search for a home appliance or a service, such as a mobile phone plan or a massage package. Typically, browsing for alternatives happens via a need-specific portal, like TripAdvisor, Yelp, CNET, etc, which provides ratings for each of the key aspects that may interest a customer. In the case of hotels, these aspects may include cleanliness, room quality, service, etc. Naturally, different users may weigh these aspects differently. When multiple decision aspects are involved, user studies [39] have shown that the way real consumers make their choices is accurately captured by a *linear utility* model. That is, in a setting that involves d decision aspects (e.g., ratings), the utility score of an option p is defined as the weighted sum of its per aspect values (attributes), where the weights are user-specific and together form her *preference vector* w . In turn, this enables the shortlisting of the k highest-scoring records via a top- k query [25].

The issue with this model is that although the user may indeed base her decisions on a (latent) w , that vector (whether input directly by the user or learned/mined algorithmically) cannot be known exactly. Consider a user who manually indicates weights for attributes *Service*, *Cleanliness*, and *Location* when looking for a hotel on a hospitality portal. Based on intuition, she may specify weights 0.3, 0.5, and 0.2 for these attributes, respectively. While this makes option ranking unambiguous, it is unrealistic to expect absolute accuracy in the user's input, especially since a minor change in w could significantly alter the top- k set [26,50]. Instead, it would be more practical to allow w a certain leeway for imprecision.

Note that the weight accuracy issue cannot be circumvented by preference learning techniques [20], because even the most sophisticated of them can only capture $\mathbf{w}$ approximately. For instance, a common preference learning approach relies on pairwise comparisons [27, 39]. That is, the user is presented with pairs of options, and every pairwise choice she makes translates to a linear constraint on $\mathbf{w}$ (i.e., a halfspace in the preference domain $\mathcal{W}$). Combined, these constraints define a convex preference region R that serves as an approximation for $\mathbf{w}$. Yet, standard top- k processing approaches do not apply to region preference input.

The above limitations motivate the *constrained-preference top- k* (CT) query. Given a dataset D of the options (e.g., hotels) available in the market and a set C of linear constraints on $\mathbf{w}$, the CT query reports the union of the top- k sets for any $\mathbf{w}$ that abides by the constraints in C . The applicability of CT is not limited to once-off processing; it could also enhance the pairwise comparison model itself (or any other methodology that learns $\mathbf{w}$ by the incremental tightening/constraining of its permissible placements, such as [35, 45]). In particular, while the series of pairwise comparisons is still in progress (and the formation of C is ongoing), periodic executions of CT for the current set C could inform the preference learning algorithm of how many possible result records remain at play, and therefore help determine whether the approximation of $\mathbf{w}$ is tight enough. For example, the termination condition of the learning algorithm in [39] currently relies on the size (i.e., diagonal) of the residual region R , which is only an indirect measure of the true selectivity rendered by C in distinguishing among the records in D . That said, given the interactive nature of the pairwise comparisons scheme and the frequency of CT executions required, the efficient processing of CT would be essential.

Consider Figure 1, where each hotel's attributes are its average guest ratings on *Service*, *Cleanliness*, and *Location*, on a scale of 0 to 10. On the right, we illustrate the *preference domain*, i.e., the domain of the preference vector¹. The axes correspond to w_1 and w_2 , i.e., to the weights for *Service* and *Cleanliness*, on a scale of 0 to 1. The CT input includes value k and the constraint set C . The latter determines a convex polygon R in the preference domain, representing the approximated or expanded user preferences. Assume that R is the axis-parallel rectangle $[0.05, 0.45] \times [0.05, 0.25]$ (note that in the figure, the preference domain is truncated for clarity). The CT query for $k = 2$ reports all the hotels that may belong to the top-2 set if $\mathbf{w}$ is free to lie anywhere in R . Importantly, the result must be *minimal*, i.e., for every hotel p in the CT output, there should be at least one $\mathbf{w} \in R$ for

Hotel	Svc.	Cln.	Loc.
p_1	8.3	9.1	7.2
p_2	2.4	9.6	8.6
p_3	5.4	1.6	4.1
p_4	2.6	6.9	9.4
p_5	7.3	3.1	2.4
p_6	7.9	6.4	6.6
p_7	8.6	7.1	4.3

(a) Data records

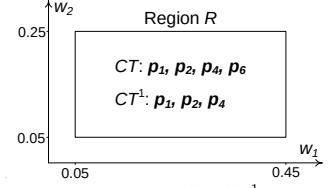
(b) Pref. domain and CT/CT^1 result

Fig. 1 Example of CT (for $k = 2$) and CT^1 on test data

which p is in the top-2 set. On our test data, the output is $\{p_1, p_2, p_4, p_6\}$.

Alongside the general CT query, we delve deeper into its special case for top-record sets (i.e., $k = 1$), denoted as CT^1 ; in the example of Figure 1, the output of CT^1 is $\{p_1, p_2, p_4\}$. Our dedicated investigation of this special case is motivated not only by its distinct practical significance, but also the technical potential it presents for superior algorithmic design. On the practicality front, in most decisions the user will eventually choose a single hotel, restaurant, service or product for her needs. If R indeed covers the user's latent preference vector, CT^1 will already include her actual most-preferred option in the output. Furthermore, regarding the aforementioned application of CT as a building block in pairwise comparison techniques, most of them (e.g., [35, 47, 51, 13]) focus explicitly on $k = 1$ (i.e., CT^1). On the technical front, our dedicated CT^1 solution is quintessentially different from the general CT algorithm, achieving significant performance improvements, but it also offers an important optimality guarantee that the general CT algorithm cannot deliver. Specifically, following the formal methodology of [23], we prove that the CT^1 algorithm accesses the minimum number of nodes in the index (e.g., R-tree) that organizes D . When dataset and index are kept in secondary (or remote) storage, this translates to optimal I/O cost. In a memory-based scenario (where performance is determined solely by computation time), said optimality effects computation savings by pruning (i.e., avoiding the consideration of) unnecessary index nodes and data records.

The current article extends the study in [32], where we defined the CT query and developed the general CT algorithm (presented in Section 4). In this extension, we investigate deeper the special CT case for top-record sets, i.e., CT^1 . Our specialized CT^1 algorithm is up to two orders of magnitude faster than the general CT approach and, at the same time, attains node-access optimality. The present article also includes an improved Introduction, an updated and better focused related work section, more formal preliminaries, the integration of new technical components which were missing in [32] (e.g., a practical approach for the derivation of R from C), and an extended experimental evaluation.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 defines the problem and pro-

¹ As we will explain in Section 3.1, for 3-dimensional data, the preference domain is practically 2-dimensional, because the third weight is dependent on the other two.

vides necessary background. Section 4 describes our algorithm for the general CT query, while Section 5 presents our specialized technique for CT^1 processing. Section 6 evaluates our proposal experimentally, and Section 7 concludes the paper.

2 Related Work

The querying of a large spatial (or low-dimensional) dataset D is typically facilitated by a general-purpose index, most commonly an R-tree [22]. When data and index reside on disk, an important performance consideration is the I/O cost, rendering *node-access optimality* a desirable property in spatial algorithm design. This widely-adopted notion of optimality was formulated in [23]. On the premise that the R-tree is given and that nodes accessed from the disk can be kept in memory until query processing terminates, the optimality analysis entails two steps. The first is to determine the query’s *search region*, i.e., the minimal area in the dataspace that any algorithm must look in in order to ensure that no result records have been missed. The second step is to prove that the algorithm at hand accesses only the index nodes whose minimum bounding box (MBB) intersects with the search region. For the nearest neighbor query, the search region is the hypersphere with center at the query point and radius equal to the distance between the query point and its nearest neighbor in D . The algorithm in [23] is proven to access only index nodes whose MBBs intersect with said hypersphere, and is thus node-access optimal.

In contexts where a user needs to choose among numerous alternatives (records) in a dataset D and her decision involves multiple aspects (dimensions), database research has explored two broad approaches to assist in her decision. The first considers that a record *dominates* another if it is more preferable in all dimensions, and presents the user with the *skyline* of D , i.e., those records that are not dominated by any other in D [8]. There are several skyline computation algorithms, reviewed in [16]. Among them, the (node-access optimal) *BBS* in [37] is the most established when D is indexed with an R-tree. Following the *branch-and-bound* paradigm [29], *BBS* utilizes a min-heap to consider index nodes and data records in increasing order of distance to the *top corner* of the dataspace (i.e., the point with maximum possible values in each dimension), while maintaining an interim skyline set S . Due to said order, every encountered record that is not dominated by any element already in S is directly added to it. Nodes and records found to be dominated by any record in S are pruned (i.e., eliminated). The skyline S is finalized when there are no unpruned nodes/records left to consider. Instead of distance to top corner, *BBS* can use any monotone metric to determine the node/record processing order; correctness is ensured as long as the metric guarantees that any record p' en-

countered after a record p cannot dominate p . *BBS* extends to k -skyband computation, i.e., to the skyline generalization that contains those records that are dominated by fewer than k others. The k -skyband is proven to be a superset of all records that could appear in the top- k set for any preference vector with non-negative weights.

The second approach assumes a linear utility function, determined by the user’s preference vector w . A top- k query reports the records in D that achieve the k highest scores, i.e., the records that have the k largest inner products with w . While a survey of top- k algorithms is offered in [25], of most relevance is the (node-access optimal) *BRS* [43] which is the state of the art when D is indexed with an R-tree. It follows the same lines as *BBS*, but it sorts records and index nodes according to their score/the score of their MBB’s top corner (which serves as an upper bound for the score of any record lying in the MBB), respectively.

The *convex hull* construct is related to the linear utility model. The convex hull is (the boundary of) the smallest convex polytope that encloses all records in a dataset [7]. The top record for any preference vector w is guaranteed to be on the convex hull. The *onion* technique [10] utilizes this property in a pre-computation approach to facilitate top- k processing. The first layer of the “onion” comprises the records that belong to the convex hull. The second layer comprises those that belong to the convex hull when the first layer is ignored. Generally, the i -th layer comprises the convex hull records when the first $i - 1$ layers are ignored. With k layers pre-computed, *onion* answers top- k queries by looking only across the records in these k layers. We stress that k -skyband and *onion* do not compute minimal subsets, i.e., they retain some records that cannot be in the top- k set for any $w \in \mathcal{W}$.

The (monochromatic) *reverse top-k* query [44] determines all regions in the preference domain where a record p ranks among the top- k . [44] provides an algorithm specifically for 2-dimensional data, while [42] proposes the $kSPR$ methodology to answer that query for $d \geq 2$. Each competitor of p is mapped into a halfspace in the preference domain, where it scores higher than p . In the resulting halfspace arrangement, those cells that are inside fewer than k halfspaces comprise the $kSPR$ output. Theoretically, to address CT , one could compute the k -skyband or the k *onion* layers of D as a filtering step, and then apply $kSPR$ (constrained to consider only region R) for each retained candidate p . If $kSPR$ reports an empty set, p is disqualified. Otherwise, it is part of the CT result. For the purposes of CT , we may terminate $kSPR$ as soon as the first sub-region of R is found where p is in the top- k set. Experiments in the preliminary version of this article [32] have demonstrated that either flavor of this baseline approach is orders of magnitude slower than our CT methodology. For completeness, we include some indicative results in Section 6.2.

The method in [17,18] considers weighted L_p norms, i.e., a family of scoring functions that is a superset of linear ones. Given a set of linear conditions on the preference weights, it reports the union of possible top records for any permissible weight combination. It employs a flexible notion of dominance, complemented with linear programming for top-record verification. That study is focused on unindexed datasets and considers every data record at least once, which is unsuitable for large D , especially if stored on the disk (the entire dataset would have to be read).

The technique in [31] receives as input the dataset D , a preference vector $\mathbf{w}$, and two positive integers, k and m . It expands $\mathbf{w}$ into a preference hypersphere with incrementally growing radius until the union of top- k sets for any preference in the hypersphere includes m distinct records. This hyperspherical region does not fit our problem definition or intended applications (recall that in CT the constraint set C defines a convex polytope R in the preference domain). Also, the main technical challenge in [31] is to determine the minimum radius required to produce an output of size m . This challenge (and the cost of the incremental expansion it necessitates) does not exist in CT .

The *regret minimization set* (RMS) query [36] selects a set S of m representatives from D so that, for any $\mathbf{w} \in \mathcal{W}$, the top record in S scores as close as possible to the top (or, in other definitions, the top- k -th [15]) record in the entire D . RMS aims to satisfy as it best can any possible user with just m representatives from D , without an intent for personalized answers. Utility loss is referred to as *regret* and its minimization is attempted via approximation (or heuristic) algorithms. RMS research is comprehensively surveyed in [48]. In a similar vein, assuming that the distribution of preferences in $\mathcal{W}$ is known, [38] (and [46]) select a set S of m representatives from D aiming to maximize the probability that S includes the top record (one of the top k records, respectively) for a random $\mathbf{w}$. Both [38] and [46] rely on sampling and provide approximate solutions.

Starting without any knowledge about the user's preference vector $\mathbf{w}$, *interactive RMS* methods go through several rounds of interaction with her [35,47,45]. In each round, they present a number of records and ask the user to choose the best, thus learning her preferences increasingly better. When the utility loss (compared to the top record for her latent $\mathbf{w}$) is guaranteed to be below a threshold, the record chosen in the last round becomes the recommendation for this user. Unlike our problem, these methods start without any preference input and focus on selecting the right records in each round, in order to tighten as much as possible an upper bound on utility loss. On the other hand, they could benefit from CT^1 (i.e., the $k = 1$ case of our problem in particular), because they intend to pick records from the convex hull of D , but resort to first computing the skyline before they can select such records. In their multiple rounds, they need to

consider different parts of the convex hull, and would therefore rely on multiple (and efficient) CT^1 calls.

3 Preliminaries

3.1 Problem Definition

Let D be the set of all available options (records). Each record $\mathbf{p}$ is represented by values $\mathbf{p}[i], i \in [1, d]$. A record may be treated as a point or a vector (depending on context) in a dataspace of dimensions $x_i, i \in [1, d]$. For simplicity, we assume that larger values are preferable in all dimensions, but handling is symmetric when the opposite holds for some/all dimensions.

