Weighted Aggregate Reverse Rank Queries

YUYANG DONG and HANXIONG CHEN, University of Tsukuba
JEFFREY XU YU, The Chinese University of Hong Kong
KAZUTAKA FURUSE and HIROYUKI KITAGAWA, University of Tsukuba

In marketing, helping manufacturers to find the matching preferences of potential customers for their products is an essential work, especially in e-commerce analyzing with big data. The aggregate reverse rank query has been proposed to return top-*k* customers who have more potential to buy a given product bundling than other customers, where the potential is evaluated by the aggregate rank, which is defined as the sum of each product's rank. This query correctly reflects the request only when the customers consider the products in the product bundling equally. Unfortunately, rather than thinking products equally, in most cases, people buy a product bundling because they appreciate a special part of the bundling. Manufacturers, such as video games companies and cable television industries, are also willing to bundle some attractive products with less popular products for the purpose of maximum benefits or inventory liquidation.

Inspired by the necessity of general aggregate reverse rank query for unequal thinking, we propose a weighted aggregate reverse rank query, which treats the elements in product bundling with different weights to target customers from all aspects of thought. To solve this query efficiently, we first try a straightforward extension. Then, we rebuild the bound-and-filter framework for the weighted aggregate reverse rank query. We prove, theoretically, that the new approach finds the optimal bounds, and we develop the highly efficient algorithm based on these bounds. The theoretical analysis and experimental results demonstrated the efficacy of the proposed methods.

CCS Concepts: • **Information systems → Database query processing**; • **Theory of computation → Data structures and algorithms for data management**;

Additional Key Words and Phrases: Database, query processing, spatial index, aggregate reverse rank query

ACM Reference format:
Yuyang Dong, Hanxiong Chen, Jeffrey Xu Yu, Kazutaka Furuse, and Hiroyuki Kitagawa. 2018. Weighted Aggregate Reverse Rank Queries. *ACM Trans. Spatial Algorithms Syst.* 4, 2, Article 5 (August 2018), 23 pages.
https://doi.org/10.1145/3225216

1 INTRODUCTION

Top-*k* query is a user-view model that can help customers find their favorite products and is widely used in many practical applications [2, 10, 11]. For two different datasets, user preferences, and products, this model retrieves the top *k* products matching a given user preference. Conversely, manufacturers must also target potential customers for their products. Reverse *k*-rank query (*RkR*)

This work is partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of RIKEN, Japan.
Authors' addresses: Y. Dong, H. Chen, K. Furuse, H. Kitagawa, Department of Computer Science, University of Tsukuba, Japan; emails: touuyou@gmail.com, {chx, furuse, kitagawa}@cs.tsukuba.ac.jp; J. Xu Yu, Department of System Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong, China; email: yu@se.cuhk.edu.hk.

User (w)	w[price]	w[rating]
Tom	0.8	0.2
Jerry	0.3	0.7
Spike	0.1	0.9

Book (p)	p[price]	p[rating]
p_1	0.6	0.7
p_2	0.2	0.3
p_3	0.1	0.6
p_4	0.7	0.5
p_5	0.8	0.2

(a) User preferences and books.

	Rank (score) on Tom	Rank (score) on Jerry	Rank (score) on Spike	RKR Result (k=1)
p_1	3 (0.62)	5 (0.67)	5 (0.69)	Tom
p_2	2 (0.22)	1 (0.27)	2 (0.29)	Jerry
p_3	1 (0.20)	3 (0.45)	4 (0.55)	Tom
p_4	4 (0.66)	4 (0.56)	3 (0.52)	Spike
p_5	5 (0.68)	2 (0.38)	1 (0.26)	Spike

(b) Rank, score and reverse 1-rank result for each book.

Bundle	ARank on Tom	ARank on Jerry	ARank on Spike	ARR Result (k=1)
p_1,p_2	5 (3+2)	6 (5+1)	7 (5+2)	Tom
p_4,p_5	9 (4+5)	6 (4+2)	4 (3+1)	Spike

(c) Aggregate rank and aggregate reverse 1-rank result for each bundled books.

