Aggregate Reverse Rank Queries

Yuyang Dong(✉), Hanxiong Chen, Kazutaka Furuse, and Hiroyuki Kitagawa

Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki, Japan
tou@dblab.is.tsukuba.ac.jp, {chx,furuse,kitagawa}@cs.tsukuba.ac.jp

Abstract. Recently, reverse rank queries have attracted significant research interest. They have real-life applicability, such as in marketing analysis and product placement. Reverse k-ranks queries return users (preferences) who favor a given product more than other people. This helps manufacturers find potential buyers even for an unpopular product. Similar to the cable television industry, which often bundles channels, manufacturers are also willing to offer several products for sale as one combined product for marketing purposes.

Unfortunately, current reverse rank queries, including Reverse k-ranks queries, only consider one product. To address this limitation, we propose the *aggregate reverse rank queries* to find matching user preferences for a set of products. To resolve this query more efficiently, we propose the concept of pre-processing the preference set and determining its upper and lower bounds. Combining these bounds with the query set, we proposed and implemented the tree pruning method (TPM) and double-tree method (DTM). The theoretical analysis and experimental results demonstrated the efficacy of the proposed methods.

Keywords: Similarity search · Aggregate reverse rank queries · Tree-based method

1 Introduction

Top-k and reverse k-rank queries are two different kinds of view-models. The top-k query is a user view-model that helps consumers by obtaining the best k products that match a user's preference. On the other hand, the reverse k-rank query [18] supports manufacturers by discovering potential consumers through retrieving the most appropriate user preferences. Therefore, it is a manufacturer view-model and can be used as a tool for identifying customers and estimating product marketing.

Figure 1 shows an example of a reverse 1-rank query. Five different cell phones (p_1–p_5) are scored on "smart" and "ratings" in a table (Fig. 1(a)). The preferences of two users Tom and Jerry are in another table (Fig. 1(b)) and consist of the weights for all attributes. The score of a cell phone based on user preference is determined from the inner product of the cell phone attributes vector and user preference vector. Without loss of generality, we assumed that minimum values are preferable. The results of the reverse 1-rank query are given in the last cells

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part II, LNCS 9828, pp. 87–101, 2016.
DOI: 10.1007/978-3-319-44406-2_8

(a) User preferences and Ranks

user	w[smart]	w[rating]	Ranks
Tom	0.8	0.2	p3,p2,p1,p4,p5
Jerry	0.3	0.7	p2,p5,p3,p4,p1

(b) Cell phone ranks and R-1Rank

	p[smart]	p[rating]	Score on Tom	Score on Jerry	Rank in Tom	Rank in Jerry	R-1Rank
p1	6	7	6.2	6.7	3rd	5th	Tom
p2	2	3	2.2	2.7	2nd	1st	Jerry
p3	1	6	2.0	4.5	1st	3rd	Tom
p4	7	5	6.6	5.6	4th	4th	Tom
p5	8	2	6.8	3.8	5th	2nd	Jerry

Fig. 1. The example of reverse 1-rank queries.

of Fig. 1(b). For example, Tom believes that p_1 is the third-best phone, while Jerry thinks that p_1 is the fifth-best. To manufacturers, Tom is more likely to buy p_1 than Jerry; hence, the reverse 1-rank query returns Tom as the result.

Motivation. Manufacturers use "product bundling" for marketing purposes. Product bundling is offering several products for sale as one combined product. It is a common feature in many imperfectly competitive product markets. For example, Microsoft Co., Ltd. includes a word processor, spreadsheet, presentation program, and other useful software into a single Office Suite. The cable television industry often bundles various channels into a single tier to expand the channel market. Manufacturers of video games are also willing to group a popular game with other games of the same theme in the hope of obtaining more benefits by selling them together.

Because product bundling is an important business approach, helping manufacturers target buyers for their bundled products is important. Unfortunately, the reverse k-rank query and other kinds of reverse ranking queries are all designed for just one product. To address this limitation, we propose a new query definition that finds k customers with the smallest aggregate rank values, where the rank of a product set is defined as the sum of each product's rank. We call this approach *aggregate reverse rank queries* (*AR-k queries*).

Group |Q| = 2	sum Rank in Tom	sum Rank in Jerry	AR-1Rank
p1,p2	5 (3 + 2)	6 (5 + 1)	Tom
p2,p3	3 (2 + 1)	4 (1 + 3)	Tom
p4,p5	9 (4 + 5)	6 (4 + 2)	Jerry

Fig. 2. The example of aggregate reverse 1-rank queries.

Figure 2 shows an example of an AR-1 query. There are three groups of bundled products: $\{p_1, p_2\}$, $\{p_2, p_3\}$, and $\{p_4, p_5\}$. The aggregate rank of $\{p_1, p_2\}$ is 5 according to Tom's preferences and 6 according to Jerry's. Thus, the AR-1 query returns Tom as the result because Tom prefers this bundle the most.

Contribution. This paper makes the following contributions:

- To the best of our knowledge, we are the first to address the "one product" limitation of reverse k-rank queries. We propose a new AR-k query that returns the k user preferences that best match a set of products.
- We propose the concept of pre-processing preferences to determine possible upper and lower bounds. This process can be done before the AR-k query is issued to enhance its efficiency and is implemented with the proposed tree-pruning method (TPM) and double-tree method (DTM).
- Along with the theoretical analysis, we also performed experiments on both real and synthetic data. The experimental results validated the efficiency of the proposed methods.

