Reverse k-Ranks Query

Zhao Zhang Cheqing Jin * Qiangqiang Kang
Institute for Data Science and Engineering, Software Engineering Institute,
East China Normal University, Shanghai, China
{zhzhang, cqjin}@sei.ecnu.edu.cn, qqkang@ecnu.cn

ABSTRACT

Finding matching customers for a given product based on individual user's preference is critical for many applications, especially in e-commerce. Recently, the reverse top-k query is proposed to return a number of customers who regard a given product as one of the k most favorite products based on a linear model. Although a few "hot" products can be returned to some customers via reverse top-k query, a large proportion of products (over 90%, as our example illustrates, see Figure 2) cannot find any matching customers.

Inspired by this observation, we propose a new kind of query (R-kRanks) which finds for a given product, the top-k customers whose rank for the product is highest among all customers, to ensure 100% coverage for any given product, no matter it is *hot* or *niche*. Not limited to e-commerce, the concept of *customer-product* can be extended to a wider range of applications, such as dating and job-hunting. Unfortunately, existing approaches for reverse top-k query cannot be used to handle R-kRanks conveniently due to infeasibility of getting enough elements for the query result. Hence, we propose three novel approaches to efficiently process R-kRanks query, including one tree-based method and two batch-pruning-based methods. Analysis of theoretical and experimental results on real and synthetic data sets illustrates the efficacy of the proposed methods.

Keywords

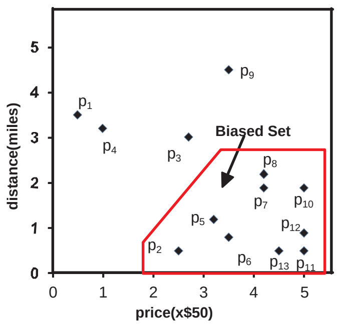
Ranking query, Reverse ranking, R-kRanks

1. INTRODUCTION

As a fundamental operator in database management, rank-aware query has been studied extensively in recent years. In general, the ranking query returns top k tuples with minimal ranking scores (or maximal ranking scores under different contexts) by employing a user-defined scoring function [7, 9, 12, 21, 26, 26]. There exist many kinds of scoring functions for rank-aware queries, among which the linear model is one of the most widely adopted models [9, 21, 26]. In the linear model, each object in the database D is described as a d-dimensional point p, and the preference of a user is described as a weight vector w. Each entry of w, $w^{(i)}$, is non-negative, and the sum of all entries is equal to 1, i.e, $\sum_{i=1}^{d} w^{(i)} = 1$. For a user with preference w, the ranking score is defined as $\sum_{i=1}^{d} p^{(i)} \cdot w^{(i)}$ where $p^{(i)}$ represents the ith entry of p. A linear top-k query aims at returning k objects in D with smallest scores. Typical examples cover a wide range of applications, including multimedia search, streaming data management, Web data analysis, etc.

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 7, No. 10
Copyright 2014 VLDB Endowment 2150-8097/14/06.



User preferences & Top-1

user	w[price]	w[distance]	Top-1
Alice	0	1	p_2
Jim	1	0	p_1
Terry	0.8	0.2	p_1
Mary	0.2	0.8	p_2
Joe	0.7	0.3	p_1

Reverse Top-1

hotel	RTop-1
p_1	Jim, Joe, Terry
p_2	Alice, Mary
p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13},	NULL

Figure 1: An example of ranking queries

The top-k query under linear model has been studied mainly from the perspective of users, aiming at finding a set of products to match their preferences. Recently, another rank-aware query has been studied from the perspective of manufacturers. Assume a manufacturer has a lot of products for sale and the manager wants to know who will be interested in their products. By employing linear ranking model, *reverse top-k* query has been proposed to return all users who treat the given product as one of their top-k products [21]. See the example below.

EXAMPLE 1. *Figure 1 illustrates a simple example of five customers and thirteen hotels in a city. Each hotel has two attributes: price per night and distance to the beach. A rational customer may prefer a hotel which is cheaper and closer to the beach. We use a weight vector to describe each customer's preference. Jim, Terry and Joe are sensitive to the price, but Alice and Mary care more about the distance to the beach. For Alice, the ranking score of p_2 (2.5, 0.5) is computed as: $0 \times 2.5 + 1 \times 0.5 = 0.5$, which is the minimum compared with other points. Hence, p_2 is the best choice* *(top-1) for Alice. Similarly, p_2 is also the best choice for Mary, while p_1 is the best choice for the remaining customers. The result sets of reverse top-1 query for p_2 and p_1 are $\{Alice, Mary\}$ and $\{Jim, Joe, Terry\}$ respectively. Moreover, it will return an empty set for the remaining eleven hotels. See the tables in Figure 1 for reference.*

Although capable of finding all users who are interested in the given product, reverse top-k query cannot deal with niche products well. In other words, we can find quite a lot of customers to match popular products, while there may exist no one to match the less popular (or niche) products since these products are not in any customer's top-k product list. We conduct a simple experiment for the illustration purpose. Consider a case that each data set (*hotel* or *preference*) contains 5,000 tuples with attribute values assigned uniformly in the domain. Figure 2 shows the different proportions of hotels that are recommended to at least one customer when k increases from 10 to 100. Even when $k = 100$, over 90% hotels have never been recommended.

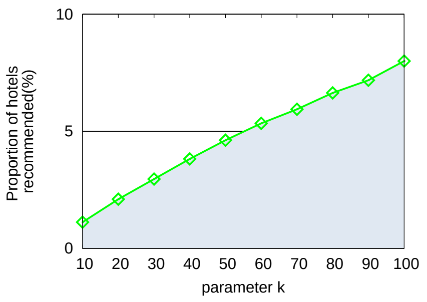


Figure 2: Proportion of the hotels with non-empty result set after executing reverse top-k query

1.1 R-kRanks and Its Variant

As illustrated above, reverse top-k query cannot ensure 100% coverage for any given product. However, it is critical to return a number of customers (not empty) to match any given product in many situations. For example, the manufacturer may expect to find potential customers for all products. Other examples include job-hunting, dating, and so on. Job-hunting applications have personal information of all job seekers, and require employers to build the preference vectors. Thus, it will try to find some candidate employers for any job seeker. In the dating sites, each woman or man describes her or his personal information and the expectation for the opposite gender, so that the web site can find some candidates to match her or him best.

Although irrelative to products and customers, the above applications can also be modeled by utilizing the concept of *customer-product*. For example, in the job-hunting applications, job seekers can be treated as *products*, while employers can be treated as *customers*. Hence, we continue to use this concept in the following context. Now, the issue turns to how to describe a set of customers who can best match a given product. We propose a new query definition that finds k customers with smallest $rank(w, q)$ values, where $rank(w, q)$ denotes the number of products ranking higher than q for w. We name this query *reverse k-Ranks* (R-kRanks in abbr.)

A variant of this new query is to find k customers with smallest ranking scores to the given product q, i.e, returning k customers w with smallest $\sum q^{(i)} \cdot w^{(i)}$. We name the former *reverse k-Ranks* (R-kRanks in abbr.), and the latter *reverse k-Scores* (R-kScores in abbr.).

Table 1: R-kRanks vs. R-kScores (assuming $q = p_8$)

(a) The ranking scores and ranks

	Alice	Jim	Terry	Mary	Joe
Ranking score	2.2	4.2	3.8	2.6	3.6
Rank	8	3	5	8	5

(b) Query result

	R-kRanks	R-kScores
$k = 1$	Jim	Alice
$k = 2$	Jim, Joe	Alice, Mary
$k = 3$	Jim, Joe, Terry	Alice, Mary, Joe

Although both query definitions return k customers for any query product, the result set may differ a lot, especially for a *biased* product data set. It's not our focus to debate which one is definitely better than the other one. So we just use a simple example to illustrate the difference. Assume p_8 is a query point. Table 1 compares two kinds of queries based on the biased data set in Figure 1 (w/o p_1, p_3, p_4 and p_9). For example, for Jim, only three objects in the biased set (p_2, p_5 and p_6) rank higher than p_3. The query results differ a lot under different k values. To some extent, R-kScores is less reasonable in this case since it treats Alice as the best customer to match p_8, whereas p_8 is the last choice for Alice (see Table 1(a)). More comparisons between R-kRanks and R-kScores appear in section 8.3.

Moreover, from a technical point of view, R-kScores can be solved directly in a way similar to traditional top-k query under linear model by treating each customer as a *product*, and the given product as a user's preference without normalization [1]. Hence, we only consider R-kRanks in this paper.

1.2 Challenges

It is challenging to handle R-kRanks query, since existing methods for reverse top-k query cannot deal with R-kRanks query straightforwardly. Representative work for bichromatic reverse top-k query includes threshold-based algorithm (RTA) [21], grid-based algorithm (GRTA) [21], and branch-and-bound algorithm (BBR) [25]. With the help of a small buffer (with a size of k), RTA visits all users one by one. For each user, if the query product's rank is lower than any product in the buffer, the query product is definitely out of the user's top-k list; otherwise, it continues to compute the exact rank of query product and updates the buffer. GRTA materializes some reserve top-k views in memory for efficient processing after dividing the whole product space into a series of cells. BBR, the most recent work, uses branch-and-bound idea without accessing each user's individual preference [25].

