Maximum Rank Query

Kyriakos Mouratidis   Jilian Zhang   HweeHwa Pang
School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
{kyriakos, jilian.z.2007, hhpang}@smu.edu.sg

ABSTRACT

The top-k query is a common means to shortlist a number of options from a set of alternatives, based on the user's preferences. Typically, these preferences are expressed as a vector of query weights, defined over the options' attributes. The query vector implicitly associates each alternative with a numeric score, and thus imposes a ranking among them. The top-k result includes the k options with the highest scores. In this context, we define the *maximum rank* query (*MaxRank*). Given a focal option in a set of alternatives, the *MaxRank* problem is to compute the highest rank this option may achieve under any possible user preference, and furthermore, to report all the regions in the query vector's domain where that rank is achieved. *MaxRank* finds application in market impact analysis, customer profiling, targeted advertising, etc. We propose a methodology for *MaxRank* processing and evaluate it with experiments on real and benchmark synthetic datasets.

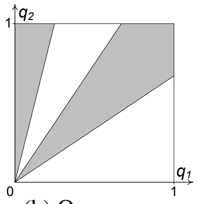
1. INTRODUCTION

A multitude of online portals allow users to browse through different *options* for a product or service they wish to buy, such as cars, phones, mobile plans, restaurants, apartments, etc. For example, TripAdvisor.com maintains ratings of different hotels in terms of location, service, sleep quality, etc. The top-k query is a common means to shortlist several options based on the user's preferences. Each option $\mathbf{r}$ has a number of *attributes*, e.g., the different aspects rated on TripAdvisor.com. In the most prevalent and intuitive top-k model, the user specifies a *query vector* $\mathbf{q}$ comprising a numeric weight q_i per attribute [11]; the score of each option is defined as the weighted sum of its attributes (equivalently, the dot product $\mathbf{r} \cdot \mathbf{q}$), which in turn imposes a ranking of the available options. The k highest ranking options form the top-k result.

In this work, we consider settings where users browse a pool of options via such top-k queries. We view the problem, however, from the perspective of the option providers. In particular, given a *focal option*, we compute the maximum rank it may achieve w.r.t. any possible query vector. Additionally, we report all the regions in the query vector's domain where that rank is attained. We call this the *maximum rank* query (*MaxRank*).

Rec.	d_1	d_2	$S_1(\mathbf{r})$	$S_2(\mathbf{r})$
$\mathbf{r}_1$	.8	.9	.83	.89
$\mathbf{r}_2$	.2	.7	.35	.65
$\mathbf{r}_3$	.9	.4	.75	.45
$\mathbf{r}_4$	.7	.2	.55	.25
$\mathbf{r}_5$	.4	.3	.37	.31
$\mathbf{p}$	.5	.5	.5	.5

(a) Records and scores



(b) Query space

Figure 1: *MaxRank* example for $d = 2$

Figure 1(a) shows a set of options (data records) in a 2-dimensional example. The options could correspond to hotels, and their attributes to hotel quality (d_1) and value-for-money (d_2). Note that we draw vectors/records in boldface to avoid confusion with scalars/attributes. The focal option is $\mathbf{p} = (0.5, 0.5)$. Given a query vector $\mathbf{q}$, we may sort the available options on descending score – the position of $\mathbf{p}$ in the sorted sequence is called the *order* of $\mathbf{p}$ w.r.t. $\mathbf{q}$. Figure 1(a) presents the scores $S_1(\mathbf{r})$ and $S_2(\mathbf{r})$ of the options w.r.t. query vectors $\mathbf{q}_1 = (0.7, 0.3)$ and $\mathbf{q}_2 = (0.1, 0.9)$, respectively. The order of $\mathbf{p}$ w.r.t. $\mathbf{q}_1$ is 4, whereas its order w.r.t. $\mathbf{q}_2$ is 3. Assuming that the query weights take values between 0 and 1, the user's query vector $\mathbf{q}$ may lie anywhere in the query space shown in Figure 1(b). The purpose of *MaxRank* is to report (i) the smallest order that $\mathbf{p}$ may achieve w.r.t. any possible query vector, and (ii) all the regions of the query space where $\mathbf{p}$ achieves that order. We denote the smallest order achievable by $\mathbf{p}$ as k^*. In our example, *MaxRank* would report that $k^* = 3$ and that it is attained when $\mathbf{q}$ lies anywhere within the shaded regions of the query space.

The *MaxRank* query offers a direct measure of visibility and market impact. In our running example, for instance, it is useful to the owner of hotel $\mathbf{p}$ to know what is the best her hotel may fare, i.e., how high it could rank, relative to competing options. Knowing k^* is also useful for a "what-if" investigation, in a scenario where the hotel is not yet priced/launched and the owner wants to examine a number of alternatives (in which case, an equal number of *MaxRank* queries is required). On the other hand, knowing the regions of the query space where $\mathbf{p}$ is the most appealing, reveals the preference profiles of its most likely customers. In the example of Figure 1(b), we infer that $\mathbf{p}$ is the most appealing either to customers that largely favour value-for-money over hotel quality (i.e., $q_2 \gg q_1$), or to those who weigh the two criteria roughly the same (i.e., $q_1 \approx q_2$). With this information, the option provider (hotel owner) may better cater to the needs of her customers, or target her marketing campaign at the right audience. Additionally, if the probability distribution of $\mathbf{q}$ in the query space is known, the *MaxRank* regions enable the computation of the probability that $\mathbf{p}$ achieves its smallest possible order k^*.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

In this paper, we introduce the *MaxRank* query and propose a framework for its processing. By exploiting the computational geometric nature of the problem, we make several crucial observations and develop scalable *MaxRank* algorithms for two or more dimensions. Moreover, we extend our methodology to a variant of the query, called *incremental MaxRank*. With experiments on real, as well as benchmark synthetic datasets, we demonstrate the practicality and efficiency of our techniques.

2. RELATED WORK

The top-k query retrieves from a dataset D the k records with the highest scores. The score is typically defined as a linear function over the data attributes. In that sense, the user's preference is represented by a vector of weights (query vector **q**), where the i-th weight corresponds to the i-th data attribute ($i \in [1, d]$). By treating a data record **r** as a vector, its score is equal to the dot product $\mathbf{r} \cdot \mathbf{q}$. A flurry of techniques have been proposed for top-k queries; Ilyas et al. [11] provide an extensive survey on the topic.

Soliman et al. [18] consider uncertain scoring functions and identify the most "representative" top-k result, under different definitions. First, they compute the most likely top-k result if the query vector is randomly chosen. Next, they compute the top-k result that is least dissimilar to all possible alternative results. Finally, they introduce sensitivity measures for a given top-k result.

Mouratidis and Pang [15] study *local immutable regions* (LIR). By isolating one weight q_i in the query vector **q**, and assuming that the rest are fixed, the LIR is the range of q_i values for which the top-k result is the same as for the original vector **q**. Zhang et al. [24] extend this concept to the *global immutable region* (GIR). The GIR is the maximal locus around **q** where the top-k result remains the same. The GIR is shown to be a convex polytope.

Das et al. [7] consider the evaluation of ad-hoc top-k queries over a data stream. The main idea is that only a small subset of the records could appear in the top-k result w.r.t. any query vector. To identify (and maintain) this small subset they rely on a geometric representation of the top-k query and a notion of duality, where records and queries are mapped into lines and rays, respectively. In Section 4, we use a similar (albeit different) mapping in a first-cut solution (called FCA) for the 2-dimensional case of *MaxRank*. Yu et al. [23] extend the principles of [7] to *continuous* top-k queries. The results of these queries must be continuously refreshed as new data records are inserted and old ones are deleted (e.g., in the form of an update stream). At the core of their approach lies the effective maintenance of the *query response surface*, which encodes the score and identity of the k-th result record for any query vector. Similarly to [7], this work solves a different problem from ours, where furthermore k must be fixed and input to the problem (whereas in *MaxRank* the value of k^* is the main unknown).

Vlachou et al. [19, 21] study the *reverse top-k query*. The input comprises a set of query vectors Q and a set of data records D. Specified a positive integer k and a record $\mathbf{p} \in D$, the query reports those vectors in Q for which **p** belongs to the top-k result. Consider the data in Figure 1 and assume that Q includes vectors $\mathbf{q}_1 = (0.7, 0.3)$ and $\mathbf{q}_2 = (0.1, 0.9)$. Figure 1(a) shows the scores of the data records w.r.t. these vectors. The reverse top-2 set of **p** is empty, because **p** does not belong to the top-2 result of any query vector in Q. The reverse top-3 set of **p** includes only $\mathbf{q}_2$, because **p** belongs to the top-3 result of $\mathbf{q}_2$ but not of $\mathbf{q}_1$, etc. Note that k must be given as input to this problem and also that it considers a finite number of query vectors with specific, discrete positions in the query space. This is different from *MaxRank* where k^* is unknown and the query vector **q** could be anywhere in the query space, i.e., there are infinite positions where it could lie at.

Building on the reverse top-k query, Vlachou at al. [20] identify the top-m most influential data records, i.e., those that have the largest number of query vectors in their reverse top-k result. This variant is also inapplicable to *MaxRank* for the same reasons as the original reverse top-k query.

Vlachou et al. [19] also discuss the *monochromatic* reverse top-k query, which computes the parts of the query space where if **q** lies, record **p** belongs to the top-k result. Although **q** is not bound to a finite set of candidate positions, k is still an input to the problem. Moreover, the proposed solution works only for two dimensions; the authors identify the challenges of extending it to higher dimensions and leave it for future work.

Zhang et al. [25] study the *reverse k-ranks* problem. They assume a set of query vectors Q and a set of data records D. Given a positive integer m and a record of interest $\mathbf{p} \in D$, the problem is to shortlist the m query vectors in Q for which **p** ranks the highest. Although conceptually affine to *MaxRank*, this problem is intrinsically different because the considered query vectors are constrained to a given set Q. The proposed pruning techniques are based on the specific positions of the query vectors and on bounds derived by these exact positions. Covering *any* position in the query space is impossible through this approach.