The rest of this paper is organized as follows: Sect. 2 summarizes related work. Section 3 states the definitions. In Sect. 4, we present the method of bounding the query set. Sections 5 and 6 propose two solutions (TPM and DTM) of AR-k. Experimental results are shown in Sect. 7 and Sect. 8 concludes the paper.

2 Related Work

Ranking is an important property for evaluating the position of a product. Many variants of rank-aware queries have been widely researched.

Ranking Query (Top-k Query). The most basic approach is the top-k query. When given a user preference, the top-k query returns k products with minimal ranking scores found by a score function. One possible approach to the top-k problem is the onion technique [1]. This algorithm pre-computes and stores convex hulls of data points in layers like an onion. [4] is an important investigation that describes and classifies top-k query processing techniques in relational databases.

Reverse Rank Query (RRQ). Reverse top-k queries [10,12] have been proposed to evaluate the impact of a potential product on the market based on the preferences of users who treat it as a top-k product. For an efficient reverse top-k process, Vlachou et al. [13] proposed a branch-and-bound algorithm (BBR) using boundary-based registration and a tree base. Vlachou et al. [11,14] have reported various applications of reverse top-k queries. However, in order to answer the reverse query for some less-popular objects, [18] proposed the reverse k-rank query to find the top-k user preferences with the highest rank for a given object among all users.

Other Reverse Queries. Other related research on reverse queries is listed below. Given a data point, queries are performed to find result sets containing this data point. In contrast to the nearest-neighbor search, Korn and

Muthukrishnan [5] proposed the reverse nearest-neighbour (RNN) query. Besides the nearest neighbor, Yao et al. [17] proposed the reverse furthest neighbor (RFN) query to find points where the query point is deemed as the furthest neighbor. For reverse k nearest neighbor (RKNN), Yang et al. [15] analyzed and compared notable algorithms from [2,7–9,16]. RKNN differs from RRQ because it evaluates the relative L_p distance between two points in one Euclidean space. However, RRQ focuses on the absolute ranking among all objects, and scores are found via the inner product function. In addition, RKNN treat the user preference and product as the same kind of point in the same space, while RRQ has two data sets of different data spaces. The reverse skyline query uses the advantages of products to find potential customers based on the dominance of competitors' products [3,6]. The preference of each user is described as a data point representing the desirable product. But in RRQ, the preference is described as a weight vector.

3 Problem Statement

The assumption of the product database, preference database and the score function between them are same with the related research [10,13,18]. Let there be a product data set P and preference data set W. Each $p \in P$ is a d-dimensional vector that contains d non-negative scoring attributes. p is represented as a point $p = (p[1], p[2], ..., p[d])$, where $p[i]$ is the attribute value of p in the ith dimension. The preference $w \in W$ is also a d-dimensional weighting vector, and $w[i]$ is a non-negative weight that evaluates $p[i]$, where $\sum_{i=1}^{d} w[i] = 1$. The score is defined as the inner product of p and w, which is expressed by $f(w, p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. Given a query q, which is in the same space as, but not necessarily an element of P, the reverse k-rank query [18] is defined as follows.

Definition 1 (*rank*(w, q)). *Given a point set P, weighting vector w, and query q, the rank of q by w is $rank(w, q) = |S|$, where $S \subseteq P$ and $\forall p_i \in S, f(w, p_i) < f(w, q) \wedge \forall p_j \in (P - S), f(w, p_j) \geq f(w, q)$.*

Definition 2 (*reverse k-ranks query*). *Given a point set P, weighting vector set W, positive integer k, and query q, the reverse k-rank query returns the set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $rank(w_i, q) \leq rank(w_j, q)$ holds.*

To deal with a query having more than one query point, we propose the AR-k query, which is formally defined as follows.

Definition 3 (*aggregate reverse rank query, AR-k*). *Given a point set P, weighting vector set W, positive integer k, and query point set Q, the AR-k query returns the set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $ARank(w_i, Q) \leq ARank(w_j, Q)$ holds.*

Three aggregate evaluation functions were considered for ARank:

- **Sum:** $ARank(w, Q) = \sum_{q_i \in Q} rank(w, q_i)$.

- **Maximum:** $ARank(w, Q) = Max_{q_i \in Q}\{rank(w, q_i)\}$.
- **Minimum:** $ARank(w, Q) = Min_{q_i \in Q}\{rank(w, q_i)\}$.

There are many other possible definitions for ARank(w,Q). We considered the above because they are the most likely to be used in real applications. Suppose that there is a set of products offered by a manufacturer and we want to help them find the most potential buyers. Then, the above three evaluating functions correspond to the following requests:
Sum: find buyers who more strongly believe that this product set is better than other people. **Maximum/Minimum:** find buyers who more strongly believe that the best/worst product in this set is better than other people.

The rest of this paper only focuses on **Sum** AR-k because **Maximum** and **Minimum** can be solved simply by using the technique of the existing reverse k-rank query. From a technical point of view, for maximum score, let q' be the query of Q such that $f(w, q') = \max_{q_i \in Q}\{f(w, q_i)\}$ with respect to w, then the rank of q', $rank(w, q')$, is also equal to $Max_{q_i \in Q}\{rank(w, q_i)\}$. Thus, we can process Maximum AR-k simply by applying the reverse k-rank query to q'. **Minimum** can be solved in a similar manner.