The intrinsic reason why the above methods cannot handle R-kRanks query is that these methods only care about the customer-product pairs (w, p) such that p is in w's top-k product list. Unfortunately, processing R-kRanks will consider different set of customer-product pairs, where p is unnecessary to be in w's top-k list. For example, as illustrated in Figure 2, more than 90% products are out of the top-k

[1]This method is available if each entry of the product is non-negative.

product list of any customer. Simply enlarging the value of k is insufficient for any product.

To some extent, the semantic of reverse k-nearest neighbor (RkNN) query is similar to R-kRanks query. It returns a number of points that treat the given object as one of the k nearest neighbors. Ranked reverse nearest neighbor(RRNN) is another similar query, which returns a number of points that are close to the query point according to the rank [14]. However, both of them use Euclidean distance to describe the metric *dist* between two points, while R-kRanks uses inner product to describe the matching degree between a query point (a product vector) and a customer vector. In other words, for a given product, if RkNN and RRNN are used to find potential users, the preference of each user must be represented by his or her ideal product. For example, in Figure 1, Mary's preference is described as a vector (0.2, 0.8), so that p_2 is her favorite hotel. Now, consider another way to represent the preference described in RkNN and RRNN. Assume Mary has visited another city, and she was satisfied with a hotel with the distance and price as same as p_5. To some extent, her preference can be represented as p_5 since then. However, in the new city like Figure 1, the hotel p_2 is significantly better than p_5 since p_2 is cheaper, and closer to the beach than p_5. Hence, the weight-represented preference is more insensitive to the environment.

1.3 Contribution

We have made the following contributions in this paper.

- We define a new ranking query (R-kRanks) that returns k customers to match a product. In comparison, although traditional reverse top-k query works well for "hot" products, it may return an empty set for non-popular query products (Figure 2).

- We propose novel solutions to handle R-kRanks query efficiently since existing approaches cannot be applied directly. Basically, R-kRanks can be processed by computing the ranking score of every product-customer pair. Except for simplicity, this method suffers from a high time-complexity of $O(m \cdot n)$, where m and n are the number of customers and products respectively. We develop three novel techniques to significantly reduce the computation cost in this work. The first method (TPA) is based on a spatial tree, the second one (BPA) evaluates the weights group by group, and the last one (MPA) further reduces the time consumption by maintaining an additional array for marking.

- We conduct extensive experiments over real-life and synthetic data sets to evaluate the efficacy of the proposed methods.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 defines the new query formally. Sections 4-6 introduce three novel approaches. Afterwards, we provide some extensive discussions in Section 7. Experimental reports are given in Section 8. Finally, we conclude the paper briefly in the last section.

2. RELATED WORK

Rank-aware query and its variants have been widely used in many applications. Nowadays, it has many variants. The most general one is ranking query, which returns a number of points with minimal ranking scores by using a ranking function. Recently, some researchers also focus on reverse ranking query that returns all customers who like a given product very much. In comparison, ranking query mainly concerns from the perspective of the user, while reverse query mainly concerns in terms of product manufacturers.

Ranking Query Ranking query (top-k query) has been studied extensively for decades [10]. Fagin's algorithm (FA) [6] and Threshold Algorithm (TA) [7] are two of the most famous algorithms. Given a relation with multiple attributes, these approaches first sort all tuples by each attribute in parallel so that only a small proportion of tuples are required to be fetched. In contrast, TA is more time-efficient than FA [7]. The research on ranking query has also been extended to various scenarios, such as distributed environment [1], streaming applications [19], and uncertain data situations [11]. One family of algorithms for ranking query is based on preprocessing, which stores some information after processing all tuples off-line. Such information can help to compute the result more efficiently once a new query arrives. Chang et al. proposed *Onion* technique to index convex hulls for linear optimization queries [3]. *Prefer* method proposed by Hristidis et al. materializes views of top-k result sets [9]. Xin et al. proposed the *robust index* method with more elaborate consideration on the minimal possible rank of each tuple in the database [26].

Reverse NN and reverse skyline Reverse NN (RNN) returns a set of query points that treat a given query point as the nearest neighbor [13]. There are two kinds of approaches dealing with RNN query: precomputation-based approaches and dynamic approaches [14]. The former store some useful information in the system after pre-computing results of kNN query for each point, so that the executing efficiency of a future query can be improved [27]. The latter often use an index structure to accelerate query-processing, such as R-tree [20]. In [14], Lee et al. studied Ranked Reverse Nearest Neighbor (RRNN). Given a query point q and a number t, RRNN returns t points close to q according to the rank. The bichromatic RRNN is similar to R-kRanks but works in the different context. Their work uses *Euclidean distance* to describe the *dist* metric between two points, but we use *inner product* to describe the matching degree between a query point (a product vector) and a customer vector. Hence, the approaches in [14] cannot deal with the issues in this paper. Skyline query is a typical multi-objective optimization issue [2]. Given a query point (a product), *reverse* skyline query returns a number of customers who like the given product most based on the dominance relationship among the products [5]. Lian and Chen extended reverse skyline query to the probabilistic data management [17]. However, in these literatures, the preference of each user is described as a data point representing the desirable product. But in our work, the preference is described as a weight vector.

Reverse ranking query Reverse ranking query aims at evaluating the rank of a query point based on a preference function [15, 16, 18]. Both reverse top-k and R-kRanks queries concern the rank of a product under different preference functions.

The work most related to ours is the reverse top-k query which aims at finding all customers who treat a given product as one of their top-k favorite products [4, 8, 21–25, 28]. Vlachou et al. proposed one solution for monochromatic reverse top-k query and two solutions for bichromatic reverse top-k query, namely RTA and GRTA [21]. Afterwards, they also studied how to handle monochromatic reverse top-k query in d-dimensional space and presented more experiments for illustration [22]. Most recently, Vlachou et al. further improved RTA to process reverse top-k queries by branch-and-bound algorithm without accessing each user's individual preferences [25]. Literature [4] indexed dataset based on a critical k-polygon for answering monochromatic reverse top-k queries in two dimensions. Reverse top-k query can be used to find the most influential data objects that are preferred by more customers [24]. In [23], some researchers considered how to monitor reverse top-k queries over mobile devices. Ge et.al proposed an approach for batch evaluation of all top-k queries by using block indexed nested loops and view-based algorithm [8]. Literature [28] presented a dynamic index that supports the reverse top-k query. In particular, a scalable solution for processing many continuous top-k queries was developed combining this index with another one for top-k queries. However, none of existing solutions for reverse ranking query can be applied to handle R-kRanks query directly due to different query semantics.

3. PROBLEM STATEMENT

Let D denote a set containing n points and each point is described as a d-dimensional vector p_i. $\forall j \in [1, d]$, $p_i^{(j)}$ denotes the value of the j-th attribute of p_i. Let W denote a set containing the weights of the preferences for m customers and each weight is described as a d-dimensional point w_i. $\forall j \in [1, d]$, $w_i^{(j)}$ denotes the value of the j-th dimension of w_i. Similarly, $\forall i \forall j (w_i^{(j)} \geq 0)$. Assume the set W has been preprocessed so that the sum of the weights of each customer is equal to 1, i.e, $\forall i (\sum_{j=1}^{d} w_i^{(j)} = 1)$. Note that this preference weight model has been widely adopted in many literatures [3, 4, 8, 9, 21–26, 28].

For any customer with a preference weight w, the scoring function $f(w, p)$ computes the ranking score of a point p. It is defined as the inner product of w and p, i.e, $f(w, p) = \langle w, p\rangle = \sum_{i=1}^{d} w^{(i)} p^{(i)}$. Without loss of generality, we assume that a customer prefers a point with smaller ranking score in this study [2]. Since both top-k and reverse top-k queries have strong connections with our work, we first review these two queries for reference and then present our new query.

Definition 1 (Top-k query, $TP(k, w)$). *Given a point set D, a user preference w and a positive integer k, the top-k query returns a set S of points, $S \subseteq D$, $|S| = k$, and $\forall p_i \in S, \forall p_j \in (D - S)$, we have: $f(w, p_i) \leq f(w, p_j)$.*

Definition 2 (Reverse top-k query [21]). *Given a point set D, a weight set W, a positive integer k, and a specified query point q, reverse top-k query returns a set S of weights, $S \subseteq W$. For each $w \in S, q \in TP(k, w)$; but for each $w \notin S$, $q \notin TP(k, w)$.*

As we mentioned before, reverse top-k query will return a non-empty set only for a small fraction of products. Consequently, we propose a new query type, namely reverse k-Ranks (R-kRanks), and its variant, reverse k-Scores (R-kScores). Given a query point q, R-kScores query returns top k customers (say, w) with the smallest $f(w, q)$, while R-kRanks query returns top k customers (say, w) with the smallest $\mathsf{rank}(w, q)$ values. Here, the subroutine $\mathsf{rank}(w, q)$ computes the number of products with smaller ranking score than q for a given customer w.