A computational geometric problem that is related to top-k processing is *half-space range reporting* [13]. The problem is to preprocess a dataset D such that, given a hyperplane, we may efficiently compute which records of D lie above it. Algorithms for this problem can be used directly to determine the rank of a data record **p** w.r.t. a given query vector **q** [1]. Although related, this problem is different from *MaxRank*. If there were a finite set of candidate query vectors, we could determine the order of **p** w.r.t. each of them (via half-space range reporting) and trivially identify those that yield the smallest order for it. In *MaxRank*, however, **q** could be anywhere in the query space.

The *skyline* operator is closely related to top-k processing [5]. We say that a record **r** *dominates* another record $\mathbf{r}'$ if all the values of **r** are no smaller than those of $\mathbf{r}'$, and the two records differ in at least one attribute. The skyline of a dataset includes only the records that are not dominated by any other. It holds that the top record w.r.t. any query vector must belong to the skyline [11]; this property has been used for the effective preprocessing of a dataset to facilitate top-k answering [27]. *Branch-and-Bound Skyline* (BBS) [17] is a skyline computation algorithm for data organized by a spatial index, e.g., an R-tree. It is I/O-optimal, i.e., it reads the minimum possible number of disk pages (R-tree nodes) to compute the skyline. BBS can efficiently update the skyline in the event of subsequent record insertions/deletions. Also, it can compute the *k-skyband* – this is a generalization of the skyline, which includes the records that are dominated by fewer than k others.

Dellis and Seeger [8] study the *reverse skyline*. Given a record **p** in a d-dimensional dataset D, the reverse skyline includes the records in D that are not dynamically dominated by any other data record w.r.t. **p**. A record **r** is said to *dynamically dominate* another $\mathbf{r}'$ w.r.t. **p** if the projection of **r** on each of the d axes lies closer to **p** than the corresponding projection of $\mathbf{r}'$. The reverse skyline deals with geographic proximity/surroundings rather than ranking.

Top-k processing is related to *convex hull* computation [3]. The convex hull of a dataset D is the smallest convex set that encloses all the records in D. The top record w.r.t. any query vector must lie on the boundary of the convex hull. Chang et al. [6] use this property in a preprocessing technique for top-k answering. They

[1] **q** and **p** define a hyperplane in data space, above which lie exactly those records that score higher than **p** [23].

materialize a number, say m, of convex hull layers in order to facilitate the processing of top-k queries with $k \leq m$.

MaxRank assesses the potential/impact of a data record in a pool of competing options. In that sense, it is somewhat related to the competitive positioning of new products in an existing market. Several studies consider the creation of new records in a dataset so that they belong to the skyline of the extended dataset, e.g., [22]. Li et al. [12] propose strategies to position a new product in the market, based on the number of alternatives it would dominate or be dominated by. Miah et al. [14] describe techniques to publicize only a subset of a record's attributes, so as to increase its visibility, i.e., the number of queries (from a known set) that would include it in their result. Unlike this stream of work, in *MaxRank* the focal record is given and fixed, i.e., we cannot alter/choose its attributes.

3. PRELIMINARIES

We consider a dataset D that contains n records. Each record $\mathbf{r} \in D$ has d numeric attributes r_i (for $i \in [1, d]$). For ease of presentation, we assume that the domain of each attribute is $[0, 1]$, yet this is not a requirement of our methods. Records are treated as vectors in d-dimensional space; the r_i value of a record is its coordinate in the i-th dimension. In the context of linear top-k queries, user preference is specified by a *query vector* $\mathbf{q}$ that comprises a weight q_i for each dimension. The score of a record $\mathbf{r}$ is defined as its dot product with $\mathbf{q}$, i.e., $S(\mathbf{r}) = \mathbf{r} \cdot \mathbf{q} = \sum_{i=1}^{d} r_i q_i$. The result of a top-$k$ query (w.r.t. $\mathbf{q}$) contains the k records with the highest scores in D. For simplicity, we ignore ties in score.

To visualize, top-k processing corresponds to the sweeping of the data space with a hyperplane that is normal to $\mathbf{q}$. The sweeping direction is from the top corner of the data space (i.e., point $(1, 1, ..., 1)$) towards the origin. The order in which data records are encountered indicates their rank w.r.t. $\mathbf{q}$. That is, the i-th encountered record is the record with the i-th highest score; we say that the *order* of that record is i. Note that a small order means that the record ranks high.

Without loss of generality, in the following we assume that $q_i > 0$ for every dimension, and that $\sum_{i=1}^{d} q_i = 1$. A query vector that abides by these assumptions is called *permissible*. Note that the normalization (i.e., $\sum_{i=1}^{d} q_i = 1$) does not constrain the semantics nor the choice of query vector. To see this, in the sweeping analogy drawn before, the ranking of the records solely depends on the direction (but not the magnitude) of vector $\mathbf{q}$.

MaxRank takes as input the dataset D and a (user- or application-specified) data record $\mathbf{p} = (p_1, p_2, ..., p_d)$, called the *focal record*. We assume that $\mathbf{p}$ belongs to D, although this does not have to be the case. The *MaxRank* problem is (i) to determine the smallest order achievable by $\mathbf{p}$ w.r.t. any permissible query vector, and (ii) to identify all the regions of the query space where $\mathbf{p}$ achieves that order. We denote the smallest order achievable by $\mathbf{p}$ as k^*.

DEFINITION 1. *Given a dataset D and specified a focal record $\mathbf{p} \in D$, the* MaxRank *query reports*

- *value k^*, i.e., the smallest possible order of $\mathbf{p}$ w.r.t. any permissible query vector, and*
- *all the regions of the query space where if $\mathbf{q}$ falls in, the focal record $\mathbf{p}$ achieves order k^*.*

Essentially, the second component of the output is a description of all query vectors that render $\mathbf{p}$ the k^*-th result of the top-k query. As we explain in Section 5, the *MaxRank* query is equivalent to finding the minimal set of records we should remove from D so that $\mathbf{p}$ may become the top record w.r.t. some query vector.

Symbol	Description
D	Dataset
d	Data dimensionality
n	Number of records in D
$\mathbf{r} = (r_1, r_2, ..., r_d)$	Data record with coordinates $r_1, r_2, ..., r_d$
$\mathbf{q} = (q_1, q_2, ..., q_d)$	Query vector with weights $q_1, q_2, ..., q_d$
$\mathbf{p} = (p_1, p_2, ..., p_d)$	Focal record with coordinates $p_1, p_2, ..., p_d$
$S(\mathbf{r})$	Score of $\mathbf{r}$
k^*	The smallest order achievable by $\mathbf{p}$
T	Query space regions where $\mathbf{p}$ has order k^*
τ	Parameter of *incremental MaxRank*
D^+	Set of dominators of $\mathbf{p}$
H_c	Set of half-spaces that contain cell c
R_c	Set of records that correspond to H_c
F_l	Set of half-spaces that contain leaf l
P_l	Set of half-spaces that partly overlap with l

Table 1: Frequently used notation

While *MaxRank* is our main focus, we also consider a generalization of the problem, the incremental *MaxRank* (i.e., *iMaxRank*). The input to *iMaxRank* includes an additional integer τ ($\tau \geq 0$).

DEFINITION 2. *Given a dataset D, a focal record $\mathbf{p} \in D$ and a positive integer τ, the* iMaxRank *query reports k^*, and all the regions of the query space where if $\mathbf{q}$ falls in, the focal record $\mathbf{p}$ is among the top-$(k^* + \tau)$ records.*

In other words, the output of the *iMaxRank* query is a description of all permissible query vectors for which $\mathbf{p}$ achieves an order between k^* and $k^* + \tau$. As mentioned in Introduction, an application of *MaxRank* is to offer the option provider a description of her most likely customer profiles; *iMaxRank* helps reach out to the parts of the query space (i.e., potential preferences) where the appeal of $\mathbf{p}$ is very strong, albeit not necessarily the strongest possible. We focus our algorithmic descriptions on *MaxRank* processing, but we extend our techniques to *iMaxRank* as well.

In terms of data organization, we assume that D is indexed by a spatial access method. In our implementation we use an R*-tree [2] due to its proliferation. By default, data and index reside in secondary storage, although our experiments evaluate the in-memory scenario too. Table 1 summarizes frequently used notation.

In Section 4, we present a first-cut algorithm for *MaxRank* in two dimensions ($d = 2$). In Section 5, we describe a basic approach for higher dimensions ($d \geq 2$), and in Section 6 we develop its optimized version. A general note on d is that the top-k problem, to begin with, and hence *MaxRank* too, suffer from the dimensionality curse and are not meaningful for large d. In the Appendix, we show that as d grows, the score of the top record approaches quickly that of the lowest-scoring record in the entire dataset, exhibiting a behaviour (and usefulness deterioration) very similar to the nearest neighbor query [4]. Thus, we focus on low-dimensional data.

4. A FIRST-CUT SOLUTION FOR $d = 2$

The characteristics of *MaxRank* in the special case of $d = 2$ allow for a *first-cut algorithm* (FCA). The 2-dimensional case reveals important facts about *MaxRank* in general dimensionality too.

Since $\sum_{i=1}^{d} q_i = 1$, in two dimensions it holds that $q_2 = 1 - q_1$. Hence, the score of a record $\mathbf{r}$ is defined as $S(\mathbf{r}) = r_1 q_1 + r_2(1 - q_1)$. In turn, this means that the plot of $S(\mathbf{r})$ versus q_1 is a line. Figure 2 plots the score of each record in Figure 1(a) versus q_1. Every intersection of the score line of $\mathbf{p}$ with the score line of another record $\mathbf{r}$ indicates a reordering between the two at the corresponding q_1 value. FCA computes all these intersections (circled in Figure 2), and sorts them in increasing q_1 order.