4 Bounding the Query Set in Advance

A naive solution to an AR-k query is to sum up the ranks for $q \in Q$ one by one against each $w \in W$ and $p \in P$. This is inefficient, especially when Q is large. Our idea is to bound the query set Q with respect to W. In this section, we introduce a sophisticated method of bounding Q with two points $Q.up$ and $Q.low$ from a subset of W. Denoted by $W_t = \{w_t^{(i)}\}_1^d$, this subset is the set of *top-weighting vector*s for all dimensions, as defined in the following,

Definition 4 *(top-weighting vector). Given a set of weighting vector W, let e_i be the direction vector for dimension i such that $e_i[i] = 1$ and $e_i[j] = 0, i \neq j$ and let $cos(a, b) = a \cdot b/(|a||b|)$ be the cosine similarity between vectors a and b. The top-weighting vector for dimension i is defined by $w_t^{(i)}$ where $w_t^{(i)} \in W$ and $\forall w \in W, cos(w_t^{(i)}, e_i) \geq cos(w, e_i)$.*

W_t can be found before the query set Q is issued, so it can be considered as cost less in terms of query processing. Because W_t contains the border of the weighting vector in all dimensions, we can use it to find the upper border and lower border points set of Q.

Definition 5 *(upper and lower border query sets Q_u and Q_l). Given a d-dimensional query points set Q,*
$Q_u = \{q_i | q_i \in Q \wedge \forall q_j \in Q, \exists w_t^{(i)} \in W_t, f(w_t^{(i)}, q_i) \geq f(w_t^{(i)}, q_j)\}$ *and*
$Q_l = \{q_i | q_i \in Q \wedge \forall q_j \in Q, \exists w_t^{(i)} \in W_t, f(w_t^{(i)}, q_i) \leq f(w_t^{(i)}, q_j)\}$.

By the definition, for each $w_t^{(i)}$ there is a corresponding $q_i \in Q_u$ (Q_l) such that q_i's score with respect to $w_t^{(i)}$ is the largest (smallest) among Q. Different $w_t^{(i)}$ may correspond to a same q_i and vice versa. Generally, it is easy to find the minimum bounding rectangle (MBR) of a point set X, and let its upper-right and lower-left corners be MBR(X).up and MBR(X).low, respectively. We show below that $Q.up = \text{MBR}(Q_u).up$ and $Q.low = \text{MBR}(Q_l).low$ bound the query set Q for the AR-k query.

Figure 3 shows the geometric view for the example of $Q.low$ and $Q.up$ where $Q = \{q_1, q_2, q_3\}$. $w_t^{(1)} = w_5$ and $w_t^{(2)} = w_1$ are the top-weighting vectors in dimensions 1 and 2, respectively. Each $w_t^{(i)}$ is also a normal vector of the hyper-planes $H(w_t^{(i)})$. For $Q.up$, in 2-dimensional space, the hyper-planes $H(w_t^{(1)})$ are the dashed lines l_1 which are perpendicular to $w_t^{(1)}$. By sweeping l_1 parallelly from far infinity toward the original point $(0, 0)$, q_1 is the first point that is touched. Hence, q_1's score with respect to w is equal to $max_{q \in Q} f(w_t^{(1)}, q)$, and q_1 is included in Q_u. In the same manner, l_2 touches q_3 first, so $q_3 \in Q_u$. $Q.up = \text{MBR}(Q_u).up$ upper-bounds the scores for Q_u. Similarly, sweeping the perpendicular dashed lines l_3 and l_4 from $(0, 0)$ toward infinity both touch q_2, hence $Q_l = \{q_2\}$ and $Q.low = q_2$.

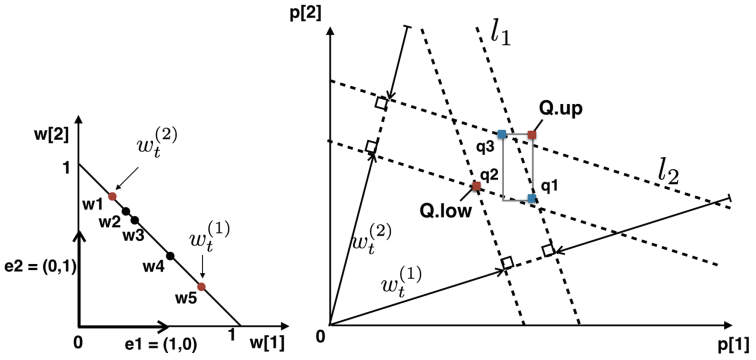


Fig. 3. A 2-dimensional example. $w_t^{(1)} = w_5$, $w_t^{(2)} = w_1$ and $Q_u = \{q_1, q_3\}$, $Q_l = \{q_2\}$, $Q.low = MBR(Q_l).low = q_2$, $Q.up = MBR(Q_u)$.up

Theorem 1 *(Correctness of Q.up and Q.low). Given top-weighting vectors set W_t, the d-dimensional query point set Q, $Q.up$ and $Q.low$. For $w \in W$ and $q \in Q$, $f(w, Q.low) \leq f(w, q) \leq f(w, Q.up)$ always holds.*

Proof. By contradiction. For $Q.up$, assume that $\exists q \in Q, q \notin Q_u$ holds so that $f(w, q) \geq f(w, Q.up)$. Therefore, $\exists q[i] > Q.up[i], i \in [1, d]$, so there must exist a $w_t^{(j)} \in W_t, j \in [1, d]$ that makes $f(w_t^{(j)}, q)$ the maximum value, and q should in Q_u. This leads to the contradiction.[1] A similar contradiction occurs with $Q.low$.