[2]For the case where the users may prefer great values in some attributes, we can replace values of such attributes with their additive inverse to adapt to this model.

Table 2: Important symbols

Symbol	Description
D, W	The product set, the preference weight set
k	A parameter in R-kRanks query
q	The query point
w	The preference weight of a customer
R	R-tree constructed by data points
r	An MBR in R-tree
$r.U, r.L$	The top right and bottom left corners of r
$\mathcal{H}$	Histogram of weight vectors
H_i	A bucket in the weight histogram
$H_i.U, H_i.L$	The top right and bottom left corners of H_i
$\mathcal{P}(H_i, q)$	The hyperplane family for H_i and q
g	The affected dimension of PoN (or NeN)
u	The split dimension of a weight bucket H_i
c	Each bucket H_i has a width of $\frac{1}{c}$

Definition 3 ($\text{rank}(w, q)$). *Given a point set D, a weight vector w, and a query point q, the rank of q for w is $\mathsf{rank}(w, q) = |S|$, where $|S|$ is the cardinality of S, a subset of D. $\forall p_i \in S$, we have $f(w, p_i) < f(w, q)$; $\forall p_j \in (D - S)$, we have $f(w, p_j) \geq f(w, q)$.*

Definition 4 (Reverse k-Ranks query, R-kRanks). *Given a point set D, a weight set W, a positive integer k, and a query point q, R-kRanks query returns a set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $\mathsf{rank}(w_i, q) \leq \mathsf{rank}(w_j, q)$ holds.*

Definition 5 (Reverse k-Scores query, R-kScores). *Given a point set D, a weight set W, a positive integer k, and a query point q, R-kScores query returns a set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $f(w_i, p) \leq f(w_j, p)$ holds.*

We focus on R-kRanks query in this work due to the limitation of R-kScores. We develop three novel solutions to process R-kRanks query in Section 4-6, including TPA, BPA and MPA. Moreover, it is not hard to devise a straightforward solution by exploring the whole space. Let q denote the query point. For any weight w in W and any product p in D, we compute $\mathsf{rank}(w, p)$ one by one, and keep k weights with the smallest $\mathsf{rank}(w, q)$ for query point q. The time complexity is $O(m \cdot n)$ where m and n are the number of customers and products respectively. The space complexity is $O(k)$. This method also acts as the baseline approach (Naive Approach, NA in abbr.) in Section 8.

Important symbols are summarized in Table 2.

4. TREE-BASED PRUNING APPROACH (TPA)

The naive approach (NA) needs to evaluate every product-customer pair, which will consume huge amounts of computation resources in datasets with a large number of customers and products. Fortunately, parts of computations

Algorithm 1: Tree-based Pruning Approach (TPA) $(R, W, \vec{q}, k)$

```
1  Let $Q$ and $Q'$ denote two queues;
2  Let $A$ denote an array to record the rank of each weight;
3  $minRank \leftarrow$ the number of objects in $R$;
4  foreach $w_i$ in $W$ do
5      Clear $Q$ and $Q'$;   enqueue($Q$, $R$.root);
6      while $((r = dequeue(Q)) \neq \emptyset)$ do
7          if $(f(w_i, r.U) < f(w_i, q))$ then
8              $A[w_i] \leftarrow A[w_i] + |r|$ ;
9          else if $(f(w_i, q) < f(w_i, r.L))$ then continue ;
10         else
11             if ($r$ is a leaf node) then enqueue($r, Q'$);
12             else $\forall r'$ ($r'$ is $r$'s child), enqueue($r', Q$) ;
13     if $(A[w_i] \leq minRank)$ then
14         foreach $r$ in the queue $Q'$ do
15             foreach point $p$ in $r$ do
16                 if $f(w_i, p) < f(w_i, q)$ then
17                     $A[w_i] \leftarrow A[w_i] + 1$;
18         $minRank \leftarrow$ the $k$th minimal value in $A$;
19 return $k$ smallest entries in $A$;
```

can be avoided with the help of an R-tree. At first, we review the *dominance* relationship between two points, which has been adopted in many literatures, including skyline [2] and reverse top-k query processing [21].

DEFINITION 6 (DOMINANCE). *We say point p_1 dominates p_2 (denoted as $p_1 \prec p_2$) if and only if the following two conditions hold: (i)$\forall l$, $p_1^{(l)} \leq p_2^{(l)}$; (ii) $\exists j$, $p_1^{(j)} < p_2^{(j)}$.*

Let $f(w, p)$ denote a scoring function that returns inner product of w and p. Since $\forall i, w_i \geq 0$, we have $f(w, p_1) < f(w, p_2)$ when $p_1 \prec p_2$. We build an R-tree to index all products. Let r denote a minimal bounding rectangle (MBR) in R-tree. Let $r.L$ and $r.U$ denote the bottom left and top right points of r respectively. We have two facts below.

FACT 1. *Given a query point q, a weight vector w_i, and an MBR r in R-tree, if $f(w_i, q) < f(w_i, r.L)$, $\forall p \in r$, we have $f(w_i, q) < f(w_i, p)$.*

FACT 2. *Given a query point q, a weight vector w_i, and an MBR r in R-tree, if $f(w_i, r.U) < f(w_i, q)$, $\forall p \in r$, we have $f(w_i, p) < f(w_i, q)$.*

Algorithm 1 illustrates the main steps of our tree-based pruning approach (TPA). Assume all products have been indexed by an R-tree R off-line. Initially, two queues, Q and Q', are created. During processing, it computes the ranking score for each preference weight vector w_i in W by traversing R-tree. For each node r in the tree, the algorithm computes the ranking scores of $r.L$ and $r.U$ with respect to w_i. According to Fact 1 and 2, the nodes satisfying $f(w_i, q) < f(w_i, r.L)$ or $f(w_i, r.U) < f(w_i, q)$ can be pruned safely. Otherwise, all child nodes of r (if r is a non-leaf node) are appended to Q for further processing. The queue Q' keeps track of all unpruned leaf nodes (at lines 10-12). The subroutine enqueue appends an entry to the tail of a queue and dequeue returns the head entry in the queue (will remove this entry afterward).

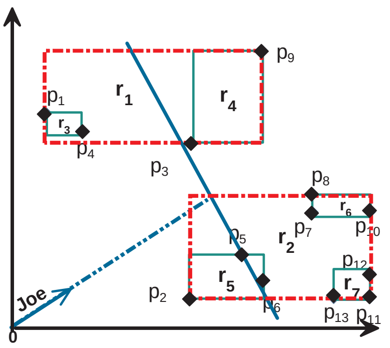


Figure 3: An example of TPA algorithm

Let $minRank$ denote a global variable to describe the k-th minimal entry in the array A. In other words, at least k users treat q as one of his or her top-$minRank$ favorite objects till now. If the new customer w_i does not treat q as one of his or her top-$minRank$ objects, it is unnecessary to visit Q' any more. Otherwise, we continue to compute its ranking score by exploring all entries in Q' (at lines 14-17). Finally, the algorithm returns k smallest entries in A.

Figure 3 illustrates how TPA works in 2-D space, where p_5 is the query point. The solid line crosses p_5 and is perpendicular to Joe(0.7,0.3), so that the ranking score of any product in this line is identical to p_5 for Joe. TPA then visits MBRs in the R-tree. For example, r_3, r_4, r_6 and r_7 can be pruned by checking two corner points. But r_5 needs further processing since it is across the solid line. Hence, the rank of p_5 is at least 2 ($= |r_3|$). Subsequently, the algorithm continues to traverse Q' ($Q' = \{r_5\}$). Finally, $\mathsf{rank}(p_5) = 3$. TPA processes other customers similarly.

TPA prunes computations from the perspective of data points. Compared with NA, TPA reduces time complexity from $O(m \cdot n)$ to $O(m_r \cdot n)$ at the cost of R-tree's construction for D off-line, where $m = |D|$, $n = |W|$ and $m_r = |R|$. Generally, $m_r \ll m$.

5. BATCH PRUNING APPROACH (BPA)

The advantage of TPA is that similar products can be evaluated together after being represented by MBRs. However, the performance is limited when encountered with a large weight data set since all preference weights need to be evaluated one by one, which inspires us to eliminate redundant computation after grouping similar weights.

We use a d-dimensional histogram to group all weights. Each dimension is divided into c equi-width intervals, denoted as $[0, \frac{1}{c}], [\frac{1}{c}, \frac{2}{c}], \cdots, [\frac{c-1}{c}, 1]$, resulting in c^d buckets. However, some buckets (called *invalid* buckets) contain no preference weight, due to the constraint: $\sum_{l=1}^{d} w^{(l)} = 1$. It is only necessary to consider the remaining *valid* buckets. Let $H_i.L$ and $H_i.U$ denote the bottom left and top right corners of an arbitrary bucket H_i. A bucket H_i is invalid if either $\sum_{l=1}^{d} H_i.L^{(l)} \geq 1$ or $\sum_{l=1}^{d} H_i.U^{(l)} \leq 1$ holds [3]. For simplicity, all the valid buckets are denoted as $H_1, H_2, \cdots$.