A preference vector $\mathbf{w}$ comprises weights $\mathbf{w}[i], i \in [1, d]$ that represent the significance of the respective decision aspects (dimensions). In the linear utility model [28, 10], $\mathbf{w}$ implicitly associates each $\mathbf{p} \in D$ with a utility score equal to their inner product, i.e., $\text{Score}_{\mathbf{w}}(\mathbf{p}) = \mathbf{p} \cdot \mathbf{w}$. We assume preference vectors with non-negative weights, normalized such that $\sum_{i=1}^d \mathbf{w}[i] = 1$. The normalization does not limit in any way the permissible preferences [24,30], but reduces the preference domain $\mathcal{W}$ to dimensions $w_i, i \in [1, d-1]$. Henceforth, by $\mathbf{w}$ we refer to the $(d-1)$ -dimensional form of the preference vector. A record $\mathbf{p} \in D$ belongs to the top- k set for a preference vector $\mathbf{w}$ if there are fewer than k records $\mathbf{p}' \in D$ with $\text{Score}_{\mathbf{w}}(\mathbf{p}') > \text{Score}_{\mathbf{w}}(\mathbf{p})$. We denote the top- k set for $\mathbf{w}$ as $\mathcal{T}_{\mathbf{w}}$. In the special case of $k = 1$, we refer to it as the top-record set and denote it as $\mathcal{T}_{\mathbf{w}}^1$. The top- k set (for any k) could include more than k records in case of score ties, but we will explicitly discuss this scenario only when it has implications for the algorithmic design/description.

In our context, user preferences are specified in terms of a set of linear constraints C on $\mathbf{w}$. This is aligned with the application environment in Introduction, e.g., the common approach of extracting user preferences via record comparisons (e.g., [39,35,47]), where each pairwise choice by the user translates to a linear constraint on $\mathbf{w}$. The intersection of the linear constraints in C is a convex polytope in $\mathcal{W}$ [7], denoted as R . Although we will refer to both C and R in our description, region R (i.e., its defining vertices) need not be explicitly input. Appendix A describes a practical approach to derive R from C . The general CT query is defined as follows:

Definition 1 (CT) Given a dataset D , a positive integer k , and a preference region R , the CT query reports the set of exactly those data records that may belong to $\mathcal{T}_{\mathbf{w}}$ when the preference vector $\mathbf{w}$ lies in R , i.e., CT reports $\bigcup_{\mathbf{w} \in R} \mathcal{T}_{\mathbf{w}}$.

“Exactly” in Definition 1 means that the reported set is *minimal*, i.e., for every output record $\mathbf{p}$, there is at least one preference vector $\mathbf{w} \in R$ for which $\mathbf{p}$ belongs to the top- k

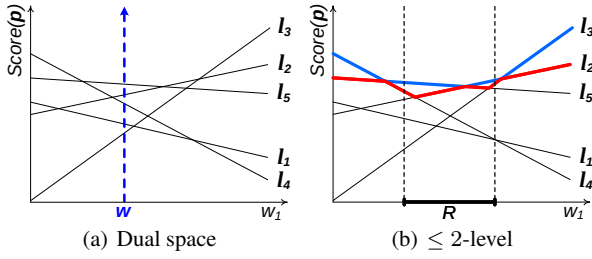


Fig. 2 Dual space and $\leq k$ -level for $d = 2$

set. Of distinct practicality and specialized algorithmic design is the $k = 1$ case of our primary query, denoted as CT^1 and defined, for completeness, as follows:

Definition 2 (CT^1) Given a dataset D and a preference region R , the CT^1 query reports the set of exactly those data records that may belong to the top-record set, $\mathcal{T}_w^1$, when the preference vector w lies in R , i.e., CT^1 reports $\bigcup_{w \in R} \mathcal{T}_w^1$.

Aiming at datasets of large scale, we assume that D is organized by a general-purpose index, such as an R-tree [22]. Our objective is to utilize the index to process the aforementioned queries efficiently in both storage scenarios, i.e., when the data/index reside in main memory or in secondary storage. Regarding dimensionality, the distinguishing power of score-based ranking diminishes with d [33, 49]. Specifically, as d grows, the scores of all records quickly converge, which renders rank-aware processing (including the standard top- k query) meaningless for more than a handful of dimensions. Therefore, we focus on low-dimensional settings (e.g., $d \leq 8$).

3.2 Example and Relationship to $\leq k$ -level

Consider a 2-dimensional D . As discussed above, the preference domain is 1-dimensional, since $w_2 = 1 - w_1$. If we plot the score of a record $p_i \in D$ as a function of w_1 , it is mapped into a line l_i . Figure 2(a) demonstrates that dual representation for $n = 5$ records. Our (1-dimensional) preference vector w corresponds to a position on the horizontal axis. If we shoot a vertical ray from that position upwards, the ray meets the lines l_i in ascending order of $\text{Score}_w(p_i)$. I.e., the top- k set for w comprises those records that correspond to the k lines that are met last by the ray. In Figure 2(a), w corresponds to the dashed ray. The ray meets lines l_2 and l_5 last, i.e., the top-2 set for w includes p_2 and p_5 .

Effectively, the dual representation maps the dataset into an arrangement of lines. Identifying all possible top- k sets (for any w in the preference domain) is equivalent to computing all points that lie on one of the lines and have fewer than k other lines above them. E.g., in Figure 2(b), the blue piecewise linear curve indicates the record that ranks 1-st for

any possible w (i.e., for any position of w on the horizontal axis), while the red curve captures the record that ranks 2-nd. The blue curve is called the 1-level of the arrangement, the red is called the 2-level, while together they form the ≤ 2 -level [5]. In general, the $\leq k$ -level captures the top- k set for any w . Therefore, our general problem can be seen as a constrained $\leq k$ -level problem, where w is bounded in a specific region R of the preference domain. In Figure 2(b), if R is the shown interval, the CT result for $k = 2$ includes those records that correspond to the part of the ≤ 2 -level that is between the dashed lines, i.e., $\{p_2, p_3, p_4, p_5\}$. The same parallel holds for any dimensionality, with records mapped into hyperplanes. A vector w still produces a ray, shot upwards from point $(w_1, w_2, \dots, w_{d-1}, 0)$.

Although conceptually interesting, the relationship to $\leq k$ -level is not useful in terms of algorithmic design. As elaborated in [4], efficiently computing the $\leq k$ -level has not been resolved completely. Specifically, the best known algorithm has a time complexity of $O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$ and is purely of theoretical interest [34]. In the context of large datasets, the only $\leq k$ -level solutions [14, 12] are explicitly for $d = 2$, which is a degenerate case, because the preference domain becomes 1-dimensional, and the problem can be addressed by an adaptation of the plane sweep algorithm [7].

3.3 Notation

Before presenting our solutions for CT and CT^1 , we summarize notation in Table 1.

Symbol	Description
D	Dataset
d	Data dimensionality
p	Data record with attributes $p[i], i \in [1, d]$
w	Preference vector with weights $w[i], i \in [1, d]$
$\mathcal{W}$	Preference domain
$\text{Score}_w(p)$	Utility score of p under w
C	Set of linear constraints on w
R	Preference region defined by C
k	Number of records required in the top- k set
$\mathcal{T}_w / \mathcal{T}_w^1$	Top- k /top-record set for w
H_i	Halfspace $\text{Score}_w(p_i) > \text{Score}_w(p)$ in pref. domain
G	r-dominance graph
ρ	A partition of R

Table 1 Summary of notation

4 Algorithm for CT Processing

In this section, we present the *Algorithm for CT (ACT)*. A key notion in *ACT* is *r-dominance*. The intuition is that, in traditional dominance, the dominator is preferable

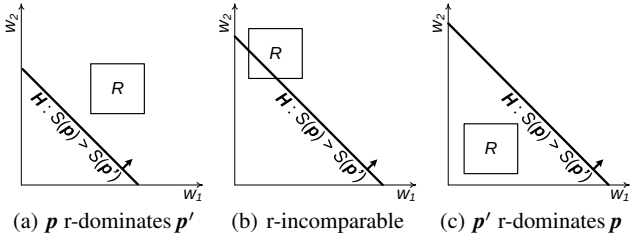


Fig. 3 r-dominance cases for records p and p'

to the dominee for any preference vector $w \in \mathcal{W}$. Instead, r-dominance is specific to preference vectors in R . Although a record may not dominate another in the traditional sense, it might always score higher when w is bounded in R .

Definition 3 (r-dominance) Given a region R in the preference domain, we say that record p r-dominates another record p' when $\text{Score}_w(p) > \text{Score}_w(p')$ for any preference vector w in R .

Consider two records, p and p' , which are *incomparable* in terms of traditional dominance, i.e., none dominates the other. The equality $\text{Score}_w(p) = \text{Score}_w(p')$ corresponds to a hyperplane in the preference domain, while the inequality $\text{Score}_w(p) > \text{Score}_w(p')$ corresponds to a halfspace. We distinguish 3 cases regarding the positioning of that halfspace w.r.t. R , shown in Figure 3. In the first case (Figure 3(a)), p r-dominates p' because R is fully covered by halfspace $H : \text{Score}_w(p) > \text{Score}_w(p')$, i.e., p scores higher for any $w \in R$. In the second case (Figure 3(b)), the hyperplane $\text{Score}_w(p) = \text{Score}_w(p')$ intersects with R ; in this case, none of the records r-dominates the other, and we therefore call them *r-incomparable*. The third case (Figure 3(c)) is symmetric to the first, with p being r-dominated by p' , i.e., scoring lower than p' anywhere in R .

Observe that in the above investigation we considered two incomparable records, i.e., records for which traditional dominance could not make safe conclusions about which ranks higher. In the first and in the third case, however, the notion of r-dominance allowed such conclusions to be safely made. Unlike traditional dominance, it is not straightforward to determine whether a record p r-dominates another record p' . We perform this test as follows. The answer is positive (i.e., p r-dominates p') if and only if all the vertices that define R are inside the halfspace $H : \text{Score}_w(p) > \text{Score}_w(p')$. That is equivalent to checking whether $\text{Score}_v(p) > \text{Score}_v(p')$ for every defining vertex v of R .

4.1 Filtering Step (r-Skyband)

The r-dominance concept is fundamental to several aspects of *ACT*. The first is that, from Definition 3, we infer that

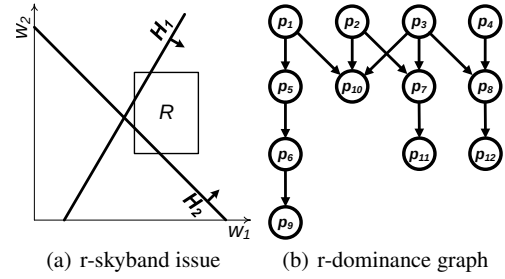


Fig. 4 r-dominance, r-skyband and r-dominance graph

any record that is r-dominated by k or more other records cannot be in the *CT* result, and can thus be eliminated from consideration. Formally, the only records that could be in the *CT* result must belong to the *r-skyband*:

Definition 4 (r-skyband) The r-skyband of a dataset includes only those records that are r-dominated by fewer than k others.

The r-skyband is a subset of the traditional k -skyband. To see this, recall that a record may r-dominate another even if they are incomparable in the traditional sense. Another important fact is that the r-skyband is still a superset of the *CT* result. The reason is that it relies on pairwise r-dominance, and is unable to capture the combined effect of multiple competitors on the ability of a specific record to enter the top- k set. To exemplify, consider record p and two of its competitors, p_1 and p_2 , in a *CT* scenario where $k = 1$. Assume that halfspaces H_1 and H_2 in Figure 4(a) correspond to $\text{Score}_w(p_1) > \text{Score}_w(p)$ and $\text{Score}_w(p_2) > \text{Score}_w(p)$, respectively. None of the halfspaces covers the entire R , thus p is not r-dominated by p_1 or p_2 . However, there is no part of R that is not covered by any of the two halfspaces. In other words, although p is a member of the r-skyband, it does not belong to the *CT* result. Thus, the r-skyband can only be used as a filtering mechanism for *CT* processing.

The computation of r-skyband (using the index on D) is similar to k -skyband computation, and follows the *BBS* paradigm [37], with a few important differences. First, instead of traditional dominance, we use r-dominance; the r-dominance test described previously applies both to record-to-record and to record-to-MBB r-dominance testing (by representing the MBB by its top corner, as per normal). Second, we use a different visiting order for R-tree nodes and records, i.e., a different sorting key for the search heap. We exploit the fact that w is bounded in R to determine a more effective order, i.e., to expedite the search by guiding it first to the most likely members of the r-skyband.

Specifically, our adapted *BBS* uses a max-heap, where the sorting key is the score (of records or of the top corner of node MBBs) according to the *pivot* vector of R . The pivot's value in each dimension is the average value of R 's vertices

in that dimension. Due to the convexity of R , the pivot is guaranteed to lie in R [7]. Other than that, the *BBS* process is the same as described in Section 2, by simply replacing traditional dominance with r-dominance tests.

Recall that *BBS* can use any monotone metric as key for its search heap (instead of distance to the top corner of the dataspace), as long as the metric guarantees that any record p' popped from the heap (i.e., considered for inclusion into the k -skyband) after a record p cannot dominate p . That requirement is upheld in our adaptation. In particular, the following holds:

Lemma 1 *Let p be a record just popped from the search heap of our adapted *BBS* algorithm. No record that is subsequently popped from the heap could r-dominate p .*

Proof. In our *BBS* adaptation, heap entries are popped in decreasing score order according to the pivot. Let p' be a record popped from the search heap after p . That is, p' does not score higher than p for the pivot. Because the pivot is a vector inside R , by Definition 3 it is impossible for p' to r-dominate p . $\square$

Practically, the lemma ensures that the interim r-skyband of our adapted *BBS* is append-only, i.e., if at the time a record p is popped it is r-dominated by fewer than k records in the interim r-skyband, p is certain to belong to the final r-skyband.

In addition to filtering (by means of the r-skyband), r-dominance is also useful in the refinement step, as we will see in Section 4.2. Thus, we record all pairwise r-dominance relationships between members of the r-skyband. The data-structure to store these relationships is a directed acyclic graph (DAG), called the *r-dominance graph* G . Using a DAG to maintain dominance relationships is common in the skyline literature (e.g., [52]). Figure 4(b) illustrates G in an example where $k = 4$. An arc from node (record) p to p' encodes the fact that p r-dominates p' . Note that an arc from p_1 to p_6 or to p_9 is not needed, because this is already implied by the transitivity of the r-dominance relationship.

Building G does not require any r-dominance tests, other than those already performed for r-skyband computation. In particular, prior to inserting any new record p into the r-skyband, *BBS* anyway needs to identify all existing r-skyband members that r-dominate it. Also, due to pivot-based sorting, these are the only records that could possibly r-dominate p . Thus, at the time of inclusion to the r-skyband, we know already all the records that r-dominate p . Their number is called the *r-dominance count* of p .

4.2 Refinement Step

As elaborated in Section 4.1, the r-skyband is a superset of the *CT* output. In this section, we describe the refinement

process to retain only those r-skyband members that belong to the *CT* output. Refinement considers candidates one by one, in decreasing order of their r-dominance counts. The reason behind this order is that if a candidate (node of G) is confirmed to be part of the *CT* result, so do all its ancestors in G , thus reducing the number of candidates to be verified.