[1] The geometric view is that there exists a hyper-plane $H(w_t^{(j)})$ that first touches q rather than others.

We can use the rank of $Q.low$ to infer the bounds of the aggregate rank of Q.

Lemma 1 *(Aggregate rank bounds of Q for w): Given a set of query points Q and a weighting vector w, the lower bound of $ARank(w, Q)$ is $|Q| \times rank(w, Q.low)$, and the upper bound of $ARank(w, Q)$ is $|Q| \times rank(w, Q.up)$.*

Proof. $\forall q_i \in Q$, $\forall w[i] \geq 0$, it holds that $f(w, q_i) \geq f(w, Q.low)$ hence $rank(w, q_i) \geq rank(w, Q.low)$. By definition, $ARank(w, Q) = \sum_{q_i \in Q} rank(w, q_i) \geq |Q| \times rank(w, Q.low)$. Similarly, $|Q| \times rank(w, Q.up)$ is the upper bound of $ARank(w, Q)$.

Having W_t, the time cost of finding $Q.low$ and $Q.up$ is reduced from $O(|Q| \times |W|)$ to only $O(|Q| \times d)$, where d is the dimension of data. Considering that $|Q| \times d$ is much smaller than the size of the data set, the overhead of finding $Q.low$ and $Q.up$ is very small.

5 Tree-Pruning Method (TPM)

To enhance efficiency, our first approach, which is the *tree pruning method (TPM)*, indexes the data set P with the R-tree to group similar points and uses the bounds of MBRs (i.e., the R-tree entries) to reduce computing costs.

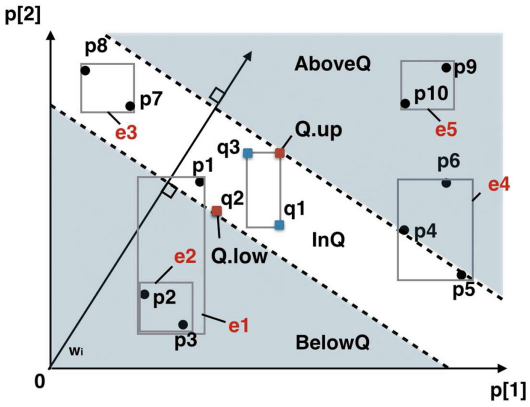


Fig. 4. The partitioned space of BelowQ, InQ and AboveQ based on $Q.low$ and $Q.up$ with a single w_i in 2d space of data set P.

First, we introduce how TPM filters P with $Q.low$ and $Q.up$. Figure 4 shows the geometric view for an example of two-dimensional data. The two dashed lines cross the boundaries ($Q.low$ and $Q.up$), and they are perpendicular to the weighting vector w_i. The space is partitioned into three parts, which are marked as $BelowQ$, InQ, and $AboveQ$ in Fig. 4. For example, e_2 is in $BelowQ$ and e_5 is in $AboveQ$. MBRs in $BelowQ$ and $AboveQ$ can be filtered by checking the upper and lower boundaries. Formally, the pruning rules are as follows.

- *Rule* 1. (MBR in $BelowQ$) If $f(w, e_p.up) < f(w, Q.low)$, count the number of points in e_p because $\forall p \in e_p, \forall q \in Q$, $f(w, q) > f(w, p)$ holds.
- *Rule* 2. (MBR in $AboveQ$) If $f(w, e_p.low) > f(w, Q.up)$, then discard e_p because $\forall p \in e_p, \forall q \in Q$, $f(w, q) < f(w, p)$ holds.
- *Rule* 3. (MBR in InQ) If $f(w, e_i.low) > f(w, Q.low)$ and $f(w, e_i.up) < f(w, Q.up)$, then add e_i to candidate for further examination.

Algorithm 1. ARank-P

Input: $P, w, Q, minRank$
Output: include: rnk; discard: -1;

```
 1: rnk ⇐ 0, Cand ⇐ ∅
 2: heapP.enqueue(RtreeP.Root())
 3: while heapP.isNotEmpty() do
 4:    e_p ⇐ heapP.dequeue()
 5:    for each e_i ∈ e_p do
 6:       if f(w, e_i.low) < f(w, Q.up) then
 7:          if e_i in BelowQ then
 8:             rnk ⇐ rnk + e_i.size() × |Q|        //Rule 1
 9:             if rnk ≥ minRank then
10:                return -1
11:          else if e_i in InQ then
12:             Cand ⇐ Cand ∪ e_i                  //Rule 3
13:          else
14:             if e_i is a data point then
15:                Cand ⇐ Cand ∪ e_i
16:             else
17:                heapP.enqueue(e_i)
18: Refine Cand by processing the MBRs and points in Cand with each q.
19: if rnk ≤ minRank then
20:    return rnk
21: else
22:    return -1
```