5.1 Four Kinds of Relationships

[3]In fact, when $\sum_{l=1}^{d} H_i.L^{(l)} = 1$ (or $\sum_{l=1}^{d} H_i.U^{(l)} = 1$), this bucket still contains one valid preference weight, $H_i.L$ (or $H_i.U$). The reason we treat the bucket invalid is because the special point will also appear in other valid bucket.

We then study the relationship between a bucket H_i and an MBR r given a query point q. At first, we review the *hyperplane* concept in d-dimensional space. Given a weight vector w and a point q, there exists one (d-1)-dimensional hyperplane, $P(w, q)$, which (i) is perpendicular to w and (ii) contains q. The ranking score of each point in the hyperplane is equal to $f(w, q)$. Furthermore, given a query point q and a weight bucket H_i, we can derive a *hyperplane family*, $\mathcal{P}(H_i, q)$, which contains all hyperplanes based on all valid weight vectors in H_i. We summarize four kinds of relationships between an MBR r and a hyperplane family $\mathcal{P}(H_i, q)$ below. For example, belowP(r, H_i)=TRUE (or aboveP(r, H_i)=TRUE) means the MBR r is entirely *below* (or *above*) the hyperplane family $\mathcal{P}(H_i, q)$. If withinP(r, H_i)=TRUE, it means for any point p in r, we can always find a hyperplane in $\mathcal{P}(H_i, q)$ that contains p and q simultaneously. Otherwise, acrossP(r, H_i) returns TRUE.

- belowP(r, H_i). It returns TRUE if and only if $\forall p \in r$: $[\forall w \in H_i : f(w, p) < f(w, q)]$ always holds.
- aboveP(r, H_i). It returns TRUE if and only if $\forall p \in r$: $[\forall w \in H_i : f(w, p) \geq f(w, q)]$ always holds.
- withinP(r, H_i). It returns TRUE if and only if $\forall p \in r$: $[\exists w \in H_i : f(w, p) \leq f(w, q) \wedge \exists w' \in H_i : f(w', q) \leq f(w', p)]$ always holds.
- acrossP(r, H_i). It returns TRUE if and only if none of the above three operators returns TRUE;

We further explore the semantics of these operators. The lower bound of q's rank, $rank(w, q)$, is the cardinality of the union of all MBRs satisfying belowP relationship. Symmetrically, the upper bound of q's rank is the cardinality of the union of all MBRs satisfying aboveP relationship. An MBR with acrossP relationship only has potential pruning capability, since parts of its descendant nodes may be below or above the hyperplane family. But if withinP(r, H_i)=TRUE, we need to visit all points in r if necessary.

Figure 4 illustrates a 2-D case. Let $c = 5$. There are $5^2 = 25$ weight buckets, but only five of them are valid: ($[0, 0.2]$, $[0.8, 1]$), ($[0.2, 0.4]$, $[0.6, 0.8]$), ($[0.4, 0.6]$, $[0.4, 0.6]$), ($[0.6, 0.8]$, $[0.2, 0.4]$), and ($[0.8, 1]$, $[0, 0.2]$). Assuming p_3 is the query point, we derive five hyperplane families: $\mathcal{P}(H_1, p_3)$, $\cdots$, $\mathcal{P}(H_5, p_3)$. Hence, we have: aboveP$(r_6, H_4) = TRUE$, withinP$(r_7, H_3) = TRUE$, accossP$(r_5, H_4) = TRUE$, and belowP$(r_3, H_5) = TRUE$.

5.2 Implementations of Relationships

According to Definition 6, any other point in r dominates $r.U$, but is dominated by $r.L$. Let $UB(H_i)$ denote the maximal ranking score difference between $r.U$ and q for any weight w in H_i.

$$UB(H_i) \triangleq \max_{\forall w \in H_i} \left(f(w, r.U) - f(w, q)\right)$$
$$= \max_{\forall w \in H_i} \left(\sum_{l=1}^{d} w^{(l)} \cdot \left(r.U^{(l)} - q^{(l)}\right) \right) \qquad (1)$$

If belowP$(r, H_i) = TRUE$ holds, then $UB(H_i) < 0$.

Lemma 1. *Let H_i denote a valid weight bucket. To compute $UB(H_i)$, there exists one dimension u $(1 \leq u \leq d)$ such that: (i) $\forall l, 1 \leq l < u : w^{(l)} = H_i.U^{(l)}$, (ii) $w^{(u)} \in (H_i.L^{(u)}, H_i.U^{(u)}]$, and (iii) $\forall l, u+1 \leq l \leq d : w^{(l)} = H_i.L^{(l)}$.*

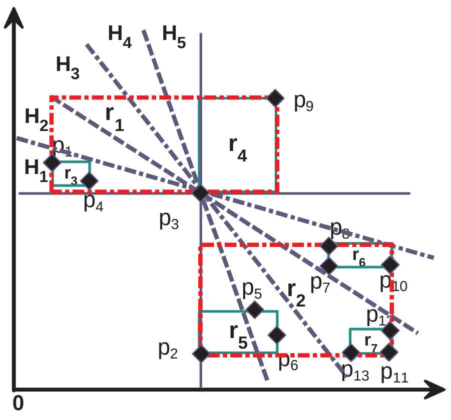


Figure 4: Illustrations of hyperplane families

Proof. The correctness comes from the following steps. First, for each dimension l, $1 \leq l \leq d$, we assign: $w^{(l)} \leftarrow H_i.L^{(l)}$ to conform to the basic requirement. Second, we reassign: $w^{(l)} \leftarrow H_i.U^{(l)}$ for the dimensions with greatest $e^{(l)}$ coefficients as many as possible with the constraint: $\sum_{l=1}^{d} w^{(l)} < 1$. Finally, assuming $u-1$ dimensions have been reassigned in the above step, and $u \leq d$, we reassign: $w^{(u)} \leftarrow 1 - \sum_{l=1}^{u-1} H_i.U^{(l)} - \sum_{l=u+1}^{d} H_i.L^{(l)}$ to make $\sum_{l=1}^{d} w^{(l)} = 1$ hold. Note that $w^{(u)}$ is in $(H_i.L^{(u)}, H_i.U^{(u)}]$. $\square$

For simplicity, we assume all coefficients of w (say, $\{r.U^{(l)} - q^{(l)}\}$) are sorted in descending order and let $e^{(l)} = \left(r.U^{(l)} - q^{(l)}\right)$ for short. $UB(H_i)$ is computed below.

$$UB(H_i) = \sum_{l=1}^{u-1} \left(H_i.U^{(l)} \cdot e^{(l)}\right) + \sum_{l=u+1}^{d} \left(H_i.L^{(l)} \cdot e^{(l)}\right)$$
$$+ \left(1 - \sum_{l=1}^{u-1} H_i.U^{(l)} - \sum_{l=u+1}^{d} H_i.L^{(l)}\right) \cdot e^{(u)} \quad (2)$$

where

$$u = \text{argmax}_j \left(\sum_{l=1}^{j} H_i.U^{(l)} \cdot e^{(l)} + \sum_{l=j+1}^{d} H_i.L^{(l)} \cdot e^{(l)} < 1 \right) + 1$$

Symmetrically, let $LB(H_i)$ denote the minimal ranking score difference between $r.L$ and q for any weight w in H_i.

$$LB(H_i) \triangleq \min_{\forall w \in H_i} \left(f(w, r.L) - f(w, q)\right)$$
$$= \min_{\forall w \in H_i} \left(\sum_{l=1}^{d} w^{(l)} \cdot \left(r.L^{(l)} - q^{(l)}\right) \right) \qquad (3)$$

If aboveP$(r, H_i) = TRUE$ holds, then $LB(H_i) \geq 0$.

Lemma 2. *Let H_i denote a valid weight bucket. To compute $LB(H_i)$, there exists one dimension u, such that: (i) $\forall l, 1 \leq l < u$, $w^{(l)} = H_i.L^{(l)}$, (ii) $w^{(u)} \in (H_i.L^{(u)}, H_i.U^{(u)}]$, and (iii) $\forall l, u+1 \leq l \leq d$, $w^{(l)} = H_i.U^{(l)}$.*

We omit the proof due to space limitations.

$$LB(H_i) = \sum_{l=1}^{u-1} \left(H_i.L^{(l)} \cdot e^{(l)}\right) + \sum_{l=u+1}^{d} \left(H_i.U^{(l)} \cdot e^{(l)}\right)$$
$$+\left(1 - \sum_{l=1}^{u-1} H_i.L^{(l)} - \sum_{l=u+1}^{d} H_i.U^{(l)}\right) \cdot e^{(u)} \quad (4)$$

where

$$u = \text{argmin}_j \left(\sum_{l=1}^{j} H_i.L^{(l)} \cdot e^{(l)} + \sum_{l=j+1}^{d} H_i.U^{(l)} \cdot e^{(l)} < 1 \right) - 1$$

Definition 7 (Split dimension). *The split dimension refers to u in Lemma 1 and 2.*

Note that the value of u in Lemma 1 and 2 (also called the split dimension) may be different.