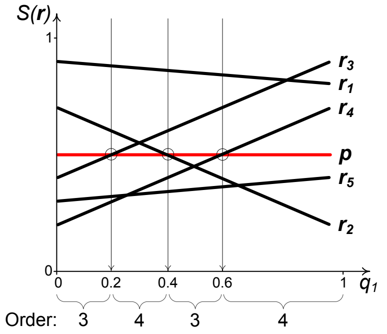
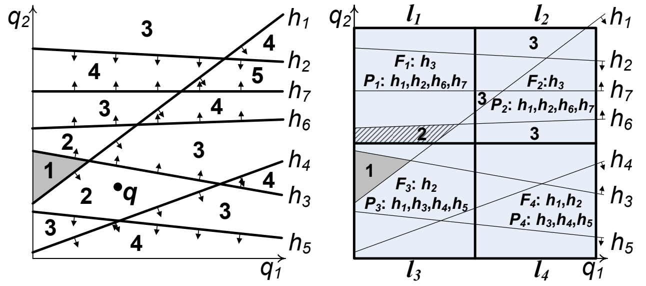


Figure 2: Score vs. q_1, and order of **p** in different intervals



(a) Half-space arrangement (b) BA example

Figure 3: *MaxRank* in (reduced) query space, $d = 3$

At $q_1 = 0$ the order of **p** is equal to the number of records that have $r_2 > p_2$ plus one. At the first (i.e., leftmost) intersection, the order of **p** increases or decreases by one (when the slope r_1 of the intersecting score line is greater or smaller than that of **p**, respectively). In our example, the initial order of **p** is 3, which at $q_1 = 0.2$ becomes 4 (due to the intersection induced by $\mathbf{r}_3$). Repeating this process for the remaining intersections (in increasing q_1 order) produces a partition of the q_1 domain into intervals, and the corresponding order of **p** in each of them[2]. In our example, the order of **p** per interval is shown at the bottom of Figure 2.

Given this information, it is trivial to answer a *MaxRank* or *iMaxRank* query. E.g., the smallest order achievable in this case is $k^* = 3$ and the corresponding intervals of q_1 are $(0, 0.2)$ and $(0.4, 0.6)$. Recall that these intervals fully determine **q**, because $q_2 = 1 - q_1$. That is, they are equivalent to the shaded regions of the query space in Figure 1(b). Computing the intersection between two lines takes constant time. Thus, probing every score line against that of **p** takes a total of $O(n)$ time, producing a maximum of n intersections. Sorting the intersections by q_1 takes another $O(n \log n)$ time, which determines the complexity of FCA.

A key observation is that there may be multiple intervals where **p** attains the minimum order k^*. Also, there is no particular trend as to whether the order increases or decreases as q_1 grows. This is a general characteristic of *MaxRank*, independent of dimensionality.

5. BASIC APPROACH FOR $d > 2$

In this section, we present our *basic approach* (BA) for *MaxRank* processing in higher dimensions. We begin with an observation that will help prune D. Every record $\mathbf{r} \in D$ that dominates **p** has a smaller order (equivalently, a higher score) than **p** w.r.t. any permissible query vector [5]. We call such records *dominators*. Letting D^+ be the set of dominators, it holds that $k^* > |D^+|$. Other than incrementing k^* however, dominators can be safely disregarded in *MaxRank* processing. Symmetrically, each record dominated by **p** is guaranteed to have an order greater than **p** w.r.t. any query vector. We call these records *dominees* and also disregard them. The remaining records, i.e., those that are neither dominators nor dominees, are called *incomparable*. For any incomparable record **r**, there are query vectors for which $S(\mathbf{r}) > S(\mathbf{p})$, and at the same time there are query vectors for which $S(\mathbf{r}) < S(\mathbf{p})$.

In general dimensionality, the focal object partitions the data space into 2^d regions – each of these regions is an axis-parallel hyper-rectangle, defined by **p** and a corner of the data space. The region with all coordinates larger than **p** contains the dominators, and the one with all coordinates smaller contains the dominees. The remaining $2^d - 2$ regions contain incomparable records – *MaxRank* (and *iMaxRank*) processing needs to focus on these regions only. The said focusing can be easily done using the R*-tree index on D. The only necessary information about the dominators is their number (as it increments k^*) which can be computed either from the R*-tree via a range search, or even faster if an *aggregate* R*-tree is used instead [16]. The latter is a regular R*-tree where each index entry is augmented with the total number of records in the sub-tree rooted at it. The same pruning applies to $d = 2$ as well, and we use it to enhance FCA too, i.e., in the 2-dimensional example of Figure 1(a), FCA disregards record $\mathbf{r}_1$ (dominator) and record $\mathbf{r}_5$ (dominee).

Consider now an incomparable record **r**. Its order is smaller than **p** when and only when $S(\mathbf{r}) > S(\mathbf{p}) \Leftrightarrow \sum_{i=1}^{d} r_i q_i > \sum_{i=1}^{d} p_i q_i \Leftrightarrow \sum_{i=1}^{d} (r_i - p_i) q_i > 0$. The interpretation of the last inequality in (d-dimensional) query space is a half-space that passes through the origin. As discussed in Section 3, however, we enforce that $\sum_{i=1}^{d} q_i = 1$ (which, we repeat, does not sacrifice generality). This allows removing the d-th dimension of the query space, because $q_d = 1 - \sum_{i=1}^{d-1} q_i$. The inequality $S(\mathbf{r}) > S(\mathbf{p})$ can therefore be rewritten as $\sum_{i=1}^{d-1} (r_i - r_d) q_i + r_d > \sum_{i=1}^{d-1} (p_i - p_d) q_i + p_d \Leftrightarrow \sum_{i=1}^{d-1} (r_i - r_d - p_i + p_d) q_i > p_d - r_d$. The latter corresponds to a half-space in the $(d-1)$-dimensional query space with axes $q_1, q_2, ..., q_{d-1}$; we refer to this $(d-1)$-dimensional space as the *reduced query space*. The supporting hyperplane of the half-space is given by equation $\sum_{i=1}^{d-1} (r_i - r_d - p_i + p_d) q_i = p_d - r_d$. To summarize, the inequality $S(\mathbf{r}) > S(\mathbf{p})$ holds if and only if the query vector lies inside that half-space.

Via the above process, each incomparable record is mapped into a half-space in the reduced query space. Assume that $d = 3$ and that there are 7 incomparable records. In Figure 3(a), we illustrate the reduced query space, with axes q_1 and q_2 (since q_3 is removed). Each incomparable record $\mathbf{r}_i$ (for $1 \leq i \leq 7$) is mapped into a half-plane h_i with supporting line $\sum_{i=1}^{2} (r_i - r_3 - p_i + p_3) q_i = p_3 - r_3$ (note that the half-planes and their supporting lines are the 2-dimensional counterparts of the half-spaces and the supporting hyperplanes we used in our general-dimensionality description). The arrows drawn on the supporting line of each half-plane point towards its interior, to indicate containment. For instance, consider h_3. Every query vector that lies inside the half-plane (i.e., above its supporting line), renders $S(\mathbf{r}_3) > S(\mathbf{p})$.

Let $\mathcal{A}$ be the arrangement of all induced half-spaces. $\mathcal{A}$ partitions the reduced query space into disjoint *cells* [3], which are convex polytopes. Any point inside a cell falls in exactly the same set of half-spaces. In our 3-dimensional example, Figure 3(a) shows the arrangement of 7 half-planes, which includes 14 cells.

[2] A similar score line intersection approach is used in [19] for the monochromatic reverse top-k query in two dimensions.

Let c be a cell, and H_c be the set of half-spaces that contain it, i.e., half-spaces that fully include c. We refer to $|H_c|$ (i.e., the number of half-spaces that contain c) as the *order* of the cell. We denote by R_c the set of incomparable records that induce the half-spaces in H_c (there is a one-to-one correspondence between records in R_c and half-spaces in H_c). Lemma 1 is derived directly from the definition of the arrangement.

LEMMA 1. *For every query vector* $\mathbf{q}$ *that lies in a cell* c *of* $\mathcal{A}$*, the incomparable records that have an order smaller than* $\mathbf{p}$ *(i.e., a score higher than* $\mathbf{p}$*) are those and only those records in* R_c*.*

It follows from the lemma that when $\mathbf{q}$ is inside c, the order of $\mathbf{p}$ is $|D^+| + |H_c| + 1$. That is, the records that score higher than $\mathbf{p}$ are exactly those in $D^+ \cup R_c$. In Figure 3(a), consider the cell where vector $\mathbf{q}$ lies. The set of half-planes that contain this cell is $H_c = \{h_1, h_2\}$, thus its order is 2. For $\mathbf{q}$ and for any query vector that falls inside that cell, records $\mathbf{r}_1$ and $\mathbf{r}_2$ are the only incomparable records that score higher than $\mathbf{p}$. The following corollary of Lemma 1 forms the basis of our methodology.

COROLLARY 1. *Let* T *be the set of cells in* $\mathcal{A}$ *with the minimum order* $|H_c|$*. The cells in* T *are those and only those regions of the (reduced) query space where* $\mathbf{p}$ *attains the minimum order* k^**. Also,* $k^* = |D^+| + |H_c| + 1$*, where* $|H_c|$ *is that minimum cell order.*

MaxRank outputs T and k^*. In the example of Figure 3(a), the number in each cell indicates its order. T includes only the shaded cell, whose order is 1. Thus, $k^* = |D^+| + |H_c| + 1 = |D^+| + 2$. Note that in general there may be multiple cells with the same (minimum) order $|H_c|$; this is the case in Figure 2, for instance.

In Section 3, we mentioned that *MaxRank* is equivalent to finding the minimal set(s) of records to remove from D so that $\mathbf{p}$ becomes the top record w.r.t. some query vector. Each of the cells $c \in T$ determines a minimal such set, that is, the union of D^+ and the corresponding R_c. E.g., in Figure 3(a) the minimal set to remove is $\mathbf{r}_2$ and the dominators of $\mathbf{p}$.