ARank-P Algorithm. Given P, w, Q, and the positive integer $minRank$, the $ARank$ algorithm checks whether the aggregate rank of Q is smaller than the given $minRank$. It also returns the value of the aggregate rank when $ARank(w, Q) < minRank$. As shown by Algorithm 1, $ARank$ uses the R-tree to prune similar points in a group (MBR). In this algorithm, the counter rnk is used to count the aggregate rank of Q (Line 1). Then, the algorithm recursively checks the MBRs in the R-tree of P from the root (Line 2). If e_i belongs to $BelowQ$, the counter rnk is increased by $e_i.size() \times |Q|$ (Lines 7–8) based on Lemma 1. When rnk becomes greater than $minRank$, the algorithm returns -1 to terminate (Lines 9–10). If e_i in InQ, we add e_i into the candidate set $Cand$ for refinement (Lines 11–12). In other situations, when a leaf node of entries is encountered, the point is added into $Cand$ for refinement (Lines 14–15). Otherwise, e_i is added to the queue (Line 17). After traversal of RtreeP, refinement

Algorithm 2. Tree-Pruning Method (**TPM**)

Input: P, W, Q
Output: result set *heap*
1: initialize *heap* with first k weighting vectors and aggregate ranks of $|Q|$
2: $minRank \Leftarrow heap$'s last rank.
3: **for** each $w \in W-$ {first k element in W} **do**
4: $rnk \Leftarrow$ ARank-P$(P, w, Q, minRank)$
5: **if** $rnk \neq -1$ **then**
6: $heap.insert(w, rnk)$
7: $minRank \Leftarrow$ last rank of *heap*.
8: **return** *heap*

is performed where the $Cand$ set is checked for each $q \in Q$ and rnk is updated (Line 18). Note that $Cand$ contains both the MBR and single p in the space part of InQ. The refinement also considers the upper and lower bounds of the MBR to filter each q. Finally, rnk is the aggregate rank if $rnk < minRank$ or -1 is returned, which indicates that the current w is not a result.

TPM Algorithm. The TPM algorithm first initializes *heap* with the first k weighting vectors and their aggregate ranks of Q (Line 1). Then, for the other weighting vectors, the ARank-P Algorithm is called to check the aggregate rank of the query set Q (Line 4). If the current w can make the rank of Q better than the last rank in *heap*, this w is inserted into *heap* with its rank. Then, *heap* automatically updates itself by removing the last element and inserting a new w and aggregate rank while keeping the sorted order of rank (Line 6). Then, $minRank$ is updated by the last rank in the updated *heap* (Line 7). Eventually, the algorithm returns *heap* as the result of the aggregate reverse k-rank query.

6 Double-Tree Method (DTM)

TPM uses an R-tree to manage similar p and avoid computing with MBRs. However, TPM is limited in that it evaluates each w one by one, and its efficiency declines when the W set is large. This limitation inspired us to remove redundant computing by grouping similar w. We propose the double-tree method (**DTM**), which also indexes W set in an R-tree. The R-trees for P and W are denoted as $RtreeP$ and $RtreeW$, respectively. Figure 5 shows the three parts of $BelowQ$, InQ and $AboveQ$, which are separated by the bounds of the MBR e_w in $RtreeW$ and $Q.up$ (Q.low). Based on the MBR features in $RtreeP$ and $RtreeW$, we can obtain the score bounds of a single data point on the MBR e_w of $RtreeW$.

Lemma 2 (*Score bound of p*)*: Given an MBR with the weighting vector* e_w *in RtreeW and* $p \in P$*, the score* $f(w, p)$ *is lower-bounded by* $f(e_w.low, p)$ *and upper-bounded by* $f(e_w.up, p)$*.*

Proof. For $w \in e_w$, $\forall w[i] \geq e_w.low[i]$ holds, so $\sum_{i=1}^{d} e_w.low[i] \cdot p[i] \leq \sum_{i=1}^{d} w[i] \cdot p[i]$ hence $f(w, p) \geq f(e_w.low, p)$. Similarly, $f(w, p) \leq f(e_w.up, p)$.

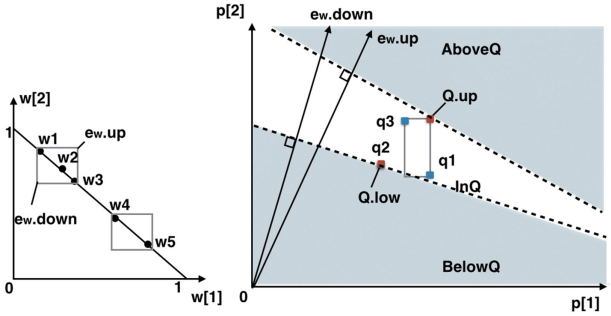


Fig. 5. The space part of BelowQ, InQ and AboveQ based on $Q.low$ and $Q.up$ with a MBR e_w in 2d space of data set P.

The score bounds of the MBR e_p of $RtreeP$ based on e_w of $RtreeW$ can also be inferred from the following lemma.

Lemma 3 (*Score bound of MBR*): *Given the MBR e_w of RtreeW and MBR e_p of RtreeP, the score of every $p \in e_p$ is lower-bounded by $f(e_w.low, e_p.low)$ and upper-bounded by $f(e_w.up, e_p.up)$.*

Proof. For $p \in e_p$, $\forall i, p[i] \leq e_p.low[i]$ holds based on the proof in Lemma 2, so $\sum_{i=1}^{d} e_w.low[i] \cdot e_p.low[i] \leq \sum_{i=1}^{d} e_w[i].low \cdot p[i] \leq \sum_{i=1}^{d} w[i] \cdot p[i]$. Hence, $f(w,p) \geq f(e_w.low, e_p.low)$. Similarly, $f(w,p) \leq f(e_w.up, e_p.up)$ holds.