For summarization, assume:

$$UB(H_i)' \triangleq \min_{w \in H_i} (f(w, r.U) - f(w, q)) \quad (5)$$

$$LB(H_i)' \triangleq \max_{w \in H_i} (f(w, r.L) - f(w, q)) \quad (6)$$

Note that $UB(H_i)'$ and $LB(H_i)'$ can be computed similarly. The four operators can be implemented in the following way.

- Implement belowP(r, H_i). If and only if $UB(H_i) < 0$ holds.
- Implement aboveP(r, H_i). If and only if $LB(H_i) \geq 0$ holds.
- Implement withinP(r, H_i). If and only if $(UB(H_i) \geq 0) \wedge (UB(H_i)' \leq 0) \wedge (LB(H_i) \leq 0) \wedge (LB(H_i)' \geq 0)$ holds.
- Implement acrossP(r, H_i). Returns TRUE once all the above three operators return FALSE.

The implementations of the first two operators are straightforward due to the definitions of $UB(H_i)$ and $LB(H_i)$, while the implementation of withinP is a bit sophisticated. Note that $r.U$ and $r.L$ are two special points in r: $\forall w \in H_i$, $r.U$ and $r.L$ have the greatest and the smallest ranking scores respectively. Since both points meet the semantic (see withinP relationship in Section 5.1), all the remaining points in r also meet the semantic.

5.3 Algorithm Description

Our batch pruning algorithm (BPA, Algorithm 2) has two phases. At the first phase, it checks each bucket H_i to compute the lower and upper rank bounds ($H_i.lb$ and $H_i.ub$) of any weight in the bucket (at lines 3-21). In other words, any customer w in H_i prefers at least $H_i.lb$ (at most $H_i.ub$) other products to q. Hence, we can avoid computing the exact rank value for each weight in the bucket. At the second phase, we compute the exact rank of each weight in all remaining buckets if the first phase is insufficient (at lines 22-23).

At the first phase, Algorithm 2 traverses R-tree in a BFS (breadth first search) manner. The upper (or lower) rank bound is updated if r is above (or below) H_i. But if r is within the range of H_i, the bounds cannot be updated for now, so we append r to the local queue Q_{H_i} for future processing.

Algorithm 2: Batch pruning approach (BPA) (R, H, q, k)

```
1  Let $Q$ denote a global empty queue, $Q_{H_i}$ denote a
   queue for bucket $H_i$ to record some MBR references;
2  $minRank \leftarrow$ MAXVALUE;
3  foreach ( valid $H_i$ in $\mathcal{H}$) do
4      clear $Q$ and $Q_{H_i}$;  enqueue($Q$, $R$.root);
5      $H_i.lb \leftarrow 0$;   $H_i.ub \leftarrow |D|$;   // $D$ is the data set
6      while (($r$ = dequeue($Q$)) $\neq \emptyset$) do
7          if (belowP($r, H_i$)) then
8              $H_i.lb \leftarrow H_i.lb + |r|$;
9              if ($H_i.lb > minRank$) then
10                 Drop $Q_{H_i}$; break;
11         else if (aboveP($r, H_i$)) then
12             $H_i.ub \leftarrow H_i.ub - |r|$;
13         else if (withinP($r, H_i$)) then
14             enqueue($Q_{H_i}, r$);
15         else
16             foreach (child node $r'$ of $r$) do
17                 if (withinP($r', H_i$)) then
18                     enqueue($Q_{H_i}, r'$);
19                 else
20                     enqueue($Q, r'$);
21     Update $minRank$ by using $\{H_i.ub\}$;
22 Remove all $H_i$ in $\mathcal{H}$ such that $H_i.lb > minRank$;
23 Compute the exact rank of each weight in all remaining
   $H_i$'s by using $\{Q_{H_i}\}$;
24 return $k$ weights with smallest ranks;
```

Otherwise (r is across H_i), we append the child nodes to the global queue Q or local queue Q_{H_i} respectively, according to the result of withinP operator. The $minRank$ refers to a global threshold value, which ensures at least k weights have a rank value smaller than $minRank$. Formally, considering triples like $(|H_i|, H_i.lb, H_i.ub)$ where $|H_i|$ denotes the number of weights in H_i, the value of $minRank$ is computed as: $minRank = \text{argmin}_x \left(\left(\sum_{H_i.ub \leq x} |H_i|\right) \geq k \right)$.

Example 2. *Figure 5 shows the number of weights and the rank bounds of five histograms ($H_1 \cdots H_5$) for Figure 4. Here, H_3 contains 5 weights, $H_3.lb = 5$ and $H3.ub = 9$. Let $k = 3$, then $minRank = 6$. Hence, H_1 and H_2 can be pruned, while H_3 cannot due to $H_3.lb < minRank$.*

BPA prunes computations from the perspectives of both data points and weight vectors. In comparison with TPA, BPA reduces the time complexity from $O(m \cdot n)$ to $O(m_r \cdot n_b)$ where $m_r = |R|$ and $n_b = |H|$. Generally, $m_r \ll |D|$ and $n_b \ll |W|$.

6. MARKED PRUNING APPROACH (MPA)

BPA needs to evaluate each (H_i, r) pair with $O(d \cdot \log d)$ cost for an arbitrary valid bucket H_i and an MBR r in the queue (Algorithm 2). This cost cannot be ignored for a large R-tree and a large weight histogram, which inspires us that the execution performance can be even improved after removing this kind of overhead. Recall that for an MBR r, $UB(H_i)$ describes the maximal ranking score difference between $r.U$ and q for any weight in H_i. We find the value

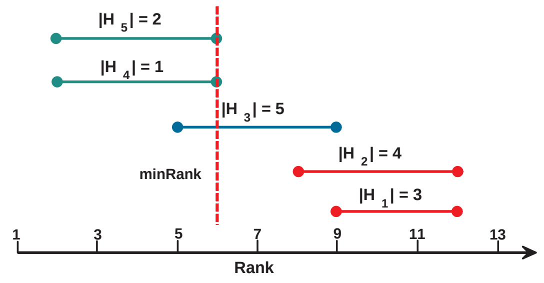


Figure 5: Computing *minRank*

of $UB(H_j)$ can be computed with $O(1)$ cost after given the value of $UB(H_i)$, where H_j is a neighbor of H_i, i.e, they only differ at one dimension. The discussion of $LB(H_i)$ is symmetric. In this section, we first analyze the computations of $UB(H_j)$ and $LB(H_j)$, and then the algorithm details.

6.1 Analysis

We first define two neighborhood relationships between a pair of weight buckets: positive neighbor (PoN) and negative neighbor (NeN).

Definition 8 (Positive Neighbor, PoN). *Let H_i and H_j denote two weight buckets. H_j is H_i's positive neighbor if and only if (i) $\exists g : H_j.L^{(g)} = H_i.L^{(g)} + \frac{1}{c}$, and (ii) $\forall l, l \neq g : H_i.L^{(l)} = H_j.L^{(l)}$. We call g the* affected dimension.

Definition 9 (Negative Neighbor, NeN). *Let H_i and H_j denote two weight buckets. H_j is H_i's negative neighbor if and only if H_i is H_j's positive neighbor. The* affected dimension *is defined as aforementioned.*

The difference between a bucket (say, H_i) and its PoN (or NeN) (say, H_j) is only one dimension, the affected dimension g. Hence, it enables us to compute $UB(H_j)$ (or $LB(H_j)$) efficiently after computing $UB(H_i)$ (or $LB(H_i)$) in advance.

Theorem 1. *Assume H_j is the PoN of H_i. Let g and u denote H_i's affected dimension and split dimension respectively, and $\Delta = w^{(u)} - H_i.L^{(u)}$. Then, $UB(H_j)$ and $LB(H_j)$ are computed by Equation (7)-(8).*

$$UB(H_j) = UB(H_i) + \frac{e^{(g)} - e^{(u-1)}}{c} + \Delta \cdot \left(e^{(u-1)} - e^{(u)}\right) \quad (7)$$

$$LB(H_j) = LB(H_i) + \frac{e^{(g)} - e^{(u+1)}}{c} + \Delta \cdot \left(e^{(u+1)} - e^{(u)}\right) \quad (8)$$

Proof. The coefficients $\{e^{(l)}\}$ are firstly used in Equation (2). We study four cases to show the correctness of Equation 7.

Case *i* ($g < u \wedge g \neq u - 1$): We raise $w^{(g)}$ by $\frac{1}{c}$, reduce $w^{(u)}$ by Δ, and reduce $w^{(u-1)}$ by $\frac{1}{c} - \Delta$.

Case *ii* ($g = u - 1$): We raise $w^{(u-1)}$ by Δ, and reduce $w^{(u)}$ by Δ.