The verification of individual candidates may further benefit from the r-dominance relationships stored in G . Consider Figure 4(b), where $k = 4$. The first candidates to verify are $p_9, p_{10}, p_{11}, p_{12}$, with r-dominance count 3. Ties are resolved arbitrarily. Assume that p_{11} is chosen to be verified first. Any of its ancestors in G is known to r-dominate it. Thus, p_{11} 's verification may ignore p_2, p_3, p_7 , and simply reduce its rank quota to $k - 3 = 1$; that is, with 3 ancestors ignored, it is as if p_{11} is to be verified for a *CT* query with parameter k reduced by 3. All remaining competitors are r-incomparable to p_{11} .

With ancestors ignored, the verification of a candidate p entails partitioning R by halfspaces $H_i : \text{Score}_w(p_i) > \text{Score}_w(p)$, each corresponding to one of the (remaining) competitors p_i . Formally, the supporting hyperplanes of these halfspaces define an arrangement bounded by R . Every cell (i.e., partition) in that arrangement lies inside a set of halfspaces. The competitors that correspond to these halfspaces are exactly those that rank higher than p if w falls in that partition. If any of the partitions in R is inside fewer halfspaces than the candidate's rank quota, then the candidate (and, consequently, each of its ancestors in G) is verified. Otherwise, it is not part of the *CT* result.

The problem with this approach is that computing the arrangement of all halfspaces H_i in R is a costly process (requiring $O(n^d)$ time [5], where n is the number of competitors). Instead, we initialize an empty arrangement in R , and insert into it halfspaces for just a small, carefully selected subset of competitors, hoping to (safely) verify or disqualify the candidate p without having to consider them all. The competitors chosen are those with the smallest r-dominance count. The rationale is that these competitors are intuitively the strongest, and in turn, the most likely to help disqualify the candidate. Note that the r-dominance counts here should ignore the candidate's ancestors (because, as stated before, they are disregarded during the verification of p).

In our running example on the verification of p_{11} , the competitors with the smallest r-dominance count (of 0) are p_1 and p_4 . Their respective halfspaces, H_1 and H_4 , are inserted into the arrangement. Assume that Figure 5(a) shows the resulting arrangement. The count in each partition indicates how many of the inserted halfspaces cover that partition. Since all partitions in R meet or exceed p_{11} 's rank quota (i.e., 1), the candidate is disqualified without a need to consider any other competitor. Observe that we consider only the arrangement within R . The rest of the preference domain is irrelevant to *CT* processing. Note also that when

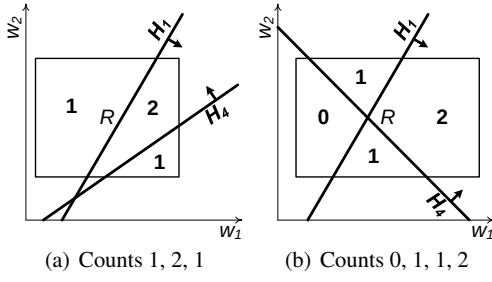


Fig. 5 Arrangement and partitions in R

a candidate is disqualified, it is directly removed from G in order to avoid waste of computations when other candidates are considered; the actual CT records are enough to disqualify any non- CT candidate.

Figure 5(b) illustrates an alternative scenario where the arrangement produced by H_1 and H_4 includes a *promising partition*, i.e., a partition with count smaller than the rank quota of the candidate. The promising partition in our case has count 0 (recall that p_{11} 's rank quota are 1). This partition could render p_{11} a member of the CT result. The question is whether any of the remaining competitors could produce a halfspace that covers or overlaps with the promising partition. Lemma 2 is instrumental to infer the answer.

Lemma 2 Let p_i be a competitor of the candidate p , whose halfspace H_i has already been inserted into the arrangement. Also, let p_j be a record that is r -dominated by p_i , and ρ be a partition in the arrangement that is outside H_i . The candidate p is guaranteed to score higher than p_j for any preference vector in ρ , i.e., $\text{Score}_w(p_j) < \text{Score}_w(p)$ for every $w \in \rho$.

Proof. First, from the definition of r -dominance, anywhere inside R it holds that $\text{Score}_w(p_j) < \text{Score}_w(p_i)$. Second, by the definition of halfspace H_i , anywhere outside H_i it holds that $\text{Score}_w(p_i) \leq \text{Score}_w(p)$. Combining the two facts, we deduce that $\text{Score}_w(p_j) < \text{Score}_w(p)$ for every $w \in R$ that is outside H_i , and thus for every preference vector in ρ . $\square$

Formally, for the candidate p at hand, we say that the count of a partition ρ is *confirmed* if every remaining competitor p_j (i.e., every competitor whose corresponding halfspace H_j has not been considered) is r -dominated by a record p_i whose halfspace H_i has already been inserted into the arrangement and ρ is outside H_i .

Returning to our example, the answer to whether the halfspace of any of the remaining competitors could cover or overlap with the promising partition in Figure 5(b) is negative. The reason is that the promising partition lies outside both H_1 and H_4 , while all remaining competitors are r -dominated by p_1 or p_4 . By Lemma 2, the count of the promising partition is confirmed, i.e., we are certain that the

count in that partition could not increase further. Hence, we are able to verify p_{11} (and all its ancestors in G) as part of the CT result.

In both previous examples, no additional competitors (i.e., other than p_1 and p_4) needed to be examined to decide on the candidate. To generalize, take Figure 5(a) as an example, but assume that the rank quota of the candidate p_{11} are 2. In this case, there are two promising partitions, both with count 1. Consider, for instance, the bottom-right promising partition. Although it lies outside H_4 (and thus all competitors r -dominated by p_4 cannot increase the partition's count), it is inside H_1 . I.e., Lemma 2 cannot disregard the competitors that are r -dominated by p_1 . To put this concretely, the examination of a promising partition needs to consider those of the remaining competitors that are not disregarded by Lemma 2. In our example, these are p_5, p_6, p_9, p_{10} .

Observe that the examination of the promising partition is effectively the original problem we started with, where: (i) instead of R , the region of interest is now the promising partition at hand; (ii) the rank quota of the candidate are reduced by the count of the partition (e.g., in the revised example in Figure 5(a), where we started off with rank quota of 2 for the candidate p_{11} , they are now reduced by the count of the promising partition (i.e., 1) to $2-1=1$); (iii) the competitors to consider are those that are not ancestors of the candidate and are not disregarded by Lemma 2. Therefore, we recursively apply the same process as we did for R .

That is, among the remaining competitors (i.e., p_5, p_6, p_9, p_{10}), we choose those with the smallest r -dominance count, and insert their halfspaces into a newly initialized, local arrangement, specific to the promising partition. Note that the r -dominance count now ignores any competitor that is either disregarded or has already been considered. For instance, this means that the r -dominance of p_{10} by p_2, p_3 (disregarded) and by p_1 (already considered in the original arrangement in R) should be ignored, rendering an r -dominance count of 0 for p_{10} . The r -dominance counts of p_5, p_6, p_9 are 0, 1, 2, respectively. Among the 4 competitors, p_{10} and p_5 (with r -dominance count 0) are inserted into a local arrangement for the bottom-right promising partition in Figure 5(a).

Figures 6(a) and 6(b) provide two alternative scenarios after the insertion of H_{10} and H_5 into the local arrangement. In Figure 6(a), there is a promising partition, which is outside both H_{10} and H_5 (recall that the rank quota of p_{11} were reduced to 1 in the gray partition). From Lemma 2 we infer that no remaining competitor could increase its count, i.e., its count of 0 is confirmed. We therefore verify p_{11} as a member of the CT result, and terminate the verification process for it.

On the contrary, in the scenario of Figure 6(b), there is no promising partition, so there is no further recursion within the current local arrangement, i.e., search within the

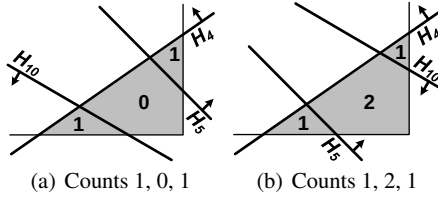


Fig. 6 Local arrangement in promising partition

gray triangular region is unfruitful. This however does not necessarily mean that the candidate p_{11} is disqualified, because the recursion in the other promising partition (i.e., the top-left partition in Figure 5(a)) could possibly discover a sub-partition where p_{11} does not exceed its rank quota and qualifies for the CT result.

An optimization here is that when there are multiple promising partitions, we apply the recursive process to them in decreasing order of their counts, so that the most promising partitions are considered first.

4.3 Drill Optimization

The verification process could be enhanced by a simple, yet effective optimization, called *drill*. The optimization applies to the verification of an individual candidate p . When region R is examined (in the initial call of the recursive verification process described previously), or when a promising partition is examined (in any of the following calls of the recursive process), before we consider the halfspace arrangement at all, we perform a regular top- k query for a *drill vector*, i.e., a vector inside R or inside the partition, respectively. If the candidate p is in the top- k set, it is directly verified as a member of the CT result.

The drill vector should be chosen such that (i) it falls within the region/partition at hand, and (ii) the candidate p is likely to rank high for it. An intuitive strategy to achieve this twofold objective is to choose the vector that maximizes the score of p (i.e., a linear expression), subject to the (also linear) constraints imposed by the (convex) region or partition. This can be done using linear programming in $O(d^{(1/2+o(1))}n)$ time, where n is the number of constraints that define the region/partition [9].

Importantly, when we make a drill, i.e., execute a top- k query for the drill vector, we do not refer at all to the dataset D or its index. Instead, our top- k search runs purely on graph G , in order to utilize directly the r-dominance information it holds. The search follows the branch-and-bound paradigm [29]. We say that we *evaluate* a node of G when we compute its score (according to the drill vector). We first evaluate the nodes with r-dominance count 0, and push them into a max-heap that uses score as the key. Then, we iteratively pop nodes from the heap. If the score of the popped

node p qualifies it for inclusion in the interim top- k set among the nodes we have popped so far, we update the interim top- k set, evaluate p 's children in G , and push them into the max-heap². Otherwise, we disregard p . The interim top- k set is finalized when the k -th smallest score in it is greater than the score of the last node popped from the heap (or the heap becomes empty).

4.4 Pseudo-code and Correctness

We present the pseudo-code for ACT in Algorithm 1, which invokes the recursive *Verify* procedure (Algorithm 2), i.e., the verification process for a given candidate p in a specified region/partition ρ . The last input of *Verify*, i.e., ignore set I , contains records to be ignored by the verification process. These records are also ignored in r-dominance counting in Line 3.

Algorithm 1 $ACT(\text{dataset } D, \text{region } R, \text{value } k)$

```

1:  $CT$  result set  $\mathcal{T} \leftarrow \emptyset$ 
2: Compute the r-skyband of  $D$ 
3:  $G \leftarrow$  build r-dominance graph
4: for each unverified  $p \in G$  in descending order of r-dominance
   count do
5:    $I \leftarrow$  ancestors of  $p$  in  $G$ 
6:   if  $Verify(p, R, k - |I|, I) == \text{True}$  then
7:     Insert into  $\mathcal{T}$  candidate  $p$  and all its ancestors in  $G$ 
8:   else
9:     Remove  $p$  from  $G$ 
10: Return  $\mathcal{T}$ 

```

Algorithm 2 $Verify(p, \rho, \text{rank quota } r, \text{ignore set } I)$

```

1: Perform drill for  $p$ ; if successful, return True
2:  $\mathcal{A} \leftarrow$  empty arrangement in  $\rho$ 
3: Insert into  $\mathcal{A}$  a halfspace for each competitor of  $p$  with r-
   dominance count = 0
4: for each promising partition  $\rho_i$  in  $\mathcal{A}$  do
5:   if Lemma 2 confirms the count of  $\rho_i$  then
6:     Return True
7:   else
8:      $S \leftarrow$  competitors with halfspaces already inserted in  $\mathcal{A}$ 
9:      $I' \leftarrow$  competitors that cannot affect  $\rho_i$  by Lemma 2
10:     $r' \leftarrow$  count of  $\rho_i$  in  $\mathcal{A}$ 
11:     $Verify(p, \rho_i, r - r', I \cup S \cup I')$ 
12: Return False

```

Lemma 3 proves ACT 's correctness.

² Since nodes in G may have multiple parents (like p_{10} in Figure 4(b)), some children of p may already be in the heap via another parent. Only children that are not already in the heap are evaluated/pushed into it.

Lemma 3 *ACT reports the correct result, i.e., exactly those records that may appear in the top- k set when the preference vector lies inside R .*

Proof. Records r -dominated by k or more others cannot belong to the CT result, thus filtering via the r -skyband is safe. Refinement considers all candidates individually, except those that can be safely verified when one of their descendants in G is confirmed to be in the CT result (which is safe by Definition 3). For each considered candidate, ACT constructs an arrangement according to its r -incomparable competitors³. If any partition in that arrangement has count smaller than its rank quota, by the definition of halfspaces H_i , it means that fewer than k competitors score higher than the candidate in that partition, hence, it is correctly reported in the CT result. Unless such a partition is discovered, the candidate is disqualified, i.e., ACT reports a tight CT result (not a superset of it). Our arrangement computation (and thus the verification process) for a given candidate p is correct, because the only competitors it ignores are (i) p 's ancestors in G , which are already taken into account by appropriate reduction of its rank quota, (ii) those safely disregarded by Lemma 2, and (iii) those whose verification has previously failed. $\square$

4.5 Implementation: Arrangement Indexing

Our discussion so far used halfspace arrangements and operations on them as building blocks, abstracting their maintenance and management. For arrangement indexing/maintenance, we employ the *CellTree* approach [42]. It relies on an implicit representation of the partitions by indicating only the halfspaces that define them (rather than computing and storing their explicit geometric representation) and organizes the partitions using a binary tree. The leaves of the binary tree are divided when a new halfspace is inserted, increasing the tree height by 1. This approach supports the incremental insertion of halfspaces into the arrangement, as well as the counting and the identification of halfspaces that cover a partition.

A remark is that there are multiple arrangement indices built and disposed during ACT execution. Specifically, an index is built for every local arrangement considered, i.e., for every call of the recursive process *Verify* (Algorithm 2). That index is discarded when the recursion proceeds to the promising sub-partitions of the local arrangement (if any). This keeps the size of the individual indices small, which implies small computation cost for operations on the respective local arrangements (e.g., halfspace insertion, coverage counting, etc).

³ Local arrangements at the leaves of the recursion tree of *Verify* together form an implicit arrangement in the entire R .