Based on the above lemmas, we can build the bounds of the aggregate rank for Q on the MBR e_w.

Theorem 2 (*Aggregate rank bounds of Q for e_w*): *Given the set of query points Q and the MBR of the weighting vector e_w, the lower bound of rank for every $w \in e_w$ is $|Q| \times rank(e_w.low, Q.low)$, and the upper bound of $ARank(w, Q)$ is $|Q| \times rank(e_w.up, Q.up)$.*

Proof. This is similar to the proof for Lemma 1.

The ARank-P algorithm checks the rank of Q with the single w. This time, we propose using ARank-WP to check a group of w, e_w. For e_w, Algorithm 3 helps check these $w \in e_w$ with Q and $minRank$. The algorithm returns 1 if all $w \in e_w$ make the Q rank in $minRank$ and returns -1 if none of $w \in e_w$ makes Q rank better than $minRank$. The algorithm returns 0 if it needs to check the child entries of e_w.

Unlike the TPM algorithm in Sect. 5, DTM uses two R-trees to index the P and W. Hence, it can prune both the weighting vectors and points. Algorithm 4 starts from the root of $RtreeW$ and calls Algorithm 3 to check the aggregate rank of Q on e_w (Line 9). If $flag$ is 0, all child MBRs are added to $heapW$ for the next loop (Lines 10–11). If $flag$ is 1, this means that every w in e_w makes Q rank better than $minRank$. Thus, we can call Algorithm 1 to compute the

Algorithm 3. ARank-WP

Input: $P, e_w, Q, minRank$
Output: include: 1; discard: -1; uncertain : 0;

$rnk \Leftarrow 0$, $Cand \Leftarrow \emptyset$
$heapP.enqueue(RtreeP.root())$
while $heapP.isNotEmpty()$ **do**
  $e_p \Leftarrow heapP.dequeue()$
  for each $e_i \in e_p$ **do**
    if $f(e_w.low, e_i.low) < f(e_w.up, Q.up)$ **then**
      if e_i in $BelowQ$ **then**
        $rnk \Leftarrow rnk + e_i.size() \times |Q|$
        if $rnk \geq minRank$ **then**
          return -1
      else if e_i in InQ **then**
        $Cand \Leftarrow Cand \cup e_i$
      else
        if e_i is a data point **then**
          $Cand \Leftarrow Cand \cup e_i$
        else
          $heapP.enqueue(e_i)$
Refine $Cand$ and process the MBRs and points in $Cand$ with each q.
if $rnk \leq minRank$ **then**
  return 1
else
  return 0

Algorithm 4. Double-tree method (**DTM**)

Input: P, W, Q
Output: result set $heap$

initialize $heap$ with the first k weighting vectors and the aggregate ranks of $|Q|$
$minRank \Leftarrow heap$'s last rank.
$heapW.enqueue(RtreeW.root())$
while $heapW.isNotEmpty()$ **do**
  $e_w \Leftarrow heapW.dequeue()$
  if e_w is a single weighting vector **then**
    call the function ARank-P and update $minRank$.
  else
    $flag \Leftarrow$ ARank-WP$(P, e_w, Q, minRank)$
    if $flag = 0$ **then**
      $heapW.enqueue($all subMBR $\in e_w)$
    else
      if $flag = 1$ **then**
        for each $w \in e_w$ **do**
          call the function ARank-P and update $minRank$.
return $heap$

rank of each w in e_w and update *heap* and *minRank* (Lines 14–15). When the leaf node of a single w is being checked, Algorithm 1 is called just like in TPM (Lines 6–7). When the algorithm terminates, *heap* is returned as the result of the aggregate reverse rank query.

Table 1 summarizes the comparison of space and time complexities for NA (naive) and the proposed TPM and DTM. NA has the highest cost in terms of time complexity because $O(|P| \cdot |W|)$. However, it requires no extra index and only needs $O(k)$ space complexity. The proposed TPM and DTM algorithms need space to store the R-tree but have lower computation costs.

Table 1. Time complexity, space complexity for algorithms NA, TPM and DTM.

Algorithm	Index	Time complexity	Space complexity
NA	None	$O(\|P\| \cdot \|W\|)$	$O(k)$
TPM	RtreeP	$O(\|W\| \cdot \log \|P\|)$	$O(\log \|P\|)$
DTM	RtreeP, RtreeW	$O(\log \|W\| \cdot \log \|P\|)$	$O(\log \|P\| + \log \|W\|)$

7 Experiment

We present the experimental evaluation of the naive, TPM, and DTM algorithms for AR-k. All algorithms were implemented in C++, and the experiments were run on a Mac with 2.6 GHz Intel Core i7, 16 GB RAM. The page size was 4K.