Returning to *MaxRank* processing, Corollary 1 is key but far from leading to a practical solution. As explained previously, there are $2^d - 2$ incomparable regions. If the data records are randomly and uniformly distributed, and $\mathbf{p}$ is near the center of the data space, the incomparable records comprise a fraction $\frac{2^d-2}{2^d}$ of the dataset. This implies a huge number of half-spaces. To compute their arrangement $\mathcal{A}$, let alone the order of its cells, has a complexity of $O(n^d)$ [1]. Furthermore, the existing algorithms for arrangement computation are theoretical in nature and involve large constant factors. The key fact is that we do not need to compute nor store the entire arrangement, but only the smallest-order cells.

To achieve this, we use a space partitioning index on the half-spaces, in the $(d-1)$-dimensional reduced query space. The structure we employ is an augmented Quad-tree. Its leaves define a partitioning of the space and are processed one by one. Leaves contained in too many half-spaces to affect the *MaxRank* (or *iMaxRank*) result are pruned. The augmented Quad-tree and the leaf pruning strategy are described in Section 5.1.

Our pruning strategy disregards the majority of Quad-tree leaves. For those that are not pruned, however, within-leaf processing is necessary. The latter is still a (constrained) half-space arrangement problem. In Section 5.2, we describe a practical technique for within-leaf processing that capitalizes again on the fact that not all cells of the (leaf-constrained) arrangement need to be considered, and relies on half-space intersection, a much simpler problem than arrangement computation.

5.1 Half-space Index and Leaf Pruning

We organize the half-spaces (induced by incomparable records) by a Quad-tree. The leaves of the Quad-tree define a partitioning of the reduced query space. The Quad-tree is augmented with information that allows deriving efficiently for any leaf l (i) the set of half-spaces F_l that fully contain l and (ii) the set of half-spaces P_l that partly overlap with it.

In particular, we maintain two sets of half-spaces for each leaf (one for full containment and another for partial overlap). In contrast, for the internal nodes we only maintain the set of half-spaces that fully contain them. Importantly, in every node of the tree, we exclude from the containment set those half-spaces that already contain the node's parent – recording such half-spaces is redundant, since every half-space that contains the parent also contains its descendants. The half-spaces are inserted one by one into the tree. A leaf l is split when $|P_l|$ exceeds a certain threshold.

Given a leaf l, we can efficiently derive set F_l as the union of the leaf's full containment set (as maintained in the Quad-tree) and the full containment sets of all its ancestors in the tree. Set P_l is simply the partial overlap set kept with the leaf node.

Having built the Quad-tree, BA extracts all its leaves. The leaves partition the reduced query space and, implicitly, the half-space arrangement too. Consider a leaf l in the Quad-tree. Cardinality $|F_l|$ is a lower bound for the order of all cells of the arrangement that fall in l. BA considers leaves, i.e., partitions of the reduced query space, in increasing $|F_l|$ order. For each of them, the within-leaf processing module of Section 5.2 identifies the cell(s) inside with the smallest order. BA terminates when the smallest cell order found so far is smaller than cardinality $|F_l|$ of the next leaf to be processed (or smaller than $|F_l| - \tau$, in the *iMaxRank* case). This technique disregards (without within-leaf processing) the leaves whose $|F_l|$ cardinality is too large to affect the *MaxRank* result.

Example: Assume that we construct an augmented Quad-tree with the half-spaces in Figure 3(a). Suppose that this produces the four leaves l_1, l_2, l_3, l_4 in Figure 3(b). Inside each of the leaves, we indicate the respective full containment and partial overlap sets. For example, l_1 has $F_1 = \{h_3\}$ and $P_1 = \{h_1, h_2, h_6, h_7\}$. BA considers the leaves in increasing order of cardinality $|F_l|$, which is 1, 1, 1 and 2 for l_1, l_2, l_3, l_4, respectively. The tie among the first three is resolved arbitrarily, e.g., l_1, l_3, l_2. The within-leaf module is invoked to compute the cell(s) in l_1 with the smallest order, i.e., the striped cell with order 2, which becomes the interim result. The next leaf, l_3, has $|F_3| = 1$ which is smaller than the order in the interim result (i.e., 2). Thus, the within-leaf module is invoked for l_3, retrieving the shaded cell with order 1, which becomes the new interim result. The next leaf, l_2, has $|F_2| = 1$. Although that cardinality is equal to the order in the interim result (i.e., 1), the within-leaf module must still be invoked for l_2 because it may include additional cells with order 1. The smallest-order cells found in l_2 have order 3 (there are three such cells), and fail to enter the interim result. The last leaf, l_4, has $|F_4| = 2$ which is greater than the order in the interim result (i.e., 1) and is pruned without any within-leaf processing, because any cell in l_4 is guaranteed to have an order of at least $|F_4| = 2$. The interim result (i.e., the shaded cell with order 1) is reported as the *MaxRank* result.

Note that if a cell of the arrangement overlaps with multiple leaves of the Quad-tree, its extent is essentially broken into an equal number of parts. Each of these parts is considered if and when BA processes the corresponding leaf. That is, if the cell belongs to the *MaxRank* result, its extent will be reported in its entirety, albeit in parts. In Figure 3(b), if the shaded cell overlapped with leaves l_1 and l_3, its respective parts would be identified independently by the within-leaf processing of l_1 and of l_3.

Before presenting the within-leaf processing module, we remark that unlike the original query space, the reduced query space is no longer a unit hyper-cube. Because $\sum_{i=1}^{d} q_i = 1$ and $q_d > 0$, it must hold that $\sum_{i=1}^{d-1} q_i < 1$. This constraint corresponds to a half-space whose supporting hyperplane intersects each axis at value 1. That is, the reduced query space is only a half of the unit hyper-cube in $(d-1)$-dimensions. In Figure 3, for example, although we visualize the reduced query space as a square, in reality it is a right triangle with unit legs and right angle at the origin. Similarly, when $d = 4$, it is a (3-dimensional) trirectangular tetrahedron with right angle at the origin, etc. The implication is that we discard any Quad-tree node that is completely outside half-space $\sum_{i=1}^{d-1} q_i < 1$. For simplicity, our visualizations ignore this constraint.

5.2 Within-leaf Processing

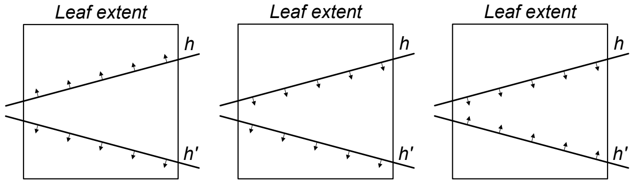
Processing within a leaf is not straightforward. Let l be the leaf to be processed. Every half-space in F_l increments the order of $\mathbf{p}$ by one (when $\mathbf{q}$ is anywhere inside l). On the other hand, the half-spaces in P_l define an arrangement that partitions the leaf into cells. Each cell c in that arrangement lies inside a subset of the half-spaces in P_l. The number of those half-spaces is called the *p-order* of c. In other words, the order of c (as defined and used so far) is equal to its *p-order* plus $|F_l|$. To avoid the cost and complexity of computing the entire arrangement of P_l, we compute individual cells one by one, prioritized by their *p-order*, as follows.

Each cell is defined by a bit-string $\mathbf{b}$. The i-th bit b_i in the string corresponds to the i-th half-space in P_l. Bit b_i is 1 if the cell is inside the half-space or 0 otherwise. Given a bit-string, we can produce the cell by half-space intersection (of the half-spaces with $b_i = 1$ and the complements of those with $b_i = 0$) and a subsequent intersection with the leaf's extent, since the module focuses only on what happens within the leaf. The number of 1 bits in $\mathbf{b}$ (i.e., the *Hamming weight* of $\mathbf{b}$) is equal to the *p-order* of the cell.

A way to implement the within-leaf module is to consider all possible bit-strings in increasing Hamming weight, starting from 0 (i.e., from bit-string 00...0). If the corresponding cell (computed as described above) has non-zero extent, it is the cell with the smallest order in l and the module terminates. Alternatively (i.e., if the intersection is empty), we consider all bit-strings with Hamming weight 1. For each of them, we compute the corresponding cell. If one or more of the produced cells have non-zero extent, they are returned to the calling algorithm (i.e., BA), and the module terminates. Otherwise, we consider all bit-strings of Hamming weight 2, and so on, until for some Hamming weight we encounter at least one cell with non-zero extent.

Example: Consider within-leaf processing for l_1 in Figure 3(b), where $P_1 = \{h_1, h_2, h_6, h_7\}$. Bit-string 0000 corresponds to the intersection of complements of all h_1, h_2, h_6, h_7, which is empty. Thus, bit-strings with Hamming weight 1 are considered, i.e., 1000, 0100, 0010, 0001. Bit-string 0100 corresponds to the intersection of h_2 with the complements of h_1, h_6, h_7, which has non-zero extent (shown striped in the figure). The other three bit-strings lead to empty intersections. Hence, the striped cell is reported as the smallest-order cell in l_1 with *p-order* 1 and cell order $1 + |F_1| = 2$.

Producing a cell (via half-space intersection) takes a hefty $O(n^{d/2})$ time [3], where $n = |P_l|$ in our case. To optimize the module, we make some observations that allow us to disregard some bit-strings without half-space intersection, because the resulting cells are guaranteed to be empty. Without loss of generality, assume that there is no pair of half-spaces in P_l with identical supporting hyperplanes. Given any two half-spaces h and h' in P_l whose supporting hyperplanes *do not intersect within* l, there are only three possible containment statuses between them; (i) they are disjoint, (ii) one (say, h) contains the other, or (iii) they overlap but none completely includes the other. These cases are illustrated in Figure 4 for $d = 3$ (i.e., 2-dimensional reduced query space). In the first case (disjoint), the bits of the two half-spaces cannot be both 1 at the same time, i.e., they would lead to an empty intersection. Similarly, in the second case (h contains h'), the bit for h' cannot be 1 when the bit for h is 0. In the third case (overlap but no complete inclusion), the bits cannot be both 0. Note that these three statuses refer only to the parts of the half-spaces that fall within l – their parts that fall inside other leaves of the Quad-tree will be considered if/when these other leaves are processed by BA.