5 Algorithm for CT^1 Processing

In this section, we present our specialized *Algorithm for CT^1* (ACT^1). The central axis we base our design on is to consider the minimum possible number of index nodes, i.e., to utilize the index to the fullest, in order to save computations (for index subtrees/data records that cannot affect the CT^1 result) and to avoid unneeded node accesses (which translates to I/O cost reduction in a disk -or remote- storage scenario). Indeed, ACT^1 is node-access optimal, as we prove in Section 5.2. To serve the optimality analysis, we apply extra formality in the presentation of ACT^1 .

Section 5.1 offers insight into the CT^1 problem, defines the search region of the CT^1 query and, accordingly, determines the smallest set of index nodes that any CT^1 algorithm must access. Section 5.2 presents the “vanilla” version of ACT^1 that attains node-access optimality, and Section 5.3 enhances it with important optimizations that lead to the complete ACT^1 method (summarized in Section 5.4).

5.1 Search Region of the CT^1 Query

An essential step in understanding the problem and, at the same time, a prerequisite for formally analyzing node-access optimality [23], is to define the *search region* of an ad-hoc CT^1 query. Denoted as SR_R , it is the minimal area of the dataspace that any algorithm has to look into in order to retrieve all the records in the query result. We present its definition, followed by elaboration.

Definition 5 (Search region of CT^1) Given a vector $w \in \mathcal{W}$, let

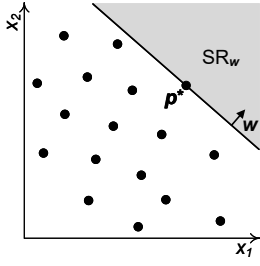
$$SR_w = \{p \in \mathbb{R}^d \mid p \cdot w \geq \text{Score}_w(p^*)\}$$

where $p^* \in \mathcal{T}_w^1$. The search region of the CT^1 query is

$$SR_R = \bigcup_{w \in R} SR_w$$

Recall that $\mathcal{T}_w^1$ is the top-record (i.e., $k = 1$) result for w . The search region SR_w of a top-record query has been analyzed in [43]. Specifically, for a given w , assume that p^* is the record with the highest score in D . In the $d = 2$ example of Figure 7, SR_w corresponds to the closed gray area defined by the line (hyperplane in $d > 2$) that is normal to vector w and passes by p^* . The search region of CT^1 is the union of the search regions of all possible top-record queries where $w \in R$.

The following lemma assumes an R-tree index on D , but extends directly to any tree index where (i) each data record is stored in a single node of the tree, and (ii) every node is associated with an axis-parallel bounding box. This class of indices is broad and covers the most common spatial access methods, such as the entire k -d tree [6] and Quadtree [40]

Fig. 7 SR_w example in 2-dimensional dataspace

families. Let $\mathcal{T} \subseteq D$ denote the CT^1 result, as stipulated in Definition 2.

Lemma 4 Any R-tree-based algorithm for CT^1 must access every index node whose minimum bounding box (MBB) intersects with SR_R .

Proof. Suppose that an R-tree-based algorithm terminates without accessing a node u whose MBB, denoted as M , intersects with SR_R . This means that $\exists w \in R$ such that SR_w intersects with M . In turn, this implies that it is possible for M to contain a record that belongs to $\mathcal{T}_w^1$ for the specific w (and thus to the CT^1 result, $\mathcal{T}$), which the algorithm will fail to report if it does not access u . $\square$

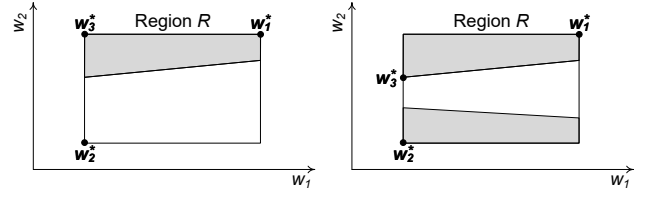
As we will prove shortly, ACT^1 visits only the R-tree nodes indicated in Lemma 4, and is thus node-access optimal.

5.2 Vanilla ACT^1

We first present the vanilla version of ACT^1 , which concretizes the main idea in our design. Assume that we compute for each record in D the maximum score it may attain under any $w \in R$, and sort all data records in descending order of said maximum score. Clearly, different records $p_i \in D$ may attain their maximum score under different preference vectors $w_i^* \in R$. However, we can safely tell that the first record in our order, say p_1 , belongs to the CT^1 result (i.e., $p_1 \in \mathcal{T}$) since it surely belongs to $\mathcal{T}_{w_1^*}^1$ and $w_1^* \in R$.

Consider the preference region R in Figure 8(a), and a dataset D comprising three records. Assume that preference vectors w_1^*, w_2^*, w_3^* is where records p_1, p_2, p_3 attain their respective maximum scores in R , and that these scores are $\text{Score}_{w_1^*}(p_1) = 4.5$, $\text{Score}_{w_2^*}(p_2) = 2.5$, and $\text{Score}_{w_3^*}(p_3) = 3$. We can safely tell that p_1 belongs to the CT^1 result.

The question now becomes whether we may inductively apply the same principle to confirm the next result record. The answer is affirmative, albeit the confirmed existence of p_1 in $\mathcal{T}$ invalidates the current order of remaining records. To see why, note that p_1 will be top-record in a non-empty



(a) p_1 confirmed as result record (b) p_2 confirmed as result record

Fig. 8 Main idea in ACT^1

part of R . Referring to the example in Figure 8(a), assume that this part of R is the gray region. Given that w_3^* falls in the gray region, reporting p_3 as the next result record based on the current $\text{Score}_{w_3^*}(p_3) = 3$ would be incorrect.

For that reason, the maximum attainable scores for the remaining records have to be recomputed by excluding that part from R (by the way, since a part of R is excluded, the maximum attainable scores can only decrease), and the records be sorted again. Assuming that the new w_3^* is as shown in Figure 8(b) and $\text{Score}_{w_3^*}(p_3) = 2$, record p_3 is now ordered after p_2 , whose maximum attainable score remains at $\text{Score}_{w_2^*}(p_2) = 2.5$.

The first record in the new order (i.e., p_2) is sure to belong to $\mathcal{T}$, because it outscores every other record (including p_1) under the updated vector w_2^* . Figure 8(b) illustrates the part of R where p_2 is top-record as the lower gray region. Henceforth, both gray regions would have to be excluded when recomputing the maximum attainable scores for the remaining records. The process can go on until none of the remaining records may outscore any of the already reported $\mathcal{T}$ members for any $w \in R$.

The above example illustrates the high-level idea. However, we must stress that, unlike the example, ACT^1 does not explicitly compute the gray regions, thus avoiding the respective computational geometric operations. Instead, it employs linear programming in the context of a score-bound that circumvents the explicit derivations of those regions. Sections 5.2.1 and 5.2.2 elaborate on the bound and its use for query processing.

5.2.1 The MaxScore Bound

Every entry M in the R-tree of D corresponds either to an MBB (of a node in the immediately lower tree level) or to a record $p \in D$. If M is an MBB, its *top corner* is the MBB's corner with the maximum coordinates, and represents the most competitive record that could exist in M (whether existing or hypothetical). If M is a record, then its top corner is simply the record itself.

Consider an arbitrary convex polytope $R \in \mathcal{W}$, an arbitrary subset $A \subseteq D$, and an R-tree entry M . We first define M 's *feasible weight region* under R and A .

Definition 6 (Feasible weight region) Given an R-tree entry M whose top corner is $\mathbf{p}^c$, M 's feasible weight region under R and A is

$$\text{WR}_R(M, A) = \{\mathbf{w} \in R \mid \forall \mathbf{q} \in A : \text{Score}_{\mathbf{w}}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}}(\mathbf{q})\}.$$

In other words, $\text{WR}_R(M, A)$ is the part of R where $\mathbf{p}^c$ is not outscored by any member of A . The *MaxScore* of M under R and A can now be formulated as:

$$\begin{aligned} \text{MaxScore}_R(M, A) &= \\ &= \begin{cases} -\infty & \text{if } \text{WR}_R(M, A) = \emptyset \\ \max_{\mathbf{w} \in \text{WR}_R(M, A)} \text{Score}_{\mathbf{w}}(\mathbf{p}^c) & \text{otherwise} \end{cases} \end{aligned}$$

In plain words, $\text{MaxScore}_R(M, A)$ is the maximum score achievable by $\mathbf{p}^c$ in the part of R where $\mathbf{p}^c$ is not outscored by any member of A . It can be computed by solving the following linear programming (LP) problem:

$$\begin{aligned} &\text{maximize } \text{Score}_{\mathbf{w}}(\mathbf{p}^c) \\ &\text{such that: } \text{Score}_{\mathbf{w}}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}}(\mathbf{q}) \quad \forall \mathbf{q} \in A \\ &\quad \mathbf{w} \text{ satisfies every condition in } C \\ &\quad \mathbf{w}[i] \geq 0 \quad \forall i \in [1, d-1] \\ &\quad \mathbf{w}[1] + \mathbf{w}[2] + \dots + \mathbf{w}[d-1] \leq 1 \end{aligned}$$

Recall that $\mathcal{W}$ is effectively $(d-1)$ -dimensional, and hence we solve the LP problem for the truncated version of $\mathbf{w}$ that excludes the d -th weight. Note that the last two lines in the LP formulation correspond to the conditions that bound $\mathcal{W}$, and the last three to the comprehensive set of conditions that define R . Together, all the conditions of the LP problem define $\text{WR}_R(M, A)$, adhering exactly to Definition 6. If the LP problem is infeasible, it means that $\text{WR}_R(M, A) = \emptyset$ and, consequently, that M cannot contain any record of $\mathcal{T}$. If the problem is feasible, its solution delivers not only the value of $\text{MaxScore}_R(M, A)$, but also the preference vector $\mathbf{w}^* \in \text{WR}_R(M, A)$ where $\mathbf{p}^c$ attains that score value. Proposition 1 is centered on the $\text{WR}_R(M, A)$ and the $\text{MaxScore}_R(M, A)$ constructs and offers a basis for ACT^1 .

Proposition 1 *The statements below are true:*

1. Let M, M' be R-tree entries such that M covers M' . Then, $\text{MaxScore}_R(M, A) \geq \text{MaxScore}_R(M', A)$ for any $A \subseteq D$.
2. Suppose that $\mathbf{p} \in \mathcal{T}_{\mathbf{w}}^1$ for some $\mathbf{w} \in R$. Then, $\text{MaxScore}_R(\mathbf{p}, A) \geq \text{Score}_{\mathbf{w}}(\mathbf{p})$ for any $A \subseteq D$.
3. Let M be an arbitrary R-tree entry. Then, $\text{MaxScore}_R(M, A) \geq \text{MaxScore}_R(M, A')$ for any $A, A' \subseteq D$ where $A \subseteq A'$.
4. If $\mathbf{p}$ belongs to the CT^1 result (i.e., $\mathbf{p} \in \mathcal{T}$), then $\text{WR}_R(\mathbf{p}, A) \neq \emptyset$ for any $A \subseteq D$.
5. If M is an MBB that covers any $\mathbf{p} \in \mathcal{T}$, then $\text{WR}_R(M, A) \neq \emptyset$ for any $A \subseteq D$.

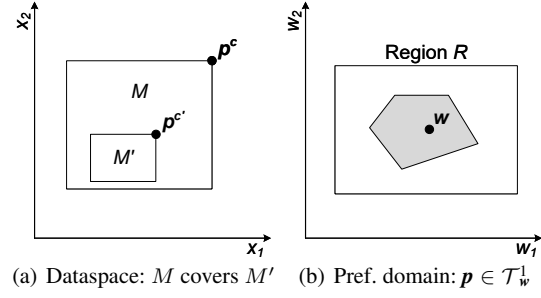


Fig. 9 Examples for Proposition 1

The formal proofs of the above statements are elementary/tedious and thus omitted. Instead, we provide the high-level rationale behind each of them. Regarding Statement (1), and with reference to Figure 9(a), if M covers M' , the top corner of M will either dominate or coincide with the top corner of M' , thus scoring no less for any $\mathbf{w} \in R$, hence, $\text{MaxScore}_R(M, A) \geq \text{MaxScore}_R(M', A)$. For Statement (2), consider Figure 9(b), where $\mathbf{p} \in \mathcal{T}_{\mathbf{w}}^1$, and the gray region is the part of R where $\mathbf{p}$ is top-record. Since that region is a superset of $\mathbf{w}$, naturally, $\text{MaxScore}_R(\mathbf{p}, A) \geq \text{Score}_{\mathbf{w}}(\mathbf{p})$. For Statement (3), if $A \subseteq A'$, the definition of the feasible weight region implies directly that $\text{WR}_R(M, A) \supseteq \text{WR}_R(M, A')$ and therefore $\text{MaxScore}_R(M, A) \geq \text{MaxScore}_R(M, A')$. For Statement (4), if $\mathbf{p} \in \mathcal{T}$, there is a preference vector $\mathbf{w} \in R$ such that $\forall \mathbf{q} \in D : \text{Score}_{\mathbf{w}}(\mathbf{p}) \geq \text{Score}_{\mathbf{w}}(\mathbf{q})$. Given that $A \subseteq D$, it follows that $\forall \mathbf{q} \in A : \text{Score}_{\mathbf{w}}(\mathbf{p}) \geq \text{Score}_{\mathbf{w}}(\mathbf{q})$ which, by the definition of the feasible weight region, implies that $\text{WR}_R(\mathbf{p}, A) \neq \emptyset$ (since it includes at least $\mathbf{w}$). Finally, for Statement (5), consider Figure 9(b), and assume that the gray region is the part of R where $\mathbf{p}$ is top-record. Since M covers $\mathbf{p}$, M 's top corner $\mathbf{p}^c$ either dominates or coincides with $\mathbf{p}$, and therefore $\text{WR}_R(M, A)$ is a superset of the gray region, i.e., non-empty.

5.2.2 Query Processing

Next, we explain how CT^1 can be answered (node-access optimally) by exploring the index of D using the branch-and-bound strategy guided by the *MaxScore* bound.

At all times, our algorithm maintains:

- a set of records $A \subseteq \mathcal{T}$ — we call A the *partial answer set*;
- a set H of R-tree entries (note: each entry is either a node's MBB or a data record).

Initially, $A = \emptyset$ and H includes all the entries of the R-tree's root. Throughout the algorithm, records can only be added to A but never deleted. At the end of the algorithm, the content of A will be exactly $\mathcal{T}$.