Data Set. Both synthetic and real data were employed for the data set P. The synthetic data sets were uniform (UN), clustered (CL) and anti-correlated (AC) with an attribute value range of $[0, 1)$ that were generated as in [13,18]. We also performed comparison experiments on two real data sets: HOUSE and NBA[2]. HOUSE contains 201760 six-dimensional tuples and represents the annual payments of American families (gas, electricity, water, heating, insurance, and property tax) in 2013. NBA is a 20960-tuple data set of box scores of players in the NBA from 1949 to 2009. We extracted the NBA statistics for points, rebounds, assists, blocks, and steals to form a 5-d vector that represents a player. For data set W, we also had the UN and CL data sets, which were generated in the same manner as the data sets of P. We generated Q by using clustered data.

Experimental Results for Synthetic Data. Figure 6 shows the experimental results for the synthetic data sets (UN, CL, AC) with varying dimensions d (2–5), where both data sets P and W contained 100 K tuples. Q had five query points, and we wanted to find the five best preferences ($k = 5$) for this Q. Figures 6a–c show that TPM and DTM were at least 10 times faster than NA in terms of CPU time. DTM performed the best because it skipped checking each p and w and was stable for all dimensional cases. Tree-based methods perform less querying for CL data than other data distributions because it is easier to index clustered data with the R-tree. Figures 6d–f show that DTM had less I/O usage

[2] NBA: http://www.databasebasketball.com/; HOUSE: https://usa.ipums.org/usa/.

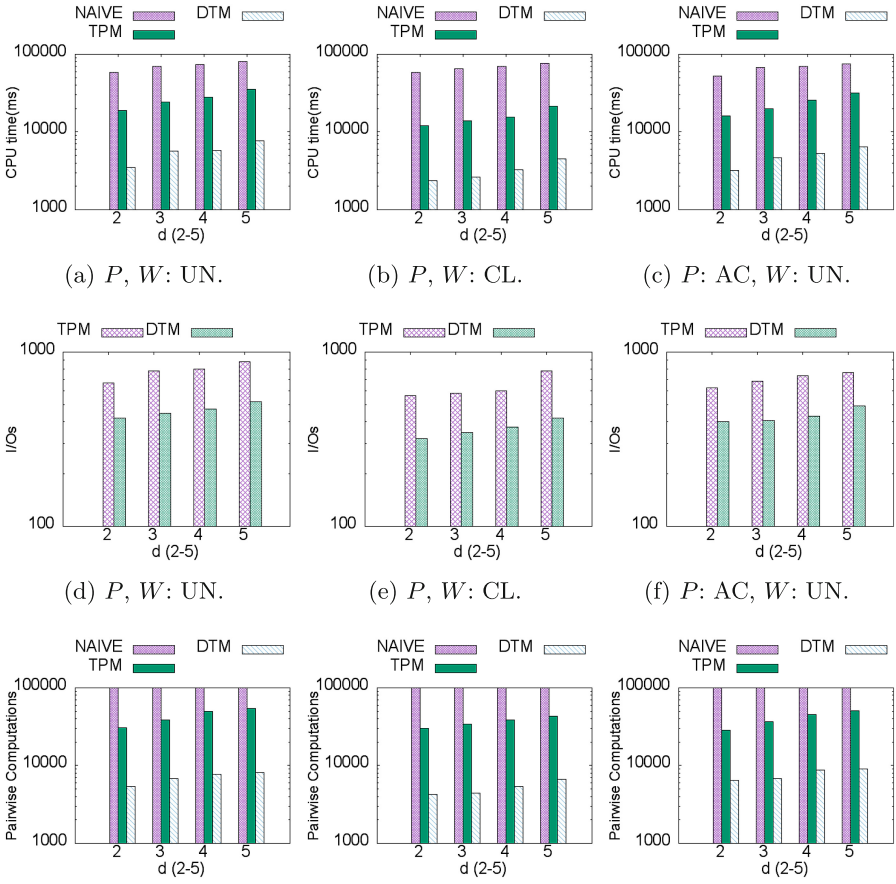
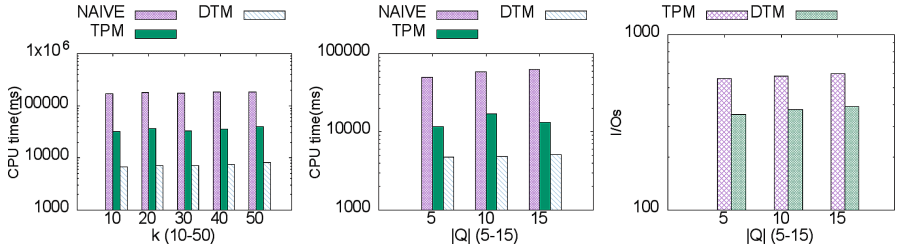


Fig. 6. Comparison results of CPU time (a, b and c), I/O cost (d, e, f) and Pairwise computations (g, h and i) on synthetic data, $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.

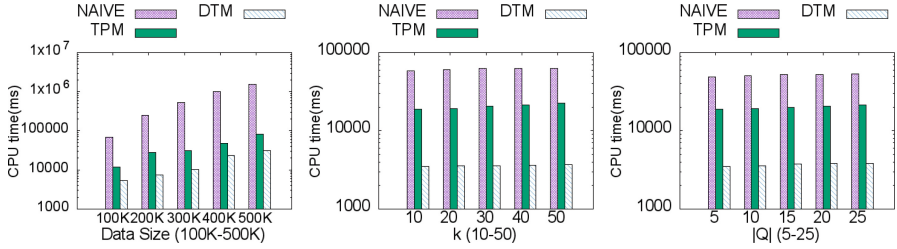
than TPM for all kinds of data. Figures 6g–i show pairwise computations with p and w for calculating the scores. DTM needed fewer computations because it can prune both points and weighting vectors with double R-trees.