Case *iii* ($g = u$): We raise $w^{(u)}$ by $\frac{1}{c} - \Delta$, and reduce $w^{(u-1)}$ by $\frac{1}{c} - \Delta$.

Case *iv* ($g \geq u + 1$): We raise $w^{(g)}$ by $\frac{1}{c}$, reduce $w^{(u)}$ by Δ, and reduce $w^{(u-1)}$ by $\frac{1}{c} - \Delta$.

In all cases, Equation 7 holds. Since the proof of Equation 8 is similar, we omit it due to space limitations. □

Theorem 2. *Assume H_j is the NeN of H_i. Let g and u denote H_i's affected dimension and split dimension, and $\Delta = H_i.U^{(u)} - w^{(u)}$. Then, $UB(H_j)$ and $LB(H_j)$ are computed by Equation (9)-(10).*

$$UB(H_j) = UB(H_i) + \frac{e^{(u+1)} - e^{(g)}}{c} + \Delta \cdot \left(e^{(u)} - e^{(u+1)}\right) \quad (9)$$

$$LB(H_j) = LB(H_i) + \frac{e^{(u-1)} - e^{(g)}}{c} + \Delta \cdot \left(e^{(u)} - e^{(u-1)}\right) \quad (10)$$

We omit the proof due to space limitations.

6.2 Algorithm Description

Algorithm 3: Marked Pruning Approach (MPA) (R, H, q, k)

```
1  Initialize all entries in $B$ to $UNKNOWN$;
2  Empty a queue $Q$;
3  $minRank \leftarrow$ the number of objects in $R$;
4  foreach valid $H_i$ in $\mathcal{H}$ do
5      clear $Q$;  enqueue($Q$, $R$.root);
6      $H_i.lb \leftarrow 0$;  $H_i.ub \leftarrow$ the number of objects in $R$;
7      while $((r = dequeue(Q)) \neq \emptyset)$ do
8          if $(B[r, H_i] = BELOW)$ then
9              $H_i.lb \leftarrow H_i.lb + |r|$ ;
10         else if $(B[r, H_i] = ABOVE)$ then
11             $H_i.ub \leftarrow H_i.ub - |r|$ ;
12         else if $(belowP(r, H_i))$ then
13             $H_i.lb \leftarrow H_i.lb + |r|$ ;
14             mark$(H_i, BELOW, r)$ ;
15         else if $(aboveP(r, H_i))$ then
16             $H_i.ub \leftarrow H_i.ub - |r|$;
17             mark$(H_i, ABOVE, r)$ ;
18         else
19             if ($r$ is non-leaf node) then
20                 $\forall r'$ ($r'$ is $r$'s child), enqueue$(r', Q)$;
21         if $(H_i.lb > minRank)$ then break;
22     Update $minRank$ by using $\{H_i.ub\}$;
23 Remove all $H_i$ in $\mathcal{H}$ such that $H_i.lb \geq minRank$;
24 Compute the rank of each weight in all remaining $H_i$'s;
25 return $k$ weights with smallest rank;
```

Algorithm 3 illustrates the main steps of our Marked Pruning Approach (MPA). Although similar to Algorithm 2 in shape, MPA still has significant improvement. We use a 2-D array B (with a size of $|R| \times |H|$) to record all relationships, where R is the R-tree and $\mathcal{H}$ is a set containing all valid weight buckets. With the help of B, the lower and upper rank bounds of a weight bucket H_i can be set more efficiently. We also use the subroutine mark (Algorithm 4, to be discussed later) to mark the entries in B with a low cost. There are four kinds of relationships: *ABOVE*, *BELOW*, *UNKNOWN*, and *OTHER*. Initially, all entries in B are set *UNKNOWN* since the exact relationships are unknown in advance. The *BELOW* (or *ABOVE*) flag is set if the rectangle r is below (or above) the corresponding hyperplane family. Similarly, the *OTHER* flag refers to the withinP or acrossP relationship aforementioned.

For each entry H_i, we check each MBR r in R iteratively. At first, if $B[r, H_i]$ has been set to *BELOW* or *ABOVE*,

Algorithm 4: mark(H_i, GF, r)

```
1  Let $Q'$ denote an empty queue ;
2  enqueue($H_i, Q'$) ;
3  while ($H_i$ = dequeue($Q'$) $\neq \emptyset$) do
4      foreach $H_j$, the $H_i$'s neighbor do
5          if ($H_j.Flag = UNKNOWN$) then
6              if ($GF = BELOW$) then
7                  Update $UB(H_j)$ by Equation (7), (9) ;
8                  $FLAG \leftarrow (UB(H_j) < 0)$;
9              else
10                 Update $LB(H_j)$ by Equation (8), (10) ;
11                 $FLAG \leftarrow (LB(H_j) \geq 0)$;
12             if ($FLAG = TRUE$) then
13                 $B[r, H_j] \leftarrow GF$ ;
14                 enqueue($H_j, Q'$) ;
15             else
16                 $B[r, H_j] \leftarrow OTHER$;
```

we update $H_i.lb$ or $H_i.ub$ immediately (at lines 9 and 11). Otherwise, we need to invoke belowP or aboveP and then mark (at lines 12-17). Finally, if r is a non-leaf node and the relationship is neither $ABOVE$ nor $BELOW$, we will push every child node of r into the queue Q for further processing.

Algorithm 4 shows how to mark the relationships between an MBR r and the neighbors of a bucket H_i in the BF-S (breadth-first search) manner. The input parameter GF refers to a global flag in $\{ABOVE, BELOW\}$. The algorithm will try to mark all neighbors of H_i (say, H_j) iteratively if the relationships have not been marked yet, i.e, $H_j.Flag = UNKNOWN$. Assuming $GF = BELOW$, it first computes $UB(H_j)$ by using Equation 7 (if H_j is PoN), or Equation 9 (if H_j is NeN) to mark $B[r, H_j]$. If $UB(H_j) < 0$ (at line 8), we set $B[r, H_j] \leftarrow BELOW$ (at line 13). Similarly, assuming $GF = ABOVE$, we will compute $LB(H_j)$ by using Equation 8 or 10 to mark $B[r, H_j]$ if possible. Finally, if the relationship is neither $BELOW$ nor $ABOVE$, $B[r, H_j]$ is set to $OTHER$ (at line 16).

Although the time complexity of MPA ($O(m_r \cdot n_b)$) is the same as BPA, MPA runs faster than BPA in most situations due to the elimination of significant overheads. Since MPA needs to maintain a 2-D array B to reserve *marking* information that costs $O(|R| \times |\mathcal{H}|)$, the overall space consumption of MPA is also $O(|R| \times |\mathcal{H}|)$.

7. DISCUSSION

Summarization of Four Approaches. Table 3 compares four approaches. Without building any index in advance, NA just scans the product and weight data sets linearly. In contrast, all three other methods maintain an R-tree index upon all products. Moreover, BPA and MPA also maintain a histogram index upon all weights.

There is actually a tradeoff between time and space complexities. For example, NA runs the slowest, but meanwhile it only consumes tiny memory space. BPA and MPA are faster than the other two approaches, after maintaining an additional Histogram index. In addition, MPA maintains a 2-D array to record marking information during execution.

Table 3: The Summary of Four Approaches

	With index	Time complex.	Space complex.
NA	None	$O(m \cdot n)$	$O(k)$
TPA	R-tree	$O(m_r \cdot n)$	$O(m_r)$
BPA	R-tree + Hist.	$O(m_r \cdot n_b)$	$O(m_r + n_b)$
MPA	R-tree + Hist.	$O(m_r \cdot n_b)$	$O(m_r \cdot n_b)$

Notations: (i) m: the number of products, (ii) n: the number of weights, (iii) m_r: the number of MBRs in R, and (iv) n_b: the number of buckets in $\mathcal{H}$.

Although MPA has the largest space consumption, the total cost is affordable since each entry in the 2-D array only costs 2 bits (there are four kinds of relationships). Detailed evaluation of memory consumption is shown in 8.2.

Counting the number of *Valid* Buckets. As mentioned in Section 5, a *valid* bucket H_i conforms to the constraint: $(\sum_{l=1}^{d} H_i.L^{(l)} < 1) \wedge (\sum_{l=1}^{d} H_i.U^{(l)} > 1)$. It is equal to find all buckets H_i such that $\sum_{l=1}^{d} H_i.L^{(l)} \in (1 - \frac{d}{c}, 1)$, since $\sum_{l=1}^{d}(H_i.U^{(l)} - H_i.L^{(l)}) = \frac{d}{c}$. Let $N(s, d)$ denote the number of buckets in d-dimensional space satisfying $\sum_{l=1}^{d} H_i.L^{(l)} = \frac{s}{c}$. We derive that $N(s, d) = \binom{s+d-1}{d-1}$. Since $\sum_{i=1}^{m} \binom{k+m}{k} = \binom{k+m+1}{k+1}$, the number of *valid* buckets is computed as:

$$\sum_{s=c-d+1}^{c-1} N(s, d) = \binom{c+d-1}{d} - \binom{c}{d} \tag{11}$$

Distributed design of R-kRanks The distributed design is important to deal with large-scale data. Fortunately, all the proposed solutions can be adapted to the distributed environment since they process each weight or bucket one by one. We draft the distributed solution briefly. Consider a distributed system with one master and several workers. At first, the master node generates an R-tree for all products (for TPA, BPA and MPA) and a weight histograms (for BPA and MPA). Next, the master sends R-tree to each worker. Subsequently, it divides all weights (or all buckets) into several groups, and then send each group to different workers. Each worker can execute the algorithm independently. Finally, all workers send the results back to the master. Details are omitted due to the lack of enough space.