For every R-tree entry M in H , we maintain its precise $\text{MaxScore}_R(M, A)$ value at all times. That is, we compute $\text{MaxScore}_R(M, A)$ when M is first added to H , and recompute it whenever set A changes (i.e., whenever a new record is added to A). An entry M is evicted from H as soon as $\text{MaxScore}_R(M, A)$ drops to $-\infty$ or, equivalently, $\text{WR}_R(M, A) = \emptyset$.

The algorithm runs in iterations, each of which processes the entry $M \in H$ with the highest $\text{MaxScore}_R(M, A)$ (possible ties are resolved arbitrarily). To process M , the iteration first removes it from H . If M is a record $\mathbf{p}$, it is added to A . Otherwise, M is the MBB of a node, and the iteration inserts all the entries of that node into H . When H becomes empty, the algorithm returns A as the final answer. We refer to the above algorithm as vanilla ACT^1 .

5.2.3 Correctness

We prove the correctness of vanilla ACT^1 by first showing that it reports every $\mathbf{p} \in \mathcal{T}$ (in Lemma 5) and, next, that it reports only records $\mathbf{p} \in \mathcal{T}$ (in Lemma 6).

Lemma 5 *Consider any record $\mathbf{p} \in \mathcal{T}$. Let u be the leaf node where $\mathbf{p}$ is stored. At any moment during processing, one of the following must be true: (i) $\mathbf{p} \in A$, (ii) $\mathbf{p} \in H$, or (iii) H contains the MBB of an ancestor node of u (note: u is an ancestor of itself).*

Proof. The lemma holds at the beginning of the algorithm because it starts by adding all the root entries to H . In general, let M be the MBB of some ancestor node of u . Statement (5) of Proposition 1 assures us that $\text{WR}_R(M, A) \neq \emptyset$ for any $A \subseteq D$. This implies that M will eventually be processed by an iteration. When this happens, M is evicted from H , but then either $\mathbf{p}$ enters H , or the MBB of another ancestor node of u does. By Statement (4) of Proposition 1, $\text{WR}_R(\mathbf{p}, A) \neq \emptyset$ for any $A \subseteq D$. Therefore, the algorithm will eventually process $\mathbf{p}$ in an iteration, and $\mathbf{p}$ will be added to A when that happens. $\square$

The above lemma implies that the algorithm reports every record in $\mathcal{T}$. It remains to prove that no record $\mathbf{p}$ outside $\mathcal{T}$ can be reported.

Lemma 6 *Consider any record $\mathbf{p} \notin \mathcal{T}$. Record $\mathbf{p}$ will not be processed by any iteration of the algorithm.*

Proof. We will prove the lemma by contradiction. Assume that a record $\mathbf{p} \notin \mathcal{T}$ is processed by some iteration of the algorithm. Let A_{snap} be the content of the partial answer set A right before $\mathbf{p}$ is processed, and H_{snap} be the content of set H at that same time. For the iteration to process $\mathbf{p}$, it means that $\text{WR}_R(\mathbf{p}, A_{\text{snap}}) \neq \emptyset$. Let $\mathbf{w}$ be an arbitrary weight vector in $\text{WR}_R(\mathbf{p}, A_{\text{snap}})$.

Pick a record $\mathbf{p}^* \in \mathcal{T}_{\mathbf{w}}^1$. Since $\mathbf{p} \notin \mathcal{T}$, we have $\text{Score}_{\mathbf{w}}(\mathbf{p}^*) > \text{Score}_{\mathbf{w}}(\mathbf{p})$; note that the inequality is strict. This implies that $\mathbf{p}^* \notin A_{\text{snap}}$ (because otherwise, by Definition 6, $\mathbf{w}$ cannot have been in $\text{WR}_R(\mathbf{p}, A_{\text{snap}})$). By Lemma 5, H_{snap} must contain an entry M that is either $\mathbf{p}^*$ or an MBB covering $\mathbf{p}^*$. By Statements (1) and (2) of Proposition 1, we have $\text{MaxScore}_R(M, A_{\text{snap}}) \geq \text{MaxScore}_R(\mathbf{p}^*, A_{\text{snap}}) \geq \text{Score}_{\mathbf{w}}(\mathbf{p}^*) > \text{Score}_{\mathbf{w}}(\mathbf{p})$, but this contradicts the fact that the iteration chooses to process $\mathbf{p}$, rather than M . $\square$

Combined, Lemmata 5 and 6 guarantee the correctness of the algorithm, i.e., that the record-set it reports includes those and only those records required by Definition 2.

5.2.4 Node-access Optimality

A principle that has guided our design is node-access optimality, framed around the model established in [23] and reviewed in Section 2. In a disk-based scenario, that model's premise of not evicting accessed nodes (disk pages) during query processing fits well the present reality of large memory sizes, thus capturing accurately I/O performance in practice. Furthermore, as aforementioned, node-access optimality has a transferable effect to the memory-based scenario too, because (although that scenario entails no actual I/O operations) it guarantees that no computations will be wasted on subtrees of the index (and underlying records thereof) that cannot affect $\mathcal{T}$. Lemma 7 proves our algorithm's node-access optimality.

Lemma 7 *Vanilla ACT^1 accesses only the nodes whose MBBs intersect with $\text{SR}_{\mathbf{w}^*}$.*

Proof. Consider any node accessed by the algorithm. Let M be the MBB of the node, and $\mathbf{p}^c$ be the top corner of M . Denote by A_{snap} the content of the partial answer set A at the beginning of the iteration that processes M , and by H_{snap} the content of set H at that time. Let $\mathbf{w}^*$ be a weight vector in $\text{WR}_R(M, A_{\text{snap}})$ satisfying $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) = \text{MaxScore}_R(M, A_{\text{snap}})$. Denote by $\mathbf{p}^*$ a record in $\mathcal{T}_{\mathbf{w}^*}^1$.

We will show that $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}^*}(\mathbf{p}^*)$. This inequality indicates that $\mathbf{p}^c$ falls in $\text{SR}_{\mathbf{w}^*}$, proving that M has a non-empty intersection with $\text{SR}_{\mathbf{w}^*}$. Our proof of the inequality proceeds differently depending on whether $\mathbf{p}^* \in A_{\text{snap}}$.

Case 1: $\mathbf{p}^* \in A_{\text{snap}}$. By Definition 6, for every record $\mathbf{q} \in A_{\text{snap}}$ and for any vector $\mathbf{w} \in \text{WR}_R(M, A_{\text{snap}})$ it holds that $\text{Score}_{\mathbf{w}}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}}(\mathbf{q})$. Since $\mathbf{p}^* \in A_{\text{snap}}$ and $\mathbf{w}^* \in \text{WR}_R(M, A_{\text{snap}})$, it follows that $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}^*}(\mathbf{p}^*)$.

Case 2: $\mathbf{p}^* \notin A_{\text{snap}}$. As mentioned, $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) = \text{MaxScore}_R(M, A_{\text{snap}})$. By Lemma 5, H_{snap} must contain an entry M^* that is either $\mathbf{p}^*$ or an MBB

covering p^* . For M to be chosen for processing in the current iteration, it must hold that $\text{MaxScore}_R(M, A_{\text{snap}}) \geq \text{MaxScore}_R(M^*, A_{\text{snap}})$, and therefore $\text{Score}_w(p^c) \geq \text{MaxScore}_R(M^*, A_{\text{snap}}) \geq \text{MaxScore}_R(p^*, A_{\text{snap}})$. Now, from Statement (2) of Proposition 1, $\text{MaxScore}_R(p^*, A_{\text{snap}}) \geq \text{Score}_w(p^*)$. Combining the above, we derive $\text{Score}_w(p^c) \geq \text{Score}_w(p^*)$, which completes the proof. $\square$

5.3 Reducing MaxScore Computation Cost

The CPU time of vanilla ACT^1 is dominated by the MaxScore calculations. In this section, we will enhance the algorithm to drastically reduce the cost of MaxScore computations, without affecting its node accesses. To this end, we propose three strategies.

5.3.1 Reducing the Number of LP Conditions

The LP formulation in Section 5.2.1 to compute $\text{MaxScore}_R(M, A)$ includes (among others) one condition for every record q that is currently in the partial answer set A . Recall that the cost to solve an LP problem is linear to the number of constraints [9]. We will demonstrate that some of the records in A (and hence the respective conditions) may be inconsequential, i.e., could be omitted from the LP formulation without affecting the correctness of the MaxScore computation.

As an example consider Figure 10, where $d = 2$ and thus $\mathcal{W}$ has a single dimension, w_1 . Suppose that $R = [0.5, 1]$, and that for $w_1 = 0.5$ the top record position is shared between records $p_1 = (3, 2)$ and $p_2 = (2, 3)$ (with score 2.5), as indicated by the intersection of their score-lines at that position in the right subfigure. Hence, they both belong to $\mathcal{T}$, and therefore they will coexist in the partial answer set A at some point during execution. However, as demonstrated by their score-lines, for every other $w_1 \in R$, i.e., for every $w_1 \in (0.5, 1]$, record p_1 scores strictly higher than p_2 . This implies that, when we calculate $\text{MaxScore}_R(M, A)$ for some MBB with top corner p^c , the condition $\text{Score}_w(p^c) \geq \text{Score}_w(p_1)$ is stricter than (i.e., already implies) $\text{Score}_w(p^c) \geq \text{Score}_w(p_2)$, and thus the latter condition can be safely omitted from the LP formulation. A similar situation occurs at the other end of R , i.e., for $w_1 = 1$. The top-record set at that position corresponds to $p_1 = (3, 2)$ and $p_3 = (3, 1)$, with score 3. Hence, they both belong to $\mathcal{T}$. However, p_1 outscores p_3 at every other position in R , rendering the p_3 -related conditions redundant in any MaxScore computation. To formalize, we must first define a *weak* type of r-dominance.

Definition 7 (Weak r-dominance) Given a region R in the preference domain, we say that record p weakly

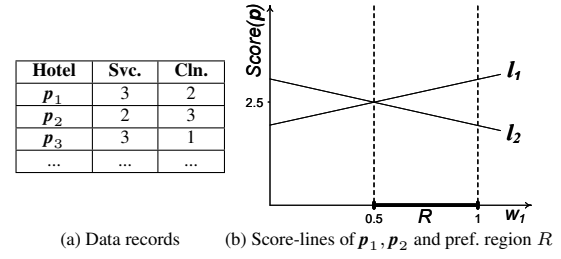


Fig. 10 Example of weak r-dominance

r-dominates (wr-dominates) another record p' when $\forall w \in R, \text{Score}_w(p) \geq \text{Score}_w(p')$, and $\exists w \in R, \text{Score}_w(p) > \text{Score}_w(p')$.

Observe that weak r-dominance deviates from regular r-dominance (i.e., Definition 3 in Section 4); it would be impossible for any member of A (or $\mathcal{T}$, for that matter) to r-dominate another member. In the example of Figure 10, p_1 does not r-dominate p_2 , but it does wr-dominate it.

Also, note that the two conditions in Definition 7 combined (i.e., $\forall w \in R, \text{Score}_w(p) \geq \text{Score}_w(p') \wedge \exists w \in R, \text{Score}_w(p) > \text{Score}_w(p')$) imply that it is impossible for any pair of records to wr-dominate each other. Furthermore, that wr-dominance is transitive, i.e., if p wr-dominates p' , and p' wr-dominates p'' , then p wr-dominates p'' .⁴

Following Definition 7, the *weak r-skyline* (wr-skyline) of A is the set of records $p \in A$ such that p is not wr-dominated by any record in A . We have:

Lemma 8 For any R -tree entry M and for any $A \subseteq D$, it holds that $\text{MaxScore}_R(M, A) = \text{MaxScore}_R(M, S)$, where S is the wr-skyline of A .

Proof. Let p^c be the top corner of M . From the definition of wr-skyline, for every record $p' \in A \setminus S$ there must be a record $p \in S$ such that p wr-dominates p' (otherwise, p' itself would have been in S , which contradicts the fact that $p' \in A \setminus S$). Hence, from Definition 7, it follows that $\text{Score}_w(p) \geq \text{Score}_w(p')$ for any $w \in R$. Therefore, in Definition 6 of the feasible weight region $\text{WR}_R(M, A)$, any vector $w \in R$ that satisfies condition $\text{Score}_w(p^c) \geq \text{Score}_w(p)$ also satisfies $\text{Score}_w(p^c) \geq \text{Score}_w(p')$, rendering the latter condition inconsequential to $\text{WR}_R(M, A)$. That is, $\text{WR}_R(M, A) = \text{WR}_R(M, S)$ and thus $\text{MaxScore}_R(M, A) = \text{MaxScore}_R(M, S)$. $\square$

We utilize Lemma 8 by maintaining the wr-skyline S of the partial answer set A , and always computing $\text{MaxScore}_R(M, S)$ instead of $\text{MaxScore}_R(M, A)$. Note

⁴ To see this, the wr-dominance between p, p' and p', p'' implies that for every $w \in R, \text{Score}_w(p) \geq \text{Score}_w(p') \geq \text{Score}_w(p'')$. Also, since p wr-dominates p' , there is $w \in R$ such that $\text{Score}_w(p) > \text{Score}_w(p')$, and since $\text{Score}_w(p') \geq \text{Score}_w(p'')$, we infer that $\text{Score}_w(p) > \text{Score}_w(p'')$. Combined, the two deductions imply that p wr-dominates p'' .

that this replacement of A by S is only for the purpose of MaxScore computations; every member of A (regardless of whether it belongs to S or not) is a confirmed member of $\mathcal{T}$ and will be reported as part of the query result.

We expect (and show experimentally in Section 6.3.2) that this approach is particularly effective when score ties occur frequently. Such cases are common when the per-aspect record ratings come from a limited discrete scale, e.g., when the different aspects are rated using integer star values from 1 to 5.

5.3.2 Reducing the Number of MaxScore Calls

Whenever a record is added to A , vanilla ACT^1 recomputes $\text{MaxScore}_R(M, A)$ for every R-tree entry M in set H . Firstly, as we have just established, we could safely perform the recomputation only if the new record also enters the wr-skyline S . However, we can reduce the number of recomputations even more drastically if we abolish the eagerness of vanilla ACT^1 with a lazy approach.

Specifically, when an R-tree entry M (be it an MBB or a data record) is first added to H , we compute its $\text{MaxScore}_R(M, S)$ according to the current set S and record the timestamp of the computation. We do not update $\text{MaxScore}_R(M, S)$, despite any possible additions to set S . We will only consider updating it when M is chosen for processing in a future iteration. When that happens, we check whether set S has changed since the last $\text{MaxScore}_R(M, S)$ computation. If not, we process M as per normal (without any MaxScore recomputation). If yes, we recompute $\text{MaxScore}_R(M, S)$ according to the current set S and (if $\text{MaxScore}_R(M, S) \neq -\infty$) add M back into H , with its updated MaxScore value and the present timestamp recorded as its computation time. Likewise, we will only consider recomputing $\text{MaxScore}_R(M, S)$ when M is chosen for processing in a future iteration, and so on.