(a) Disjoint (b) Inside (c) Overlap

Figure 4: Containment statuses, reduced query space, $d = 3$

In the beginning of the within-leaf module, we consider each pair of half-spaces in P_l. For pairs where hyperplanes do not intersect within l, we classify them under one of the three categories, and record the respective binary condition. Bit-strings are considered in increasing Hamming weight, as per normal, but we dismiss without half-space intersection those that violate a binary condition.

Within-leaf processing for *iMaxRank* is as described above, but we examine bit-strings with Hamming weights up to τ units larger (compared to the *MaxRank* case).

5.3 Correctness and Completeness of BA

Lemma 1 and Corollary 1 are the basis of BA. By constructing the augmented Quad-tree to reflect all incomparable records, its leaves partition the (reduced) query space, and along with it, the half-space arrangement $\mathcal{A}$. That is, every cell in T (i.e., in the *MaxRank* result) is covered by one or more leaves of the Quad-tree. BA prunes a leaf l only if its full containment set F_l includes more half-spaces than the smallest cell order found so far. By definition, every arrangement cell (or part of an arrangement cell) that falls in l is contained in at least the half-spaces in F_l, and thus its order is at least $|F_l|$. Hence, BA pruning is safe, i.e., it only disregards leaves that are guaranteed not to include any cell from T.

It remains to show that within-leaf processing is also correct and complete. The module considers exhaustively all possible combinations of bits (i.e., combinations of containment in the half-spaces of P_l), in increasing Hamming weight. Let i be the first (i.e., smallest) Hamming weight where we find a bit-string that corresponds to non-empty intersection. First, we are certain that i is the smallest *p-order* in the leaf (equivalently, that the smallest cell order in the leaf is $i + |F_l|$), and thus there is no need to consider any bit-string with Hamming weight greater than i. Second, the within-leaf module reports *all* the cells (non-empty intersections) in the leaf with *p-order* i, since it examines *every* bit-string of Hamming weight i.

6. ADVANCED APPROACH

The basic approach (BA) offers a viable *MaxRank* solution. It suffers, however, from a major drawback. To exemplify, in the simple scenario we assumed in Section 5, we estimated the incomparable records to comprise a fraction $\frac{2^d-2}{2^d}$ of the dataset. While for $d = 2$ that is half the dataset, as the dimensionality grows, the fraction quickly approaches 1. This implies that BA needs to access almost the entire dataset and insert into the Quad-tree an excessive number of half-spaces. The *advanced approach* (AA) circumvents this problem, while retaining the correctness and exactness of the solution. It achieves this by accessing incomparable records (equivalently, by inserting half-spaces) progressively and only when they could affect *MaxRank* processing.

6.1 Outline of Methodology

In Section 5, we discussed pruning the dataset based on the dominance relationship between **p** and the data records. The crux in AA is to leverage on the dominance relationship *among* the data records, in order to avoid considering some of them, thus saving on the access and processing costs. In the rest of this section, we focus on data records that are incomparable to **p**; dominators and dominees are disregarded, similarly to BA.

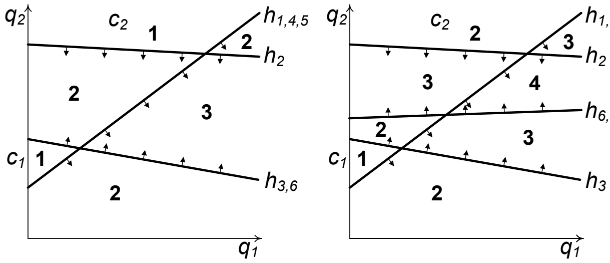
Consider two records **r** and **r**′, where the first dominates the second. By definition, the order of **r** is always smaller than that of **r**′, for any permissible query vector. This implies that **r**′ cannot score higher than **p** unless **r** already scores higher than **p**. In the reduced query space, this means that the half-space induced by **r** contains that of **r**′, regardless of what the focal record is. The main idea in AA is to refrain from processing **r**′ unless **r** is processed first. We achieve this by *subsuming* (the half-space induced by) **r**′ under **r**. The objective is to accurately identify the minimum order cell(s) of the half-space arrangement, without considering the majority of subsumed half-spaces.

To formalize, the half-spaces of records that subsume no other records are called *singular* – such are all the half-spaces we discussed in the previous sections. If a record subsumes at least one other, its half-space is called *augmented*. Each augmented half-space is (implicitly) associated with the half-spaces/records it subsumes. Note that we do not subsume all the dominees of a record under it, but just some (we will explain which later). On the other hand, only the dominees of a record could be subsumed under it. AA executes in iterations, through which it maintains a *mixed arrangement*, i.e., the arrangement of a mix of singular and augmented half-spaces. Similarly to Section 5.1, the mixed arrangement is only implicitly maintained, using an augmented Quad-tree.

In the first iteration, AA constructs the mixed arrangement from half-spaces (be they singular or augmented) that are not subsumed under any other. Consider again the example in Figure 3(a), and assume that $\mathbf{r}_1$ dominates $\mathbf{r}_4, \mathbf{r}_5$; also, that $\mathbf{r}_3$ dominates $\mathbf{r}_6$, which in turn dominates $\mathbf{r}_7$. Figure 5(a) shows the result of subsumption. Due to the stated dominance relationships, half-space h_1 subsumes h_4 and h_5, becoming an augmented half-space denoted by $h_{1,4,5}$. Similarly, h_3 subsumes h_6 becoming the augmented half-space $h_{3,6}$. Half-space h_6 subsumes h_7, however, h_7 does not appear in the figure (neither as a singular half-space nor as the augmented $h_{6,7}$). The reason is that, as already stated, the first iteration considers only the half-spaces that are not subsumed under any other.

Returning to our example, in the first iteration, AA initializes the mixed arrangement by the half-spaces in Figure 5(a), and identifies the cell(s) with the smallest order. Practically, this means that the half-spaces are inserted into the (initially empty) Quad-tree, and the smallest order cell(s) are identified by the process in Section 5.1. In the following, we refer directly to the mixed arrangement and to its cells (rather than the inner workings of the Quad-tree).

In Figure 5(a), the number drawn in each cell indicates its order in the mixed arrangement. The cells with the smallest order are c_1 and c_2 (both have order 1). All half-spaces (i.e., h_2) that contain c_1 are singular. Therefore, *its extent and order in the mixed arrangement is identical to what it would be in the arrangement induced by all incomparable records* (shown in Figure 3(a)). Note that the boundary of c_1 may be defined by an augmented half-space (i.e., $h_{1,4,5}$), but since that half-space does not contain c_1, the half-spaces subsumed under it (i.e., h_4 and h_5) cannot affect the order nor the extent of c_1.



(a) First iteration (b) Second iteration

Figure 5: AA example in (reduced) query space, $d = 3$

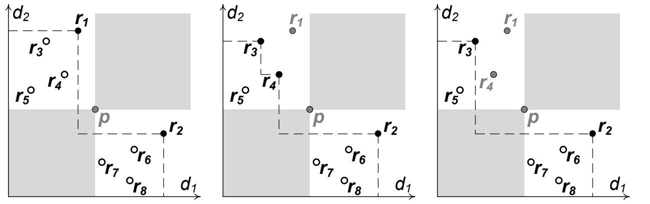
On the contrary, c_2 is inside an augmented half-space, i.e., $h_{3,6}$. To determine accurately the order and extent of c_2, AA needs to *expand* that augmented half-space, i.e., to replace it by its singular form and the half-spaces it subsumes. The rationale is that some of the half-spaces subsumed under $h_{3,6}$ could overlap with c_2. In our example, $h_{3,6}$ subsumes $h_{6,7}$. Thus, $h_{3,6}$ is replaced by h_3 and $h_{6,7}$. Figure 5(b) illustrates the arrangement after the expansion. If in the beginning of the first iteration, c_2 were inside more augmented half-spaces (in addition to $h_{3,6}$), they too would be expanded. The first iteration terminates after the expansion.

The second iteration commences with c_1 already identified with (accurate) order 1, and the mixed arrangement shown in Figure 5(b). The smallest cell order in this arrangement is 2, which is already greater than that of c_1. Thus, AA terminates with $k^* = |D^+| + |H_{c_1}| + 1 = |D^+| + 2$, and $T = \{c_1\}$ as the region of the query space where this order is achieved (recall that $|D^+|$ denotes the number of dominators of **p**, and $|H_{c_1}|$ the order of c_1). Observe that AA processes the *MaxRank* query without considering half-spaces h_4, h_5, h_7.

To provide perspective, the order of each cell in the mixed arrangement is a lower bound of the cell's actual order (i.e., its order in the complete arrangement induced by all incomparable records). For instance, cell c_2 has an order of 2 in Figure 5(b), whereas its actual order (in Figure 3(a)) is 3. AA progressively and selectively drills down into the smallest-order cells of the mixed arrangement, by expanding the augmented half-spaces that contain those cells. As explained previously, *if a cell is not inside any augmented half-space, its order and extent are accurate*. AA terminates when the smallest *accurate* cell order is smaller than all the other cell orders in the mixed arrangement. The cells with that smallest accurate order form set T of the *MaxRank* result.

An important feature in AA is nested subsumption, e.g., $h_{3,6}$ subsumes $h_{6,7}$ which subsumes h_7. The nested subsumption allows for prioritized processing and enables AA to defer/avoid consideration of half-spaces that might not affect the *MaxRank* result. For example, referring to Figure 5 and the expansion of $h_{3,6}$, AA inserts into the mixed arrangement half-space $h_{6,7}$, but not h_7. The latter will only be inserted if $h_{6,7}$ is expanded in a future iteration.

Extending AA to *iMaxRank* is straightforward. After the identification of k^* (as in the basic *MaxRank* case), AA keeps considering cells in increasing order and terminates when all un-reported cells in the mixed arrangement have order larger than $k^* + \tau - |D^+|$.