8. EXPERIMENTS

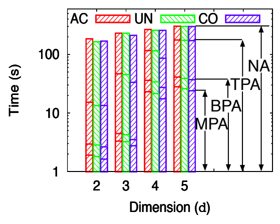
We report experimental reports in this section. All codes are written in JAVA, and run on a stand-alone computer with an Intel CPU/2GHz and 8GB memory. In each experiment, we implement the queries for 1,000 times, each with a randomly selected query point, and report the average.

8.1 Dataset and setting

We use two kinds of data sets in this study, including product set D and customer set W.

Product data set D: We use synthetic and real-life product data sets.

- *Synthetic datasets*: We generate three synthetic datasets, which follow uniform (UN), correlated (CO) and anti-correlated (AC) distributions respectively. See [2] for detailed generation methods.


(a) Average time

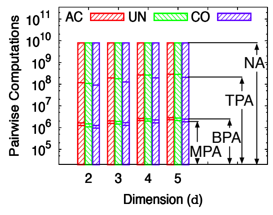
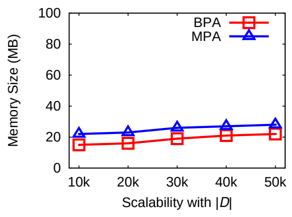

(b) Pairwise comp.

Figure 6: Performance by varying d [NA (the longest bar) vs. TPA (the second longest bar) vs. BPA (the third longest bar) vs. MPA (the shortest bar)]


(a) Synthetic dataset

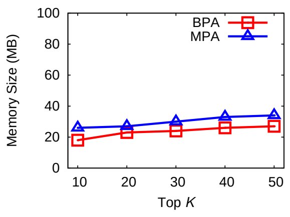

(b) Real dataset

Figure 7: Memory consumption

- *Real datasets*: We use two real datasets. The first dataset, namely *Bank* dataset, contains 45,211 records and is related to direct marketing campaigns (phone calls) of a Portuguese banking institution, including customer's age, balance, duration etc [4]. The second data set, namely *Taobao* dataset, contains 51,026 records and is related to the product information in Taobao (`http://www.taobao.com`), the most popular e-commerce web-site in China, including current price, sold amount, the lowest price and so on. We choose 3 and 5 numeric dimensions from these datasets respectively for further testing.

Preference weight data set W: We generate two synthetic data sets, which are also used in [21, 22, 24, 25].

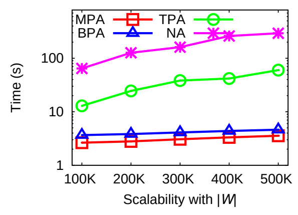
- *Uniform distribution (UNI)*: We repeatedly select one vector from d-dimensional space, and then normalize it to a standard form.

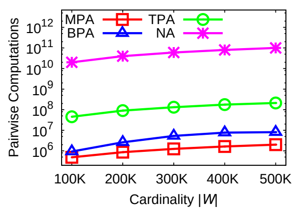
- *Clustered distribution (CLU)* : This data set is created based on two parameters: g and σ^2. We first randomly select g cluster centroids in d-dimensional space. Then we generate some weights around the centroids with a variance of σ^2 in each dimension.

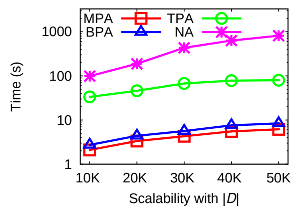
8.2 Experimental Reports

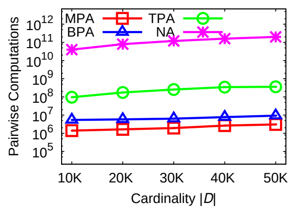
Time-efficiency: Figure 6 illustrates the performance of four methods (NA, TPA, BPA and MPA) when varying the number of dimensions. We use UNI weight set and three product sets, including UN, CO and AC. We set $|D| = 20K$, $|W| = 400K$ and $k = 10$. Each query point is randomly selected in the domain. The y-axes of Figure 6(a) and (b) represent the executing time and the total number of pairwise computations (refers to computing the ranking score for

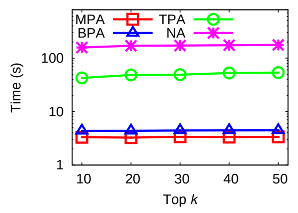
[4]`http://archive.ics.uci.edu/ml/datasets/Bank+Marketing`.


(a) Time for $|W|$


(b) Computations for $|W|$


(c) Time for $|D|$


(d) Computations for $|D|$


(e) Time for k

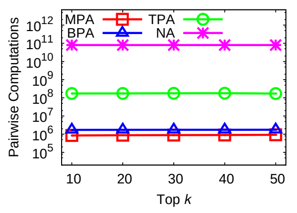

(f) Computations for k

Figure 8: Scalability of four approaches [NA vs. TPA vs. BPA vs. MPA]

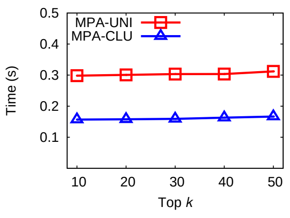
a product-customer pair) respectively. The longest, the second longest, the third longest and the shortest bars represent the performance of NA, TPA, BPA and MPA respectively.

TPA, BPA and MPA outperform NA for at least one order of magnitude, since NA has the time complexity of $O(m \cdot n)$, while the other three algorithms can reduce the number of comparison operations efficiently by employing different pruning rules. TPA prunes some unnecessary pairwise computations from the perspective of data points, while BPA and MPA simultaneously prune unnecessary pairwise computations from the perspectives of both data points and weight vectors, avoiding scanning all $w \in W$ and $p \in D$. BPA performs worse than MPA, since MPA maintains a 2-D array to avoid unnecessary computations. In general, Figure 6(a) shows that efficiencies of TPA, BPA and MPA decrease with the increment of d. Figure 6(b) shows the pairwise computations are insensitive to the number of dimensions because the number of pairwise (w, p) is almost unchanged when d increases.

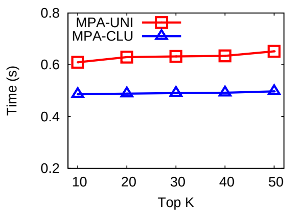
Space-efficiency: We employ Jprofile [5] to monitor memory consumption of BPA and MPA. By default, we set $|W| = 100k$ and use CLU weight dataset. Figure 7(a) uses UN product dataset ($d = 3$). We can observe that BPA is more space-efficient than MPA, since MPA uses a 2-D array to reduce the processing time. In addition, such kind of space overhead is also not so significant, because each entry is tiny.

Scalability: Figure 8 reports the scalability of NA, TPA, BPA and MPA under different $|D|$, $|W|$ and k. The metrics are time cost and pairwise computations. By default, $|D| =$

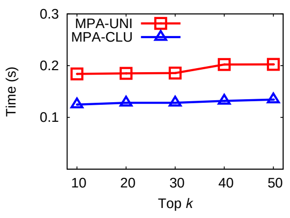
[5]`http://www.ej-technologies.com/products/jprofiler/overview.html`



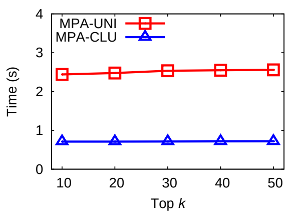
(a) Synt. dataset(UN)



(b) Synt. dataset (AC)

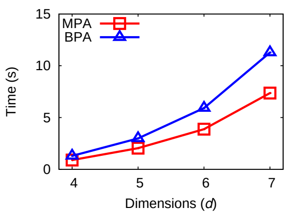


(c) Real dataset (Bank)

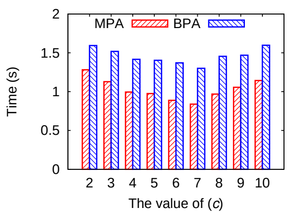


(d) Real dataset (Taobao)

Figure 9: Performance of MPA for various data set D



(a) The effect of d (UN)



(b) The effect of c (Bank)

Figure 10: the factors: d and c

$20K$, $k = 10$, $|W| = 400K$ and $c = 10$. In different series of experiments, $|D|$ varies from $10K$ to $50K$, $|W|$ varies from $100K$ to $500K$, and k varies from 10 to 50.