Experimental Results for Real Data. Figure 7a shows the performance with the HOUSE data set and different k (10–50). DTM again performed the best. We found that the major dimensions of HOUSE were similar to an exponential distribution. The NBA data set was used to solve another practical query: who likes a team more than others? We selected five, ten, and fifteen players from the same team as Q and then generated the data set W as various user preferences. As expected, DTM found the answer the fastest. Figure 7c shows the I/O cost of the two proposed tree-base algorithms (TPM and DTM). DTM required less I/O usage.



(a) HOUSE, $|Q| = 5$. (b) NBA, $k = 10$. (c) I/O cost of NBA.

Fig. 7. Real data, HOUSE and NBA, W:UN, $|W| = 100$K, $k = 10$.



(a) Varying $|P|$,$|W|$. (b) Varying k. (c) Varying $|Q|$.

Fig. 8. Scalability on varying $|P|$,$|W|$ ($k = 5$, $|Q| = 5$, $d = 4$); varying k ($|P| = |W| = 100K$, $|Q| = 5$); varying $|Q|$ ($|P| = |W| = 100K$, $k = 5$).

Scalability. Figure 8a shows the scalable property for varying $|P|$ and $|W|$. The CPU cost of DTM increased slightly with increasing $|P|$ and $|W|$ because most pairwise computations were filtered by R-treeW and R-treeP. According to Figs. 8b and c, all of the algorithms were insensitive to k and $|Q|$ because both were far smaller in value than the cardinality of $|P|$ and $|W|$.

8 Conclusion

Reverse rank queries have become important tools in marketing analyzing. However, related research on reverse rank queries has only focused on one product. We propose the aggregate reverse rank query to address the situation of multiple query products for applying to the product bundling. We devised the TPM and DTM methods for efficient querying. TPM is a tree-based pruning method that prunes unnecessary products with the help of an R-tree. DTM uses two R-trees to manage products and user preferences and prune both of them. We compared the methods through experiments on both synthetic data and real data and the results show that DTM is the most efficient one.

As future work, we first plan to investigate approaches for other ARank functions, such as evaluating the aggregate rank by the harmonic average of each rank. We also want to consider approximate solutions for AR-k queries.

References

1. Chang, Y.-C., Bergman, L.D., Castelli, V., Li, C.-S., Lo, M.-L., Smith, J.R.: The onion technique: Indexing for linear optimization queries. In: SIGMOD Conference, pp. 391–402 (2000)
2. Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: Influence zone: efficiently processing reverse k nearest neighbors queries. In: Proceedings of the 27th ICDE 2011, pp. 577–588 (2011)
3. Dellis, E., Seeger, B.: Efficient computation of reverse skyline queries. In: Proceedings of the 33rd International Conference on VLDB, pp. 291–302 (2007)
4. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4), 11:1–11:58 (2008)
5. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: Proceedings of the ACM SIGMOD, pp. 201–212 (2000)
6. Lian, X., Chen, L.: Monochromatic and bichromatic reverse skyline search over uncertain databases. In: Proceedings of the ACM SIGMOD, pp. 213–226 (2008)
7. Stanoi, I., Agrawal, D., El Abbadi, A.: Reverse nearest neighbor queries for dynamic databases. In: ACM SIGMOD Workshop, pp. 44–53 (2000)
8. Tao, Y., Papadias, D., Lian, X.: Reverse knn search in arbitrary dimensionality. In: Proceedings of the 13th International Conference on VLDB, pp. 744–755 (2004)
9. Tao, Y., Papadias, D., Lian, X., Xiao, X.: Multidimensional reverse k NN search. VLDB J. **16**(3), 293–316 (2007)
10. Vlachou, A., Doulkeridis, C., Kotidis, Y., et al.: Reverse top-k queries. In: ICDE, pp. 365–376 (2010)
11. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Identifying the most influential data objects with reverse top-k queries, pp. 364–372 (2010)
12. Vlachou, A., Doulkeridis, C. , Kotidis, Y., et al.: Monochromatic and bichromatic reverse top-k queries, pp. 1215–1229 (2011)
13. Vlachou, A., Doulkeridis, C., Kotidis, Y., et al.: Branch-and-bound algorithm for reverse top-k queries. In: SIGMOD Conference, pp. 481–492 (2013)
14. Vlachou, A., Doulkeridis, C., et al.: Monitoring reverse top-k queries over mobile devices. In: MobiDE, pp. 17–24 (2011)
15. Yang, S., Cheema, M.A., Lin, X., Wang, W.: Reverse k nearest neighbors query processing: experiments and analysis. Proc. VLDB Endowment **8**, 605–616 (2015)
16. Yang, S., Cheema, M.A., Lin, X., Zhang, Y.: SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In: IEEE 30th International Conference on Data Engineering, ICDE, pp. 760–771 (2014)
17. Yao, B., Li, F., Kumar, P.: Reverse furthest neighbors in spatial databases. In: Proceedings of the 25th ICDE, pp. 664–675 (2009)
18. Zhang, Z., Jin, C., Kang, Q.: Reverse k-ranks query. PVLDB **7**(10), 785–796 (2014)