(a) First iteration (b) Second iteration (c) Third iteration

Figure 6: Implicit subsumption in data space, $d = 2$

In our presentation so far, we assumed that subsumption was somehow performed before AA commences. To subsume half-spaces beforehand would incur high access cost, because it requires reading all incomparable records. It would also be unadaptive to the data and the AA process. In Section 6.2, we solve these problems by making the subsumption *implicit* and *decided dynamically*, depending on which half-spaces are being expanded.

6.2 Implicit Subsumption Strategy

The description holds for any dimensionality, but for the sake of visualization, we use $d = 2$. Consider the example in Figure 6. Figure 6(a) shows the records that are incomparable to $\mathbf{p}$. The dominators and dominees fall in the gray areas and are disregarded, as per normal. In its first iteration, AA computes the skyline of the incomparable records. The skyline is represented by the dashed line and comprises $\mathbf{r}_1, \mathbf{r}_2$. AA inserts into the mixed arrangement one half-space for each skyline record. At this point, all half-spaces are treated as augmented, since AA is oblivious of what lies beneath; non-skyline records ($\mathbf{r}_3, \mathbf{r}_4, ..., \mathbf{r}_8$, shown hollow) have not been accessed nor reflected into the mixed arrangement.

Assume that the first iteration requires expansion of h_1 (the half-space induced by $\mathbf{r}_1$). AA first marks h_1 as a singular half-space. Then, it implicitly removes $\mathbf{r}_1$ from the set of incomparable records, and recomputes/updates the skyline. This introduces $\mathbf{r}_3$ and $\mathbf{r}_4$ into the skyline, and their respective half-spaces are inserted into the mixed arrangement (marked as augmented). Figure 6(b) shows the updated skyline. In retrospect, we can consider that $\mathbf{r}_3$ and $\mathbf{r}_4$ were previously subsumed under $\mathbf{r}_1$. However, their subsumption was not determined in advance, neither did it require accessing $\mathbf{r}_3$ and $\mathbf{r}_4$ prior to the expansion of h_1.

Suppose that the second iteration requires expansion of h_4. This half-space is marked as singular and $\mathbf{r}_4$ is removed from the skyline. The updated skyline includes $\mathbf{r}_2, \mathbf{r}_3$, as shown in Figure 6(c). Since the skyline includes no new records, no half-spaces are inserted into the mixed arrangement. Focus now on $\mathbf{r}_5$, which is not yet encountered/accessed. That record could be subsumed under either $\mathbf{r}_3$ or $\mathbf{r}_4$. Since $\mathbf{r}_4$ is expanded first, AA implicitly subsumes $\mathbf{r}_5$ under $\mathbf{r}_3$, thus postponing the access of $\mathbf{r}_5$ (and the insertion of h_5 into the mixed arrangement) until h_3 is expanded, if at all. That is, AA decides subsumption dynamically, aiming to defer, and potentially avoid, unnecessary record accesses and half-space insertions.

Practically, AA maintains the skyline of non-expanded records, and maps its members into augmented half-spaces. If the current iteration of AA expands some skyline records, they are removed from the skyline and their half-spaces are marked as singular. The skyline is then updated, and its new members introduce new (augmented) half-spaces. We perform skyline computation and maintenance by the I/O-optimal BBS algorithm [17] on the R*-tree of D. When records are expanded (and removed from the skyline), BBS reuses its search heap to incrementally update the skyline, without re-accessing the same R*-tree nodes and records.

Algorithm 1 summarizes the complete AA process. The mixed arrangement is abbreviated as MA. Variable o^* stores the smallest (accurate) cell order found so far. Set E holds the augmented half-spaces to be expanded in the current iteration. Lines 15-17 perform half-space expansion and update the skyline of incomparable records accordingly.

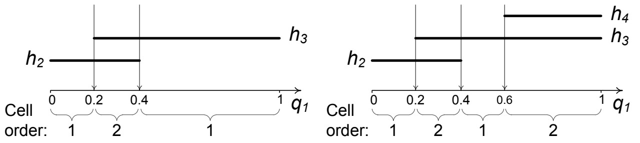
```
Algorithm 1: Advanced Approach
1  Set $o^* = +\infty$ and $T = \emptyset$;
2  $SL$ = the skyline of incomparable records;
3  Insert into MA an augm. half-space for each record in $SL$;
4  while TRUE do
5      Identify the cell(s) in MA with the minimum order;
6      if that minimum order is greater than $o^*$ then
7          Break;
8      $E = \emptyset$;
9      for every cell c identified in Line 5 do
10         if c is not inside any augm. half-space then
11             $T = T \cup \{c\}$;
12             Set $o^*$ to the order of $c$;
13         else
14             Insert into $E$ all augm. half-spaces that contain $c$;
15     Mark every half-space $h \in E$ as singular in MA;
16     Update $SL$ by ignoring all expanded records;
17     Insert into MA an augm. half-space for each new record in $SL$;
18 Report $T$ and $k^* = |D^+| + o^* + 1$;
```

6.3 AA in Special Case of $d = 2$

A specialized version of AA is necessary for $d = 2$, since the reduced query space is 1-dimensional. AA proceeds as before, but the mixed arrangement is maintained using a sorted list instead of a Quad-tree. Consider the data in Figure 1(a). The incomparable records are $\mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4$, where $\mathbf{r}_3$ dominates $\mathbf{r}_4$.

In the first iteration, the skyline includes $\mathbf{r}_2, \mathbf{r}_3$ whose half-spaces need to be reflected in AA's mixed arrangement. Figure 7(a) illustrates the reduced query space; it is 1-dimensional, reflecting the domain of q_1. Here, half-spaces become half-lines. E.g., inequality $S(\mathbf{r}_2) > S(\mathbf{p})$ translates to $q_1 < 0.4$, and $S(\mathbf{r}_3) > S(\mathbf{p})$ to $q_1 > 0.2$. To store the mixed arrangement, we represent each half-line by a value-direction pair. E.g., the half-line of $\mathbf{r}_2$ is represented as $\langle 0.4, \leftarrow\rangle$ and that of $\mathbf{r}_3$ as $\langle 0.2, \rightarrow\rangle$. We maintain the pairs in a sorted list in ascending order of q_1 values, and keep track of the number of ← pairs in the list. The sorted list defines as many cells as its cardinality plus one, delimited by the values of consecutive pairs. In Figure 7(a), for instance, there are 3 cells. To determine the order of the cells, we scan the list from left to right. The first cell $(0, 0.2)$ has order equal to the total number of ← pairs in the list, i.e., 1. If the first pair in the list is ←, the second cell has the order of the first cell decremented by one; otherwise, the order is incremented by one, and so on for the subsequent cells.

In Figure 7(a), the minimum order cells are $(0, 0.2)$ and $(0.4, 0.6)$. Expansion of their containing augmented half-lines (i.e., h_2 and h_3, respectively) turns them into singular, accesses $\mathbf{r}_4$ (previously subsumed under $\mathbf{r}_3$), and introduces h_4 into the mixed arrangement, i.e., inserts $\langle 0.6, \rightarrow\rangle$ into the sorted list. Figure 7(b) illustrates the new arrangement. AA terminates in the second iteration since the minimum order cells $(0, 0.2)$ and $(0.4, 0.6)$ fall inside singular half-lines only, i.e., their order is accurate. To support efficient updates, the sorted list is implemented as a sorted container, e.g., a red-black tree.



(a) First iteration (b) Second iteration

Figure 7: AA example for $d = 2$

Compared to FCA (in Section 4), the 2-dimensional version of AA is expected to access/process significantly fewer records. Recall that FCA needs to consider *all* incomparable records in order to compute the intersections of their score lines with that of **p**.

6.4 Correctness and Completeness of AA

Each incomparable record corresponds to a half-space. Every such half-space is either explicitly reflected in the mixed arrangement (as a singular or augmented half-space) or subsumed under an augmented half-space. Thus, the mixed arrangement accounts for *all* incomparable records to **p** (be it explicitly or implicitly).

AA terminates when the smallest-order cell(s) in the mixed arrangement is/are not inside any augmented half-space, and any other cell in the mixed arrangement has greater order. The aforementioned smallest-order cells form set T in the AA result. To prove correctness and completeness we must show that (i) the order and extent of the reported cells are both accurate, and (ii) the remaining part of the mixed arrangement cannot include cells with equal or smaller order. To see (i), if a cell is not inside any augmented half-space, it cannot be inside any subsumed half-space (hence, no subsumed half-space can affect its order), neither can it have an overlap with a subsumed half-space (thus, its extent is accurate too). Regarding (ii), the order of a cell in the mixed arrangement is a lower bound of its actual order in the complete arrangement[3]. Since all remaining cells in the mixed arrangement have order greater than those in T, it is safe to disregard them.

7. COMPLEXITY ANALYSIS

As demonstrated by Lemma 1 and Corollary 1 in Section 5, *MaxRank* can be solved by computing the complete arrangement of all incomparable records. However, the best known algorithms for this have a time and space complexity of $O(n^d)$ [1]. The crux of BA and AA is to only compute the smallest-order cells of the arrangement and to only implicitly maintain it. We first analyze their time and then their space complexity. For brevity, we denote by d_r the dimensionality of the reduced query space (i.e., $d_r = d - 1$).

BA builds an augmented Quad-tree that holds one half-space for every incomparable record, that is, $O(n)$ half-spaces. Assume that the Quad-tree leaves are all in the same level, and their total number is C. We focus the analysis on the leaf level, because internal nodes account for a trivial fraction of the Quad-tree. Insertion of a half-space alters the full and partial containment sets of $\frac{C}{2}$ leaves on the average[4]. Thus, Quad-tree construction takes $O(n\frac{C}{2})$ time. Leaf sorting takes another $O(C \log C)$. Yet the dominant cost is the computational geometric processing that follows.