Further improvements are possible. In addition to the timestamp, whenever $\text{MaxScore}_R(M, S)$ is computed, we also record the specific vector $\mathbf{w}^*$ reported in the solution of the respective LP problem, i.e., the vector $\mathbf{w}^* \in \text{WR}_R(M, S)$ that satisfies $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) = \text{MaxScore}_R(M, S)$, where $\mathbf{p}^c$ is the top corner of M . Assume that M is chosen for processing in a future iteration and that a (non-empty) set Δ_S of records have meanwhile entered S since M 's last MaxScore computation. Instead of recomputing the MaxScore directly, we first check whether the recorded $\mathbf{w}^*$ already satisfies $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}^*}(\mathbf{q})$ for every $\mathbf{q} \in \Delta_S$. If so, $\mathbf{w}^*$ is inside the current $\text{WR}_R(M, S)$ too, and therefore $\text{MaxScore}_R(M, S)$ is unaffected, i.e., the algorithm may go on to process M as per normal, without recomputing its MaxScore. Formally:

Lemma 9 *Let M be an R-tree entry in H , and $\mathbf{p}^c$ be M 's top corner. Also, let $\mathbf{w}^*$ be the preference vector reported*

in the LP solution when M 's MaxScore was last computed, and Δ_S be the set of records that have entered set S since that computation. If $\text{Score}_{\mathbf{w}^}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}^*}(\mathbf{q})$ for every $\mathbf{q} \in \Delta_S$, then $\text{MaxScore}_R(M, S)$ is unaffected by Δ_S , i.e., it remains unchanged since its last computation.*

Proof. Define by S_{snap} the content of set S when M 's MaxScore was last computed. That is, $\Delta_S = S \setminus S_{\text{snap}}$, where S is the current wr-skyline of A .

Since $S_{\text{snap}} \subseteq S$, Statement (3) of Proposition 1 assures us that:

$$\text{MaxScore}_R(M, S_{\text{snap}}) \geq \text{MaxScore}_R(M, S) \quad (1)$$

On the other hand, from (i) $\mathbf{w}^* \in \text{WR}_R(M, S_{\text{snap}})$ and (ii) $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) \geq \text{Score}_{\mathbf{w}^*}(\mathbf{q})$ for every $\mathbf{q} \in \Delta_S$, we can assert that $\mathbf{w}^* \in \text{WR}_R(M, S)$ and, hence, $\text{Score}_{\mathbf{w}^*}(\mathbf{p}^c) \leq \text{MaxScore}_R(M, S)$. Since $\text{MaxScore}_R(M, S_{\text{snap}}) = \text{Score}_{\mathbf{w}^*}(\mathbf{p}^c)$, we derive:

$$\text{MaxScore}_R(M, S_{\text{snap}}) \leq \text{MaxScore}_R(M, S) \quad (2)$$

Together, inequalities (1) and (2) imply that $\text{MaxScore}_R(M, S_{\text{snap}}) = \text{MaxScore}_R(M, S)$. $\square$

Another strategy to avoid MaxScore computations is to apply a fast filtering (pruning) of R-tree entries M that are guaranteed to have an empty $\text{WR}_R(M, S)$. Specifically, prior to any $\text{MaxScore}_R(M, S)$ computation (be it the first time M is considered for insertion to H or in subsequent iterations that require its up-to-date MaxScore), we check whether M 's top corner $\mathbf{p}^c$ is r-dominated (according to Definition 3 in Section 4) by any member of the current set S . If so, we may directly determine that $\text{WR}_R(M, S) = \emptyset \Rightarrow \text{MaxScore}_R(M, S) = -\infty$ and prune (i.e., disregard) M without any further consideration.

5.3.3 Jump-starting for Earlier Node Elimination

An indirect, yet effective strategy to reduce the number of required MaxScore computations is to prune early R-tree entries that cannot affect the query result. Observe that in vanilla ACT^1 , several non-leaf R-tree entries will have to be inserted into set H (and, possibly, even multiple times, due to the lazy MaxScore recomputation of Section 5.3.2) before the first few records are added to the partial answer set A and lend the MaxScore bound sufficient pruning power. Our strategy is to populate A (and therefore its wr-skyline S too) with strong records early, so that “weak” R-tree entries M can quickly receive a $\text{MaxScore}_R(M, S) = -\infty$ and be disregarded without any further investment of computations in them.

To achieve this, at the very beginning of ACT^1 , we compute the top-record set $\mathcal{T}_v^1$ for each defining vertex v of R , and add all retrieved records into set A (with a subsequent computation of its wr-skyline S) before the algorithm

continues as usual. Note that these top-record retrievals do not invalidate ACT^1 's node-access optimality, because (i) for every defining vertex v it holds that $v \in R$ and (ii) top-record retrieval for each of these vertices can be executed node-access optimally by BRS , as elaborated in Section 2.

5.4 Complete ACT^1

To put it all together, in this section we describe concretely the complete ACT^1 technique and provide its pseudo-code in Algorithm 3.

We maintain at all times:

- a partial answer set $A \subseteq \mathcal{T}$;
- the wr-skyline S of A ;
- a set H of R-tree entries.

Every entry $M \in H$ is associated with a real-valued key, denoted as $M.key$. We ensure the following invariant which enables, among others, the lazy MaxScore approach in Section 5.3.2:

$$M.key \geq \text{MaxScore}_R(M, S)$$

According to the optimization in Section 5.3.3, we initialize A as the union of the top-record results for the defining vertices of R , and accordingly initialize S to be its wr-skyline. We maintain set H in the form of a max-heap, organized according to the aforementioned $M.key$ values. For simplicity, in the following, symbol H refers to that max-heap.

The algorithm still runs in iterations, but now we maintain a *clock*, which is originally set to 0, and is increased by one (Line 10) each time a new iteration starts. The reader may conveniently interpret the clock as the sequence number of the current iteration. The clock plays a vital role in several aspects of our algorithmic design. In particular:

- For each entry $M \in H$, value $M.keytime$ records the clock value c when its $M.key$ was last computed. If A_{snap} is the content of A at the beginning of the c -th iteration, we enforce the invariant that $M.key = \text{MaxScore}_R(M, A_{\text{snap}})$. Note that if $M.key$ has never been computed, then $M.keytime = -\infty$.
- $S.time$ records the clock value c when the content of S was last updated.
- For each $p \in A$, we store in $p.time$ the clock value when p was added to A .

After the initialization, in Lines 5-8 we call function *getMaxScore* (explained later) for each entry M in the root of the R-tree. If the function returns *Feasible*, the respective entry M is associated with $M.key = \text{MaxScore}_R(M, S)$ and pushed into the max-heap H . If *getMaxScore* returns *Infeasible*, M is ignored, i.e., pruned. This completes iteration 0.

Each of the following iterations starts by popping an entry M from the max-heap H (Line 11). The iteration then checks whether $M.key$ is “fresh”, namely, whether it was computed using the most updated S . The check is performed in Line 12, and directs processing as follows:

Case 1: $M.key$ is not fresh [Lines 12-15]. If $M.keytime < S.time$, the key is not fresh, and is thus updated to $\text{MaxScore}_R(M, S)$ using function *getMaxScore* (Line 13). It is possible for $\text{MaxScore}_R(M, S)$ to be $-\infty$, in which case *getMaxScore* returns *Infeasible*; otherwise, the function returns *Feasible*. In the latter case, we push M back into H (internally, *getMaxScore* has meanwhile updated $M.key$ and $M.keytime$ to their current values). The iteration then finishes (Line 15).

Case 2: $M.key$ is fresh [Lines 16-24]. If the condition in Line 12 is not satisfied (i.e., if it holds that $M.keytime \geq S.time$), $M.key$ is fresh, prompting the execution to continue at Line 16. Recall that M can be either a data record $p \in D$ or the MBB of an index node. In the former case, we directly add p to the partial answer set A , set $p.time$ to the current clock value, and update the wr-skyline S accordingly (Line 17). If the addition of p to A has affected S (i.e., p is a member of the updated wr-skyline S), we also set $S.time$ to the current clock value. On the other hand, if M is the MBB of an R-tree node, we access that node (observe that in a disk-based scenario, this requires reading the respective page from the disk). For every entry M' in that node, we use *getMaxScore* to compute its $M'.key = \text{MaxScore}_R(M', S)$ for the first time (Lines 22-23). If the function returns “feasible”, M' is pushed into H .

When H becomes empty, the algorithm terminates with $\mathcal{T} = A$ as the output.

The *getMaxScore* Function: It remains to explain the details of *getMaxScore*. This function takes three parameters: an R-tree entry M , the wr-skyline S of the current partial answer set A , and the current clock value. Let p^c be the top corner of M . Its behavior can be summarized as follows:

- If $\text{MaxScore}_R(M, S) = -\infty$, it returns *Infeasible*.
- Otherwise, it sets $M.key$ to $\text{MaxScore}_R(M, S)$, sets $M.keytime$ to the current clock value, sets $M.w$ to a preference vector $w^* \in \text{WR}(M, S)$ satisfying $\text{Score}_{w^*}(p^c) = \text{MaxScore}_R(M, S)$, and returns *Feasible*. $M.w$ can be thought of as a “witness” vector proving that p^c may potentially attain the score of $M.key$.

The main principle behind this function's design is to avoid as much as possible the solving of an LP problem to derive $\text{MaxScore}_R(M, S)$. Algorithm 4 presents the function's pseudo-code. Lines 2-3 apply the last of the three techniques presented in Section 5.3.2, i.e., return *Infeasible* if p^c is r -dominated by any record in S . Line 4 checks whether the MaxScore of M has been previously computed and if so, Lines 5-8 implement the second technique in Section 5.3.2;

Algorithm 3 ACT^1 (dataset D , region R)

```

1:  $A \leftarrow A_{init}$ , and  $p.time \leftarrow 0$  for each  $p \in A$ 
2:  $S \leftarrow$  the wr-skyline of  $A$ , and  $S.time \leftarrow 0$ 
3:  $clock \leftarrow 0$ 
4: Initialize a max-heap  $H$ 
5: for each entry  $M$  in the root of the R-tree do
6:    $M.keytime \leftarrow -\infty$ 
7:   if  $getMaxScore(M, S, clock) == Feasible$  then
8:     Push  $M$  into  $H$ 
9: while  $H$  non-empty do
10:   $clock \leftarrow clock + 1$ 
11:   $M \leftarrow$  the entry popped from  $H$ 
12:  if  $M.keytime < S.time$  then
13:    if  $getMaxScore(M, S, clock) == Feasible$  then
14:      Push  $M$  into  $H$ 
15:    go to Line 9
16:  if  $M$  is a data record  $p \in D$  then
17:     $p.time \leftarrow clock$ , add  $p$  to  $A$ , and update  $S$ 
18:    if  $p \in S$  then
19:       $S.time \leftarrow clock$ 
20:  else  $\triangleright M$  is the MBB of an R-tree node
21:    for each entry  $M'$  in the R-tree node of  $M$  do
22:       $M'.keytime \leftarrow -\infty$ 
23:      if  $getMaxScore(M', S, clock) == Feasible$  then
24:        Push  $M'$  into  $H$ 
25: Return  $\mathcal{T} \leftarrow A$ 

```

Algorithm 4 $getMaxScore(M, S, clock)$

```

1:  $p^c \leftarrow$  top corner of  $M$ 
2: if  $p^c$  is r-dominated by any record in  $S$  then
3:   return Infeasible
4: if  $M.keytime \geq 0$  then
5:    $w^* \leftarrow M.w$ 
6:   if  $Score_{w^*}(p^c) \geq Score_{w^*}(q)$  for all  $q \in S$  with  $q.time > M.keytime$  then
7:      $M.keytime \leftarrow clock$ 
8:   return Feasible
9: Compute  $MaxScore_R(M, S) \triangleright$  LP problem using  $S$  instead of  $A$ 
10: if the LP problem in Line 9 is infeasible then
11:   return Infeasible
12:  $w^* \leftarrow$  the preference vector reported in the LP solution of Line 9
13:  $M.key \leftarrow Score_{w^*}(p^c)$ ,  $M.w \leftarrow w^*$ , and  $M.keytime \leftarrow clock$ 
14: return Feasible

```

i.e., they utilize the witness vector $M.w$ and Lemma 9 to avoid $MaxScore$ recomputation if $M.w$ is guaranteed to be in $WR_R(M, S)$ for the current set S , leaving $M.key$ as is, updating $M.keytime$ to the current clock value, and returning *Feasible*.

If after the aforementioned steps the function has not yet returned, Line 9 resorts to solving the LP problem to compute $MaxScore_R(M, S)$ (that is the LP problem defined in Section 5.2.1 using, however, S instead of A). If the LP problem is infeasible, the function returns the same (Lines 10-11). Otherwise (i.e., the LP problem is feasible) the function updates $M.key$, $M.w$ and $M.keytime$ accordingly, returning *Feasible* (Lines 12-14).

6 Experiments

We evaluate our techniques using synthetic and real datasets. The synthetic are standard benchmarks for preference queries [8], i.e., *Independent* (IND), *Correlated* (COR), and *Anticorrelated* (ANTI). The real ones are *HOTEL* (166,378 records, 4D) [1], *HOUSE* (315,265 records, 6D) [2], and *NBA* (21,960 records, 8D) [3]. All datasets are indexed by R-trees with 4Kb page size. The performance metrics assessed are computation time and node accesses. The former alone represents performance in a memory-based scenario, while the latter reflects disk (or remote) I/O cost when index and data are stored in secondary (or remote) storage. Every reported measurement is the average (unless otherwise specified) of 50 queries, for axis-parallel hyper-cubes R randomly generated in the preference domain. The side-length of R is expressed as a percentage σ of the axis length. All methods were implemented in C++. *SK* and *ON* denote the k -skyband and the *onion* variants of the baseline in Section 2. For their $kSPR$ component, we used the most efficient algorithm (*LP-CTA*) in [42]. All experiments run on a PC with Intel i7-4770 3.4GHZ CPU and 16GB RAM.

6.1 Case Studies and Qualitative Evaluation

We first conduct a case study on *CT* using real data, and then we present a qualitative comparison, in measurable terms, with traditional operators, like k -skyband, *onion*, and regular top- k queries.