Figures 8(a)-8(b) show the performance changes when $|W|$ increases from 100K to 500K. For all methods, the processing time will rise when $|W|$ increases. We also observe that NA and TPA are more sensitive to the value of $|W|$. According to Table 3, for NA and TPA, the processing cost is linear to the number of weights. Meanwhile, both BPA and MPA maintain a weight histogram for pruning, so that the processing cost increases much slower than NA or TPA.

Figures 8(c)- 8(d) show the performance changes when $|D|$ increases from 10K to 50K. For all methods, the processing time will rise when $|D|$ increases. Moreover, NA is more sensitive to $|D|$. According to Table 3, the processing cost of NA is linear to the number of products. Meanwhile, TPA, BPA and MPA maintain an R-tree for pruning, so that the processing cost increases much slower than NA.

Figures 8(e)-8(f) show the time consumption and pairwise computations of NA remain almost unchanged when k increases, because the time complexity of NA is $O(m \cdot n)$ for any k. TPA, BPA and MPA are also insensitive to k, since in general $k \ll |W|$.

The effect of weight data set: Assuming $|W| = 10K$ and $c = 4$, Figure 9 illustrates the behaviors of MPA when using different weight datasets (UNI, CLU) when k increases. MPA behaves better on the uniform weight set (UNI) than the clustered weight set (CLU) under all settings, since MPA can mark more neighbor buckets and thus prune more buckets when the weight set is clustered.

The effect of dimensionality and cell width: Assuming $|D| = 20K$, $|W| = 100K$, $k = 10$, $d = 3$, and $c = 4$. We use CLU weight dataset in this series of experiments. Figure 10(a) shows the execution time when varying the number of dimensions d upon the UN product dataset. In all situations, MPA outperforms BPA. Figure 10(b) illustrates the execution time when varying the cell width (c) upon the *Bank* dataset. It is interesting that the peak performance is obtained when $c = 7$, neither too small nor too great. The reason is that the filtering capability is insufficient when c is small. Meanwhile, the overhead of evaluating all valid buckets will rise when c increases.

8.3 Effectiveness

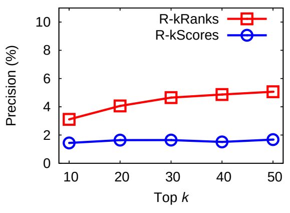
We use a real dataset (Dianping) [6] to test the effectiveness of R-kRanks, in comparison with R-kScores. The dataset contains millions of rating records about a large number of restaurants in Shanghai from Jan. 2009 to Dec. 2013. Each record has four numeric scores in four dimensions: *average cost per person* (ac), *taste* (ta), *environment* (en) and *service* (se). The training set contains records in the first four years, while the testing set contains records in the last year. In every set, each user has at least 10 rating records .

- *product dataset* consists of 9,800 restaurants rated at least ten times. The value in each dimension is computed as the average score of all corresponding records.
- *preference dataset* refers to 3,096 users who have rated at least ten times. In general, a user prefers a restaurant with low *average cost per person*, but high *taste*, *environment* and *service*. Hence, the weight of a dimension should be set low if a user does not care about that dimension. For example, it is reasonable to set a low weight on *average cost per person* for a user who often choose expensive restaurants. Let s_{ac}, s_{ta}, s_{en} and s_{se} denote the average scores of a user in four dimensions, and ϕ_{ac}, ϕ_{ta}, ϕ_{en} and ϕ_{se} denote the proportions of restaurants lower than the corresponding score. The weight vector is represented as $(\frac{1-\phi_{ac}}{sum}, \frac{\phi_{ta}}{sum}, \frac{\phi_{en}}{sum}, \frac{\phi_{se}}{sum})$, where $sum = 1 - \phi_{ac} + \phi_{ta} + \phi_{en} + \phi_{se}$.

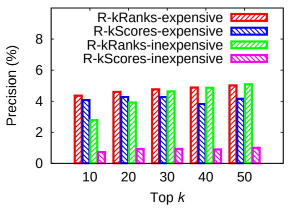
We evaluate the precision of two query types upon all hot restaurants with at least 100 ranking records, where precision is defined as the proportion of users who have visited the query restaurant. Figure 11(a) shows R-kRanks outperforms R-kScores significantly in all situations: the precision of R-kRanks is at least 1.2 times greater than that of R-kScores. Figure 11(b) reports the effect after dividing all restaurants into two groups: expensive (with average cost per person ≥ 100 RMB) and inexpensive (with average cost per person < 100 RMB). For the expensive restaurants, R-kRanks and R-kScores behave similarly, but for the other group, R-kRanks is significantly better than R-kScores. The reason is that R-kScores only works well for a small part of restaurants (especially expensive ones), while R-kRanks works well for all restaurants. Table 4 explains this phenomenon below.

We also test the impact of different query restaurants. Along each dimension, we first sort the restaurants and then split them into three groups of equal size, denoted as H (highest ones), M(middle ones) and L (lowest ones). We select

[6]`http://www.dianping.com`, one of the biggest lifestyle and group buying websites in China



(a) All hot restaurants



(b) Two groups

Figure 11: Effectiveness of R-kRanks

Table 4: Query Results on Dianping Dataset

	R-kRanks				R-kScores			
	ac	ta	en	se	ac	ta	en	se
R_1	L	H	L	L	H	H	H	H
R_2	M	L	L	L	H	H	H	H
R_3	H	M	H	M	H	H	H	H
R_4	H	H	H	H	H	H	H	H

four different restaurants, $R_1(L, H, L, H)$, $R_2(M, L, L, L)$, $R_3(M, M, M, M)$ and $R_4(H, H, H, H)$. For example, R_4 describes an expensive restaurant with good taste, environment and service. Table 4 lists users most likely returned by R-kRanks and R-kScores where $k = 10$. For example, given R_2 as the query restaurant, most of users in the result set are (M, L, L, L). The users returned by R-kRanks are more matchable with the query restaurant compared with other users, while R-kScores tends to return the users who prefer expensive restaurants with good taste, environment and service, since their ranking scores are low.

9. CONCLUSION

R-kRanks, proposed in this paper, is critical in various applications, such as job-hunting and dating. We devise three pruning-based methods to answer R-kRanks query efficiently. Tree-based pruning approach (TPA) eliminates some unnecessary pairwise computations from the perspective of data points. Batch pruning approach (BPA) reduces unnecessary pairwise computations from both perspectives of data points and weight vectors. Besides the two pruning rules above, marked pruning approach (MPA) also reuses previous computation results of some buckets to further reduce the time consumption. Thereafter, we conduct extensive experiments on real and synthetic data sets to verify the effectiveness and efficiency of the proposed methods.

There are two possible pieces of future work. The first one is to devise approximate solutions for R-kRanks query. The second one is to devise distributed solutions. Such two directions are helpful to handle large-scale data.

Acknowledgement

Our research is supported by the 973 program of China (No. 2012CB316203), NSFC (61321064, 61370101 and 61033007), Shanghai Knowledge Service Platform Project (No. ZF1213).

10. REFERENCES

[1] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of ACM SIGMOD*, pages 28–39, 2003.

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

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

[4] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top-k queries in two dimensions. In *Proc. of DASFAA (1)*, pages 201–208, 2013.

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

[6] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of PODS*, pages 216–226, 1996.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS*, pages 102–113, 2001.

[8] S. Ge, L. U, N. Mamoulis, and D. Cheung. Efficient all top-k computation - a unified solution for all top-k, reverse top-k and top-m influential queries. *IEEE Trans. Knowl. Data Eng.*, 25(5):1015–1027, 2012.

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

[10] 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, Oct. 2008.

[11] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *Proc. of the VLDB Endowment*, 1(1):301–312, 2008.

[12] W. Jin, M. Ester, and J. Han. Efficient processing of ranked queries with sweeping selection. In *Proc. of PKDD*, pages 527–535, 2005.

[13] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 201–212, 2000.

[14] K. C. Lee, B. Zheng, and W.-C. Lee. Ranked reverse nearest neighbor search. *IEEE Trans. Knowl. Data Eng.*, 20(7):894–910, 2008.

[15] K. C. K. Lee, M. Ye, and W.-C. Lee. Reverse ranking query over imprecise spatial data. In *Proc. of COM.Geo*, 2010.

[16] C. Li. *Enabling data retrieval: by ranking and beyond*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[17] X. Lian and L. Chen. Reverse skyline search in uncertain database. *ACM TODS*, 35(1):3:1–3:49, 2010.

[18] X. Lian and L. Chen. Probabilistic inverse ranking queries in uncertain databases. *VLDB J.*, 20(1):107–127, 2011.

[19] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proc. of ACM SIGMOD*, pages 635–646, 2006.

[20] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *Proc. of VLDB*, pages 99–108, 2001.

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

[22] 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.

[23] A. Vlachou, C. Doulkeridis, and K. Nørvåg. Monitoring reverse top-k queries over mobile devices. In *Proc. of MobiDE*, pages 17–24, 2011.

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

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

[26] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *Proc. of VLDB*, pages 235–246, 2006.

[27] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. of ICDE*, pages 28–39, 2001.

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