We first analyze the cost of within-leaf processing for a single leaf. Consider the arrangement $\mathcal{A}$ of all incomparable records. $\mathcal{A}$ contains $O(n^{d_r})$ cells [1], a fraction $\frac{1}{C}$ of which overlap with the leaf. In the worst case, within-leaf processing needs to compute all these cells (i.e., the entire part of the arrangement that overlaps with the leaf's extent) in $O(\frac{1}{C}n^{d_r})$ time. A reasonable estimate is that the number of Quad-tree leaves processed by BA is proportional to $|T|$ (while the rest are pruned). Thus, the computational geometric part of BA takes $O(\frac{|T|}{C}n^{d_r})$ time in total.

Turning to AA, let n_a be the total number of incomparable records it processes. We expect that n_a is proportional to the cardinality of the skyline of incomparable records, i.e., n_a is $O(\frac{\log^{d-1} n}{d!})$ [9]. The rationale is that AA only considers the records that appear in the (progressively updated) skyline of incomparable records in some iteration.

Quad-tree construction takes $O(n_a\frac{C}{2})$ time. Since every iteration in AA processes additional records, the costs for leaf sorting and computational geometric processing are dominated by those in the final iteration, i.e., $O(C \log C)$ and $O(\frac{|T|}{C}n_a^{d_r})$, respectively. Note that C here refers to the Quad-tree of AA, which is smaller than in BA, because it holds only n_a half-spaces. Skyline computation and maintenance considers $O(n_a)$ records in total[5] in $O(n_a \log^{d-2} n_a)$ time [26].

Overall, we see analytically that our algorithms are expected to improve vastly over the straightforward application of an off-the-shelf algorithm for arrangement computation, and that AA is expected to largely outperform BA, because n_a is much smaller than the total number of incomparable records ($O(\frac{\log^{d-1} n}{d!}) \ll O(n)$).

The space complexity of BA is determined by the size of the Quad-tree, which in turn is dominated by the size of its leaf level. As explained previously, we expect that $O(\frac{1}{C}n^{d_r})$ arrangement cells overlap with a leaf l. These cells are induced by $O(\frac{n}{\sqrt[d]{C}})$ half-spaces [1], i.e., the partial overlap set P_l includes as many elements. On the other hand, for a half-space to be in the full containment set F_l, the half-space must partly overlap with at least one of the siblings of l (recall from Section 5.1 that if the half-space fully contains all sibling leaves, it is only stored in one of their ancestors). Hence, a half-space is included in a number of full containment sets that is proportional to the number of leaves that it partly overlaps. Thus, the size of F_l is expected to be proportional to P_l. That is, BA requires $O(\frac{n}{\sqrt[d]{C}})$ space per leaf, i.e., $O(C\frac{n}{\sqrt[d]{C}})$ in total. Similarly, AA requires $O(C\frac{n_a}{\sqrt[d]{C}})$ space, where $n_a \ll n$ as described earlier. Both space complexities largely improve on the $O(n^d)$ storage required for complete arrangement computation.

8. EMPIRICAL EVALUATION

In this section, we empirically evaluate the *MaxRank* algorithms and present measurements that provide insight into the nature of the problem. We use real and synthetic datasets. The real datasets are HOTEL, HOUSE, NBA, PITCH, and BAT. HOTEL (from *hotels-base.org*) contains 418,843 hotel records with four attributes, i.e., number of stars, price, number of rooms, and number of facilities. HOUSE (from *ipums.org*) contains 315,265 records; each holds six values that represent an American family's spendings in gas, electricity, water, heating, insurance, and property tax. NBA (from *basketballreference.com/stats*) contains 21,961 records of performance statistics for NBA players. PITCH and BAT hold 43,058 and 99,847 records of performance statistics for baseball pitchers and batters (both from *baseball1.com/statistics*).

[3] This is so because cell orders in the mixed arrangement are calculated based only on incomparable records that are explicitly reflected in the arrangement (i.e., ignoring the subsumed ones).

[4] This overestimate ignores the redundancy avoidance optimization in our Quad-tree (by keeping full containment sets in internal nodes) but simplifies the time analysis. Later on, in our space analysis, we quantify it accurately as $O(\frac{n}{\sqrt[d]{C}})$.

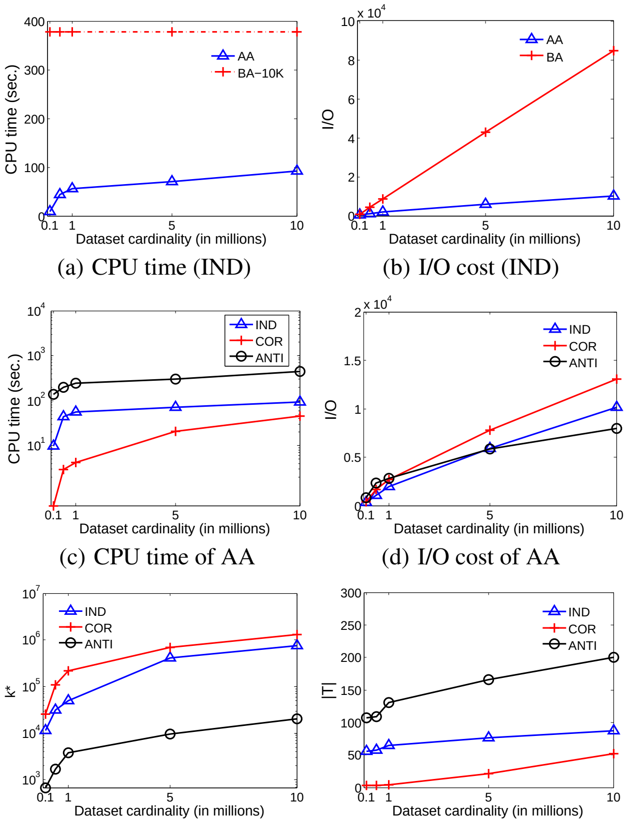
[5] BBS considers all records stored in leaves of the R*-tree on D that hold some of the n_a records that appear in the skyline [17].

We use three types of synthetic datasets, namely *Independent* (IND), *Correlated* (COR), and *Anti-correlated* (ANTI). These types of data are standard benchmarks for preference-based queries [5]. IND data are generated randomly and uniformly across the data space. In COR, if a record has a large value in an attribute, it tends to have large values in the other attributes too, and vice versa. In ANTI, if a record has a large value in an attribute, it is highly likely to have small values in the remaining ones.

We index each dataset by an R*-tree, and store data and index on the disk. The performance metrics are CPU time and I/O cost (measured in seconds and number of page accesses, respectively). Note that the CPU charts by themselves represent performance in the scenario where data and index reside in main memory. Table 2 lists the experiment parameters, their tested value ranges, and their default values (typed in boldface). In each experiment we vary one parameter, while setting the remaining ones to their defaults. Every presented measurement is the average over 40 queries for randomly selected focal records. All methods were implemented in C++, using the Qhull library from *qhull.org* for half-space intersection. Experiments were performed on a machine with Intel Xeon 2.67GHz CPU. The disk page size is set to 4KBytes.

Parameter	**Range of values**
Dataset cardinality, n	**100K**, 500K, 1M, 5M, 10M
Dimensionality, d	2, 3, **4**, 5, 6, 7, 8
iMaxRank parameter, τ	**0**, 1, 2, 3, 4, 5

Table 2: Experiment parameters and tested values



(a) CPU time (IND)

(b) I/O cost (IND)

(c) CPU time of AA

(d) I/O cost of AA

(e) Effect on k^*

(f) Effect on $|T|$

Figure 8: Effect of dataset cardinality n ($d = 4$)

In Figure 8, we investigate the effect of dataset cardinality n, as it varies from 100K to 10M records. The first pair of plots represents the performance of AA and BA on 4-dimensional IND data. BA fails to terminate within reasonable time for more than 10K records. Hence, in Figure 8(a), we only present its CPU time for the 10K dataset, drawn as a dashed line and labeled 'BA-10K'. However, its I/O cost in Figure 8(b) is accurate – this can be measured because BA performs all its data access before the heavy computation part.

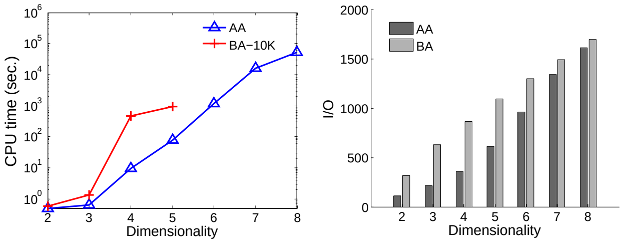
BA faces serious scalability issues, because every incomparable record needs to be accessed and reflected into the augmented Quad-tree. This implies a large I/O cost, but also an excessive amount of calculations for processing within the leaves of the Quad-tree. On the other hand, AA needs to access/process only a fraction of the incomparable records, and scales well with n in terms of both CPU time and I/O cost. The comparison between AA and BA verifies the effectiveness of the half-space subsumption methodology.

On another note, 94% to 97% of the computations in AA are spent on within-leaf processing. The percentage in BA (for the 10K dataset) is 71% because it reflects all incomparable records in its Quad-tree, thus spending a more significant amount of computations on Quad-tree construction. This difference is reflected in their space requirements too. AA takes up 3.8MBytes for $n = 100$K, and 13.3MBytes for $n = 10$M records; 85% and 87% of that space, respectively, is occupied by the augmented Quad-tree. In contrast, in BA the Quad-tree takes up 98.8% of the 212MBytes utilized.

In Figures 8(c) and 8(d), we vary n in the same range, but focus only on AA and its performance on the three benchmark distributions. AA scales gracefully with dataset cardinality for all three. To understand the performance differences among these distributions, in Figures 8(e) and 8(f), we plot the values of k^* and $|T|$ (i.e., the number of regions where order k^* is attained). Value k^* is the largest in COR, and it is achieved in relatively few regions of the query space. This means that the order of **p** is rather stable, which is reasonable because if a record **r** has a larger value than **p** in an attribute, its other values tend to also be larger, thus overshadowing **p** in terms of score w.r.t. to any (or almost any) query vector. Conversely, in ANTI we observe the smallest k^* values and the most regions where that order is achieved. The reason is that if a record **r** has a larger value than **p** in an attribute, it is highly likely to have a smaller value than **p** in another attribute. That is, the order of **p** fluctuates vastly for different query vectors, thus achieving a small order for some of them. The differences in CPU time among the three distributions are due to the great differences in $|T|$ – the larger the number of regions in *MaxRank* result, the higher the computational cost to identify them.