In Figure 11, we use actual NBA statistics for the season 2016-2017. We use $d = 2$ data dimensions, namely, average *Rebounds* and *Points* for each player. The preference domain is 1-dimensional, and corresponds to the weight for *Rebounds*, denoted as w_r . We set $k = 3$ and $R = [0.64, 0.74]$. In the figure, we represent the dataspace, and show as red points the players output by *CT*, i.e., *Russell Westbrook*, *Anthony Davis*, *Hassan Whiteside*, and *Andre Drummond*. For comparison, we also illustrate the first $k = 3$ layers of *onion*, which include the 4 players in the *CT* result plus another 7, shown as solid squares. The k -skyband ($k = 3$), which is always a superset of the k *onion* layers, contains the *onion* players plus another 2 shown as crosses, i.e., 13 players in total.

In Figure 12, we look deeper into the relationship between *CT* and traditional operators. We use the complete *NBA* dataset, set σ to 10%, and vary k . In Figure 12(a), we compare the number of records in the *CT* result with that in the first k *onion* layers and in the k -skyband. *CT* reports 30 to 100 times fewer records than *onion* and k -skyband. The main reason is that these traditional operators are insensitive to the preferences of the user expressed via region R . Furthermore, even if the R constraint were dropped, *onion* and k -skyband would still not be minimal, i.e., they generally

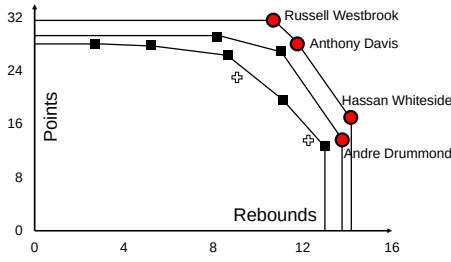


Fig. 11 Case study on NBA data for season 2016-2017

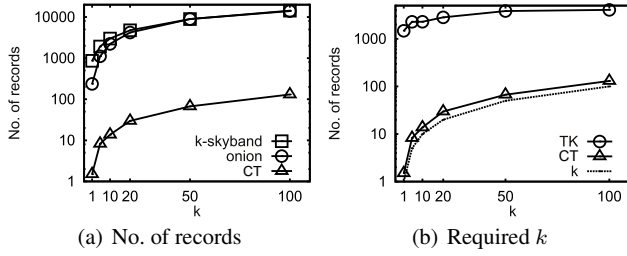


Fig. 12 Comparison with traditional operators (NBA)

contain some records that cannot belong to the top- k set for any preference vector [25].

Next, using the same data and experiment settings, we examine whether a regular top- k query could simulate CT , i.e., whether a top- k query for a slightly larger k can produce all CT records. In particular, for each tested value of k , we first processed CT , and then executed an incremental top- k query using as preference vector w the pivot of R (as the most representative vector therein). We invoked the top- k query iteratively until it output all records in the CT result, and we recorded how many records it had to output in total. We plot that number in Figure 12(b) (labeled as TK), together with the number of records in CT result, and the original value of k for reference. We observe that the top- k approach practically required to increase parameter k by 40 to 460 times, and output 30 to 230 times more records than CT in order to cover its result. These facts confirm that the regular top- k query cannot effectively simulate CT , and that simply increasing k (in a standard top- k query) is not equivalent to CT processing.

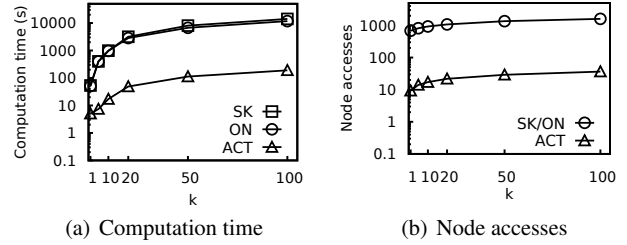
6.2 Performance Evaluation for CT

In this section, we focus on CT processing (for general k) and evaluate the performance of ACT . Table 2 presents the experiment parameters, along with tested and default values (in bold). In each experiment, we vary one parameter and keep the rest at their defaults.

In Figure 13, we use IND and study the effect of k on ACT and the two baselines, SK and ON . As k increases, more records and more possible top- k sets belong to the CT

Parameter	Tested values
Dataset cardinality n	100K, 200K, 400K , 800K, 1600K
Data dimensionality d	2, 3, 4 , 5, 6, 7
Value k	1, 5, 10 , 20, 50, 100
R 's side-length σ	0.1%, 0.5%, 1% , 5%, 10%

Table 2 Experiment parameters

Fig. 13 Effect of k (IND)

result. Therefore, both the computation time and the number of node accesses increase for all methods. Note that SK and ON perform the same node accesses (thus the common line in Figure 13(b)), since the first step in both of them is to compute the k -skyband, which is also the only step they need to access the index for. The main observations are:

- (i) ACT outperforms the baselines by one to two orders of magnitude in terms of computation time, and by two orders in terms of node accesses. The reason is twofold; the baselines consider more candidates than ACT , and they make numerous calls to their costly $kSPR$ building block.
- (ii) ON has slightly shorter computation time than SK , because of its tighter, convex-hull-based filtering.

Having established that the baselines are impractical, we henceforth exclude them from the experiments. Focusing on ACT , in Figure 14 we employ three different data distributions and vary the dataset cardinality n . As expected, performance is the best for COR data (because records that are good in one dimension, tend to be good in all dimensions, i.e., r -dominate almost the entire dataset), and the worst for ANTI (because records that are good in one dimension, are poor in the others, thus leading to less effective filtering, and more diversity in the possible top- k sets). The computation time and the node accesses of ACT increases sub-linearly to n , indicating good scalability. The sub-linear increase is in line with the fact that the cardinality of the k -skyband (and therefore of the r -skyband too) is sub-linear to n [21], linking directly to the number of candidates produced by ACT 's filtering step.

In Figure 15(a), we vary the data dimensionality d in IND. Computation time and node accesses are presented in the same plot, which shows, among others, a correlation between the two performance metrics. Due to the computational geometric nature of the CT problem, its hardness rises with d , which explains the increase in ACT 's costs.

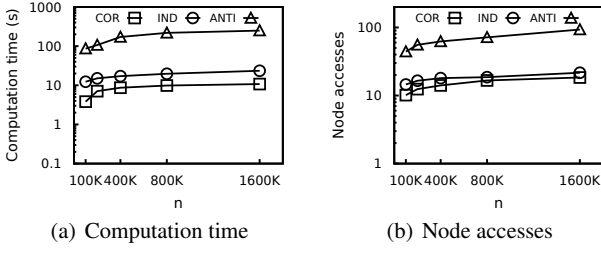


Fig. 14 Effect of n and data distribution on ACT

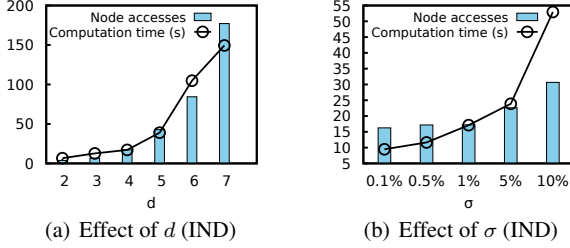


Fig. 15 Effect of data dimensionality d and of R size on ACT

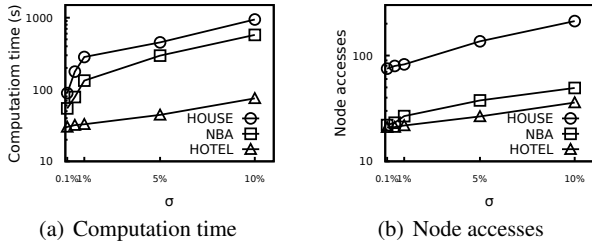


Fig. 16 Effect of R size on ACT (real datasets)

In Figure 15(b), we vary σ , which determines the size of region R . As expected, larger R implies larger output (i.e., more records in the CT result), and thus longer computation time and more node accesses.

Next, we experiment on real data. In Figure 16, we use *HOTEL*, *HOUSE*, and *NBA*, and we vary σ . The trends are similar to the synthetic data. Processing in *HOUSE* is costlier than *HOTEL*, because the former has more records and more attributes (6D versus 4D). Processing in *NBA* is costlier than *HOTEL* because, although smaller, it has twice as many attributes (8D versus 4D).

6.3 Performance Evaluation for CT^1

In this section, we focus on the CT^1 problem. First, in Section 6.3, we compare ACT (for $k = 1$) with our specialized ACT^1 approach. Next, in Section 6.3.2, we evaluate the impact of individual optimizations in ACT^1 through an ablation study. We use the same experimental process as Section 6.2, and the same tested/default values for parameters

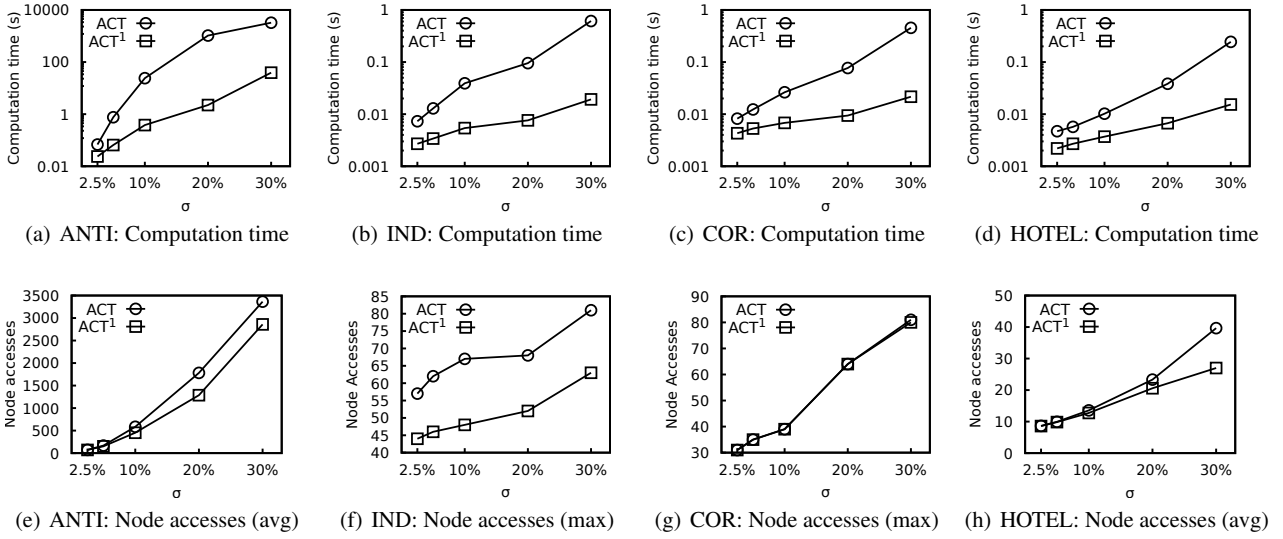
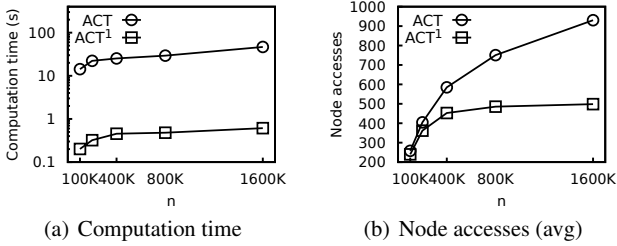
n, d as in Table 2. For $k = 1$, the σ values in Table 2 render trivial output sizes, hence, we use range 2.5%, 5%, 10%, 20%, 30%.

6.3.1 Comparison of ACT^1 with ACT

In Figure 17, we present the computation time (charts in the first row) and node accesses (second row) of ACT and ACT^1 as we vary the size of R via parameter σ . We use all synthetic distributions. We also use *HOTEL*, as a representative for the real datasets (*HOUSE* and *NBA* exhibit similar trends). ACT^1 consistently outperforms ACT in all settings. Regarding computation time: in *ANTI*, ACT^1 is one to two orders of magnitude faster; in *IND*, it is 2.7 to 32 times faster; in *COR*, it is 2 to 21 times faster; in *HOTEL*, it is 2 to 16 times faster. The reason for those major improvements is that (i) ACT^1 examines the minimum number of index and data nodes, as proven in Section 5.2.4, and (ii) ACT^1 does not rely on any computational geometric operations, which are typically expensive. On the latter point, observe that ACT performs halfspace operations and arrangement maintenance (see Section 4.5), which are avoided altogether by ACT^1 . An important remark is that as σ grows, so does the number of considered data records, which widens the performance gap between ACT and ACT^1 .

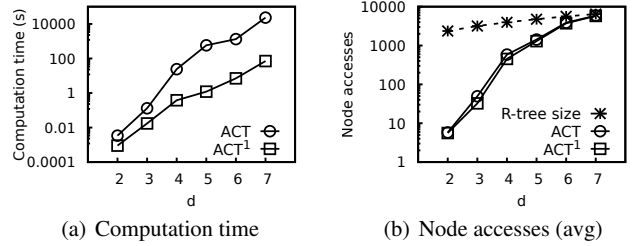
In terms of node accesses (second row in Figure 17), ACT^1 is optimal, and naturally performs the fewest possible. In *ANTI*, ACT accesses up to 40% more nodes than ACT^1 , and in *HOTEL*, up to 47%. For *IND* and *COR*, on the other hand, the accesses of ACT are close to ACT^1 . The reason is that in *IND* and *COR*, by design, the members of the r -skyline are universally strong in all dimensions [8], and are hence highly likely to belong to the CT^1 result too (implying few false positives in ACT 's filtering). For this reason, specifically in Figures 17(f) and 20(d), instead of the average, we present the maximum node accesses across the 50 queries we executed, for both methods.

In Figures 18 and Figure 19, we investigate the effect of n and d , respectively. We use *ANTI* data, because they represent a strong stress test, but also because they reveal an interesting insight. In Figure 18, both performance metrics (for both methods) increase sub-linearly with n , however, ACT^1 maintains a clear lead, requiring two orders of magnitude shorter computation time than ACT and performing a fraction of its node accesses (e.g., about half the accesses of ACT for $n = 1.6M$). Turning to Figure 19, the processing costs for both methods increase with d , for the same reason we explained in Section 6.2. That said, ACT^1 scales more gracefully than ACT . To understand why, the size of the skyline is known to rise steeply with d [8, 21] and, by extension, the same holds for the size of the r -skyline. In turn, this implies more false positives in ACT 's filtering step, which translates to increased waste of computations and more un-

Fig. 17 Effect of σ Fig. 18 Effect of n (ANTI)

necessary node accesses. Concordantly, the gap in computation time between ACT and ACT^1 widens steadily with d (see Figure 19(a)). Interestingly, however, the gap in node accesses originally widens but then it starts to narrow for the largest values of d (see Figure 19(b)). The latter is because in high dimensions, and especially in ANTI data, the dimensionality curse plagues the index's efficiency, causing even a node-access-optimal algorithm to read almost the entire index (thus limiting the effect of said optimality). To verify this, in Figure 19(b) we include the total number of nodes in the index, with the optimal number of node accesses (as represented by the line of ACT^1) converging to that total number for the highest d values.

For completeness, in Tables 3 and 4 we present the computation time, the average and the maximum node accesses of ACT and ACT^1 in the default setting for the NBA and HOUSE datasets. The results are consistent with the findings presented so far and verify the general superiority of ACT^1 over ACT .

Fig. 19 Effect of d (ANTI)

Method	Comp. Time	Node Acc. (avg)	Node Acc. (max)
ACT	0.3684s	87.56	133
ACT^1	0.0675s	82.44	119

Table 3 Performance in default setting for HOUSE

Method	Comp. Time	Node Acc. (avg)	Node Acc. (max)
ACT	1.3805s	37.26	47
ACT^1	0.0717s	31.68	42

Table 4 Performance in default setting for NBA

6.3.2 Effect of Individual Optimizations in ACT^1

Focusing on ACT^1 in particular, in this section we examine its specific optimizations. To assess their effectiveness, we individually disable each of them and compare with the complete ACT^1 .

First, in Figure 20 we consider the optimizations in Sections 5.3.2 and 5.3.3, both of which aim to reduce the number of LP problems to be solved. We use COR, ANTI, and HOTEL as representative datasets, and vary σ . The charts in the first row present the computation time and those in the second row the number of LP problems solved. Note that

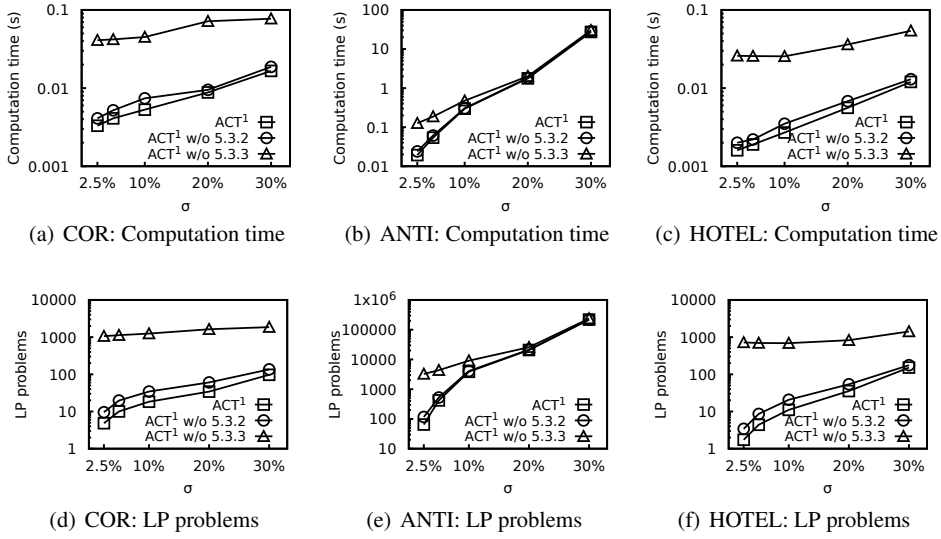


Fig. 20 Effectiveness of optimizations in Sections 5.3.2 and 5.3.3

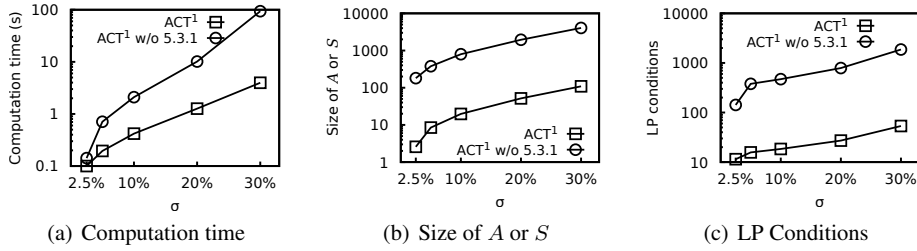


Fig. 21 Effectiveness of optimization in Section 5.3.1

we do not credit to Section 5.3.2 the use of a heap to organize set H , because not employing a heap would severely degrade performance, and would mask the effects of the (subtler) optimizations in that section, namely, the use of timestamps (to avoid computing $\text{MaxScore}_R(M, S)$ if the wr-skyline S has not been updated since entry M was last encountered) and the use of Lemma 9.

Comparing the full ACT^1 and its version where Section 5.3.2 is disabled, we observe that this optimization is responsible for a reduction of up to 49.4% in the number of LP problems solved, and a reduction of up to 28.4% in computation time. Comparing the full ACT^1 and its version without Section 5.3.3, we see that this enhancement reduces the number of LP problems up to 414 times, and the computation time up to 16 times. We infer that both sections contribute significantly to ACT^1 's efficiency, with Section 5.3.3 having the largest impact.

For both optimizations, their effect is most pronounced when σ is small. For Section 5.3.2, that is because a small R implies fewer (i.e., less frequent) changes in the wr-skyline S , and thus a more effective use of timestamps. For Section 5.3.3, a smaller R implies that jump-starting

fetches a larger proportion of the final CT^1 result, $\mathcal{T}$, which in turn leads to more effective pruning of R-tree nodes, and leaves fewer result records to retrieve in the subsequent steps of the algorithm. This is also the reason why both sections achieve greater improvements in COR and HOTEL than in ANTI. Specifically, in ANTI both the wr-skyline S and the final result $\mathcal{T}$ tend to be larger, which means more updates in S (thus, less effective use of timestamps) and a smaller proportion of $\mathcal{T}$ retrieved by jump-starting (thus, less effective node pruning and more result records remaining to retrieve). That said, even in ANTI, Section 5.3.2 achieves up to 20% reduction in computation time (and up to 44.5% fewer LP problems), and Section 5.3.3 achieves up to 6.5 times faster processing (and up to 50.2 times fewer LP problems).

Finally, we turn to the enhancement in Section 5.3.1, which aims to reduce the number of conditions in the solved LP problems. This technique is tailored to situations where score ties are frequent. That scenario is rare in our main datasets, but very common in contexts where the per-aspect ratings of products (records) take values from a small discrete domain. Examples include editor/critic reviews that rate each decision aspect in number of stars (i.e., 5 discrete

values) or using an integer from 1 to 10. To simulate such a scenario, we generated a dataset where record values in each dimension are randomly and uniformly drawn from the integer domain $[1, 10]$, i.e., a domain of 10 discrete values. We kept the default $n = 400K$ and $d = 4$, and imposed a cap of 8 to the sum of values a record could have, to avoid pathological instances where two or three records dominate every other record in the dataset (which probabilistically is very likely to happen when the attributes are defined over small discrete domains).

Using this dataset, in Figure 21 we compare the version of ACT^1 without the optimization in Section 5.3.1 against the complete ACT^1 as we vary σ . We present: the average computation time of 50 queries; the average final size of the partial answer set A (for the ACT^1 version without Section 5.3.1) or of the wr-skyline S (which takes the place of A in MaxScore computations, for the complete ACT^1); and the average number of conditions in the solved LP problems. We observe that when the respective algorithm terminates, S is up to 70 times smaller than A . This large size difference results in up to 35 times fewer conditions in the LP problems when the optimization is enabled and, overall, up to 23.6 times shorter computation time, confirming the effectiveness of the enhancement.

7 Conclusion

In this paper, we formalize a general approach to enhance standard top- k queries by taking into account inherent inaccuracies in weight specification. In particular, we study the CT query, which identifies the records that may belong to the top- k set when the preference vector lies in a given region. We propose algorithms for the general version of the problem and for a special case of distinct practical interest. Using real and benchmark datasets, we demonstrate the practicality and scalability of our techniques. A challenging direction for future work is to extend CT to uncertain data. Another promising direction is to introduce preference regions as input to other rank-aware queries.

Acknowledgments

We thank Yufei Tao for insightful discussions, which helped shape this work. This work was partially supported by the National Natural Science Foundation of China (Grant No. 62422206, 62532007, 62532001).

This version of the article has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: <http://dx.doi.org/10.1007/s00778-026-00972-w>. Use of this Accepted Version is sub-

ject to the publisher's Accepted Manuscript terms of use <https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms>.

References

1. Hotel dataset. <https://github.com/RyanBalfanz/hotelsbase>.
2. House dataset. <https://www.ipums.org>.
3. NBA dataset. <https://www.basketball-reference.com>.
4. P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order voronoi diagrams. *SIAM J. Comput.*, 27(3):654–667, 1998.
5. P. K. Agarwal and M. Sharir. Arrangements and their applications. *Handbook of computational geometry*, pages 49–119, 2000.
6. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
7. M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, 2008.
8. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
9. T. M. Chan. Improved deterministic algorithms for linear programming in low dimensions. In *SODA*, pages 1213–1219, 2016.
10. Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference*, pages 391–402, 2000.
11. B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10(4):377–409, 1993.
12. M. A. Cheema, Z. Shen, X. Lin, and W. Zhang. A unified framework for efficiently processing ranking related queries. In *EDBT*, pages 427–438, 2014.
13. Q. Chen and R. C. Wong. Robust best point selection under unreliable user feedback. *PVLDB*, 17(11):2681–2693, 2024.
14. S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top- k queries in two dimensions. In *DASFAA*, pages 201–208, 2012.
15. S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Computing k -regret minimizing sets. *PVLDB*, 7(5):389–400, 2014.
16. J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *SIGMOD Rec.*, 42(3):6–18, 2013.
17. P. Ciaccia and D. Martinenghi. Reconciling skyline and ranking queries. *PVLDB*, 10(11):1454–1465, 2017.
18. P. Ciaccia and D. Martinenghi. Flexible skylines: Dominance for arbitrary sets of monotone functions. *ACM Trans. Database Syst.*, 45(4):18:1–18:45, 2020.
19. H. Edelsbrunner. *Algorithms in combinatorial geometry*. Springer, 1987.
20. J. Fürnkranz and E. Hüllermeier, editors. *Preference Learning*. Springer, 2010.
21. P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004.
22. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.
23. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.
24. V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, pages 259–270, 2001.
25. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

26. D. R. Insua and S. French. A framework for sensitivity analysis in discrete multi-objective decision-making. *Eur. J. Oper. Res.*, 54(2):176–190, 1991.
27. K. G. Jamieson and R. D. Nowak. Active ranking using pairwise comparisons. In *NIPS*, pages 2240–2248, 2011.
28. R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Trade-Offs*. Wiley, 1976.
29. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
30. H. Li, T. N. Chan, M. L. Yiu, and N. Mamoulis. FEXIPRO: fast and exact inner product retrieval in recommender systems. In *SIGMOD Conference*, pages 835–850, 2017.
31. K. Mouratidis, K. Li, and B. Tang. Marrying top-k with skyline queries: Relaxing the preference input while producing output of controllable size. In *SIGMOD Conference*, pages 1317–1330, 2021.
32. K. Mouratidis and B. Tang. Exact processing of uncertain top-k queries in multi-criteria settings. *PVLDB*, 11(8):866–879, 2018.
33. K. Mouratidis, J. Zhang, and H. Pang. Maximum rank query. *PVLDB*, 8(12):1554–1565, 2015.
34. K. Mulmuley. On levels in arrangements and voronoi diagrams. *Discrete & Computational Geometry*, 6:307–338, 1991.
35. D. Nanongkai, A. Lall, A. D. Sarma, and K. Makino. Interactive regret minimization. In *SIGMOD Conference*, pages 109–120, 2012.
36. D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. J. Xu. Regret-minimizing representative databases. *PVLDB*, 3(1):1114–1124, 2010.
37. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.
38. P. Peng and R. C. Wong. k-hit query: Top-k query with probabilistic utility function. In *SIGMOD Conference*, pages 577–592, 2015.
39. L. Qian, J. Gao, and H. V. Jagadish. Learning user preferences by adaptive pairwise comparison. *PVLDB*, 8(11):1322–1333, 2015.
40. H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
41. R. Seidel. Convex hull computations. In *Handbook of Discrete and Computational Geometry, Second Edition*, pages 495–512. Chapman and Hall/CRC, 2004.
42. B. Tang, K. Mouratidis, and M. L. Yiu. Determining the impact regions of competing options in preference space. In *SIGMOD Conference*, pages 805–820, 2017.
43. Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.
44. A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørsvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1215–1229, 2011.
45. W. Wang, R. C. Wong, and M. Xie. Interactive search for one of the top-k. In *SIGMOD Conference*, pages 1920–1932, 2021.
46. X. Xiao and J. Li. rkhit: Representative query with uncertain preference. *Proc. ACM Manag. Data*, 1(2):126:1–126:26, 2023.
47. M. Xie, R. C. Wong, and A. Lall. Strongly truthful interactive regret minimization. In *SIGMOD Conference*, pages 281–298, 2019.
48. M. Xie, R. C. Wong, and A. Lall. An experimental survey of regret minimization query and variants: bridging the best worlds between top-k query and skyline query. *VLDB J.*, 29(1):147–175, 2020.
49. A. Yu, P. K. Agarwal, and J. Yang. Top-k preferences in high dimensions. *IEEE Trans. Knowl. Data Eng.*, 28(2):311–325, 2016.
50. J. Zhang, K. Mouratidis, and H. Pang. Global immutable region computation. In *SIGMOD Conference*, pages 1151–1162, 2014.
51. J. Zheng and C. Chen. Sorting-based interactive regret minimization. In *APWeb-WAIM*, pages 473–490, 2020.
52. L. Zou and L. Chen. Pareto-based dominant graph: An efficient indexing structure to answer top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(5):727–741, 2011.

A Derivation of Region R from Constraint Set C

There are several computational geometric approaches to derive the vertices that define R from the set C of linear constraints on $\mathbf{w}$ [11, 41]. That said, the following method works very efficiently in practice.

Firstly, the total set of conditions that define R comprises not only C but also the conditions that bound the preference domain (i.e., $\mathbf{w}[i] \geq 0, \forall i \in [1, d-1]$ and $\sum_{i=1}^{d-1} \mathbf{w}[i] \leq 1$). For each condition in this comprehensive set (e.g., $\mathbf{w}[1] \geq 0$), consider the respective equality (i.e., $\mathbf{w}[1] = 0$). Every defining vertex of R corresponds to the solution of a linear system of $d-1$ of these equalities [19]. We therefore consider every combination of $d-1$ conditions from the comprehensive set. If the respective linear system is feasible, and its solution satisfies every condition in the original comprehensive set, the said solution is a defining vertex of R . When each combination has been considered, we have derived all defining vertices of R .