Regarding I/O cost in Figure 8(d), BA accesses all the dominators of **p** (to derive $|D^+|$) and all the incomparable records, resulting in linear increase with n. In AA the I/O cost is spent on accessing (i) all the dominators of **p** (for counting), and (ii) those of the incomparable records that are needed for AA processing. In ANTI, by definition, **p** has fewer dominators (i.e., factor (i) is smaller) and more incomparable records (leading to a larger factor (ii)) than in IND/COR. For small n, factor (ii) outweighs factor (i), thus the more I/Os in ANTI than in IND/COR. As n increases, however, the relative impact of the two factors is reversed, and the I/O cost in ANTI drops below IND/COR. This is because factor (i) increases linearly with n, whereas factor (ii) corresponds to value n_a which, as estimated in Section 7, increases sub-linearly to n.

In Figure 9, we study the effect of dimensionality d using IND data. For AA the measurements are for fixed cardinality $n = 100$K (the default). The CPU results for BA correspond to 10K datasets (due to the scalability limitations explained previously), on which



(a) CPU time (b) I/O cost

Figure 9: Effect of dimensionality d (n = 100K, IND)

d	k^*	$\|T\|$
2	39,199	1.6
3	13,663	6.5
4	11,316	55
5	4,393	436
6	3,495	11,493
7	522	74,280
8	214	149,732

Table 3: Effect of dimensionality on k^* and $|T|$

it still cannot terminate when $d > 5$. For $d = 2$ the measurements correspond to FCA (which we treat as the 2-dimensional version of BA) and to the 2-dimensional version of AA from Section 6.3. We take a closer look into $d = 2$ via a specialized experiment later.

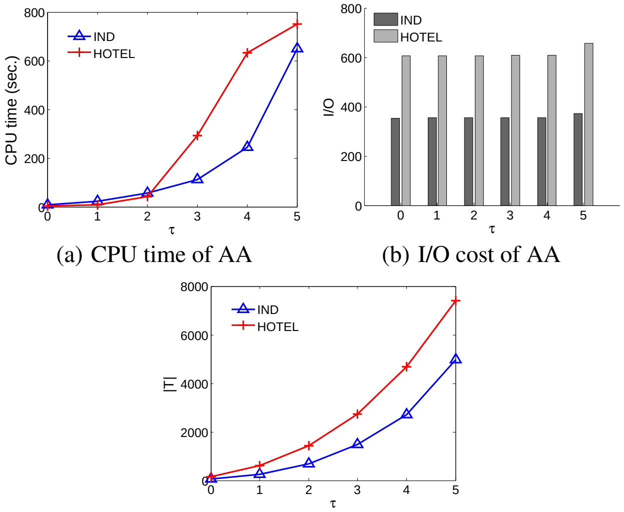
Other than the problematic performance of BA, our findings suggest that AA scales effectively with d. The costs of AA (CPU and I/O), and especially its CPU time, increase with d. The reason is the sharply growing number of regions $|T|$ where k^* is achieved. To substantiate this, in Table 3 we present the values of k^* and $|T|$ for the same experiment. As d grows, $|T|$ increases exponentially; conversely, k^* drops sharply. As we mentioned in Section 3 and empirically demonstrate in the Appendix, as d grows the score differences among the data records diminish (together with the usefulness of the top-k and *MaxRank* queries). This effect leads to numerous regions in the query space where the focal record attains a small order.

In Table 4, we present the performance of AA on the real datasets, together with the corresponding k^* and $|T|$ values (BA is not represented because it fails to scale to these datasets). We used all the attributes in HOTEL and HOUSE. In NBA we used the eight attributes that are suitable for rank-based processing as suggested in [10]. In PITCH and BAT we eliminated columns with many missing values and used, respectively, the eight and nine remaining attributes that are meaningful for scoring. The dimensionality and the cardinality of each dataset are listed in the first two columns of the table.

HOTEL and HOUSE have comparable cardinality, but the CPU time and the I/O cost are higher for HOUSE because it has more dimensions (i.e., six instead of four). NBA has the same dimensionality as PITCH, but half the size. Hence, the I/O cost is smaller for NBA. However, the CPU time in NBA is longer than in PITCH. The reason is that NBA is less correlated, because it includes statistics for players that play in different positions (e.g., point guards, power forwards, etc), whereas in PITCH all players are pitchers. This is also witnessed by $|T|$, which is more than double in NBA. Finally, BAT has the largest dimensionality and is quite voluminous. Nonetheless, AA terminates in 1005 seconds, demonstrating efficiency and efficacy.

Dataset	n	k^*	$\|T\|$	CPU time	I/O cost
HOTEL (4d)	418,843	19,403	179	4.45s	606.5
HOUSE (6d)	315,265	3,258	30,022	784.14s	1852.2
NBA (8d)	21,961	4,550	36,648	912.40s	245.4
PITCH (8d)	43,058	2,268	16,579	267.96s	527.9
BAT (9d)	99,847	20,530	18,096	1004.48s	1173.5

Table 4: Performance of AA on real datasets



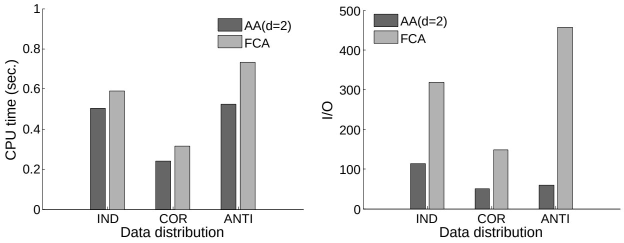
(a) CPU time of AA (b) I/O cost of AA

(c) $|T|$

Figure 10: *iMaxRank* processing, effect of τ

In Figure 10, we consider *iMaxRank*, using HOTEL and IND. For IND, $d = 4$ and n = 100K. We vary τ between 0 and 5 (where $\tau = 0$ corresponds to plain *MaxRank*). We only represent AA, because BA fails to terminate. The CPU cost increases significantly with τ, due to the sharply growing number of result regions $|T|$ (see Figure 10(c)). The I/O cost, however, increases only slightly, since the additional incomparable records that need to be processed for larger τ, often reside in disk pages fetched for $\tau = 0$ anyway.

In Figure 11, we look into the special case of $d = 2$. We compare FCA with the 2-dimensional AA, using IND, COR and ANTI datasets of cardinality n = 100K. Unlike AA, FCA accesses and processes all incomparable records, which is reflected in their performance difference in both CPU and I/O cost. The difference in CPU time is not as wide as in I/O cost, since AA needs to spend some calculations on half-line expansion and skyline updates.



(a) CPU time (b) I/O cost

Figure 11: FCA and AA in special case of $d = 2$ (n = 100K)

9. CONCLUSION

In this paper, we formulate the *MaxRank* query and develop a scalable framework for its processing. The problem is defined in the context of ranking queries, where the scoring function is a weighted sum of the data attributes. Given a dataset and a focal record in it, *MaxRank* computes the highest rank that the focal record may achieve w.r.t. any permissible weight setting. It also reports a description of all weight settings that yield that rank. An interesting direction for future work is to extend *MaxRank* processing to incomplete or uncertain data.

10. ACKNOWLEDGMENTS

This work was supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office. Jilian Zhang was also supported by China NSF grant 61363009.

11. REFERENCES

[1] P. K. Agarwal and M. Sharir. Arrangements and their applications. In *Handbook of Computational Geometry*, pages 49–119. Elsevier, 1998.

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

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

[4] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, pages 217–235, 1999.

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

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

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

[8] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.

[9] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004.

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

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

[12] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. DADA: a data cube for dominant relationship analysis. In *SIGMOD*, pages 659–670, 2006.

[13] J. Matousek. Reporting points in halfspaces. *Comput. Geom.*, 2:169–186, 1992.

[14] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *ICDE*, pages 356–365, 2008.

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

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

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

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

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

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

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

[22] Q. Wan, R. C. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.

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

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

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

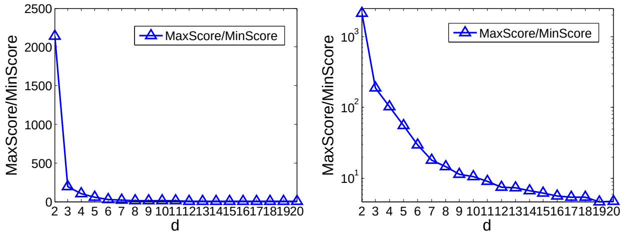
[26] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. K. H. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD*, pages 509–522, 2009.

[27] 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.

APPENDIX

In Section 3, we made a claim that the top-k query (let alone *MaxRank*) suffers from the dimensionality curse. In particular, that as the dimensionality grows, the highest score across the dataset approaches the lowest score. That is, the distinguishability between records (in terms of score) diminishes and, along with it, ranking by score loses usefulness in shortlisting the most preferable records.

We generated IND datasets with fixed cardinality $n = 100$K, for various d. In each of them, we identified the highest-scoring record and the lowest-scoring record (w.r.t. a randomly chosen query vector $\mathbf{q}$), and recorded the ratio of their scores. In Figure 12, we plot this ratio versus d in linear and in logarithmic scale, for clarity. The results validate our claim. We note that the trends resemble closely those in [4], where the nearest neighbor query is shown to suffer a similar loss of meaning with d (the distances of the nearest and of the furthest data record to the query point converge).



(a) Ratio in linear scale

(b) Ratio in log. scale

Figure 12: Effect of d on MaxScore/MinScore ratio