Fig. 1. Example of *RkR* and *ARR* queries.

[32], a manufacturer-view query model, solves this problem by returning the top k user preferences for a given product. Another manufacturer-view query, aggregate reverse rank query (*ARR*) [6] addressed a limitation of *RkR* query, allowing it to retrieve user preferences for a set of products and help manufacturers with product bundling.

An example of *RkR* and *ARR* queries is shown in Figure 1. Five different books (p_1–p_5) are scored on the attributes "price" and "rating." The preferences of three users (Tom, Jerry, and Spike), consist of the weights for all attributes of the book. The score of a book w.r.t. a user is found by the inner product value of the book attribute and user preference vectors (Figure 1(b)). The results of the reverse 1-rank query are given in the last cells of Figure 1(b)[1]. For example, Jerry believes that p_2 is the best book, while Tom and Spike think that it is the second best. Jerry is more likely to buy p_2

[1]Without loss of generality, we assume that minimum values are preferable in this research.

than Tom and Spike are, based on this ranking; hence, the reverse 1-rank query returns Jerry as a result. The table at Figure 1(c) shows an example of *ARR* query for product bundling. In this case, two books (p_1 and p_2) are bundled as a set for sale. *ARR* query evaluates aggregate rank (*ARank*) with the sum of each book, so the bundle's rank is $3 + 2 = 5$ based on Tom's preference. This *ARR* query returns Tom as its result ($k = 1$) because Tom would rank a bundle of p_1 and p_2 higher than others would.

1.1 Definitions

Assumptions of the product and preference databases, and the score function between them, are in line with those in related research [6, 24, 26, 32]. The querying problems are based on a user-product model that has two types of database: product dataset P and user preference dataset W. Each product $p \in P$ is a d-dimensional vector that contains d non-negative scoring attributes. Therefore, p is represented as a point $p = (p[1], p[2], \ldots, p[d])$, where $p[i]$ is the attribute value of p in the ith dimension. 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 of product p based on preference w is defined as the inner product of p and w, which is expressed by $f(w, p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. All products are ranked by their scores, with a minimum score preferred. A query product is denoted as q, which is in the same space as, but not necessarily an element of P.

For a specific w, the rank of q is defined as the number of products whose score is less than q's.

Definition 1 ($rank(w, q)$). Given a product dataset P, a preference w, and a query q, the rank of q according to w is $rank(w, q) = |S|$, where $S \subseteq P$ and $\forall p_i \in S, f(w, p_i) < f(w, q) \wedge \forall p_j \in (P - S), f(w, p_j) \geq f(w, q)$.

Reverse k-rank query [32] can retrieve top k preferences that improve q's rank over others.

Definition 2 (*reverse k-ranks query, RkR*). Given a product dataset P and preference dataset W, positive integer k, and query q, 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.

Aggregate reverse rank query [6] extends *RkR* query to handle queries with more than one query point.

Definition 3 (*aggregate reverse rank query,* ARR). Given datasets P and W, positive integer k, and query product set Q, *ARR* 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.

The aggregate evaluation functions ARank were considered for finding the sum of each rank(w,q): $ARank(w, Q) = \sum_{q_i \in Q} rank(w, q_i)$.

1.2 Motivation for Weighted Aggregate Reverse Rank Query

ARR query [6] is an essential tool for finding potential customers for a given product bundle. However, the aggregate rank function (*ARank*), which can be used to evaluate the bundled products, is a simple sum operation ($ARank(w, Q) = \sum rank(w, q), q \in Q$). This is a limitation of *ARR* query because the summing function is only one scenario in which ranks are evaluated in real-world applications; neither customers nor manufacturers consider every product in the bundle to be equal in most situations. Many customers buy a product bundle because they want some subset of the products, and they may not care much about the others. Sellers also make use of this point, bundling unpopular products with attractive ones to maximize profits or liquidate inventories. For example, Xiaomi[2], a famous cell phone company in China, always bundles a newly released cell

[2]http://www.mi.com/.

Weights (p_1,p_2)	Weighted ARank on Tom	Weighted ARank on Jerry	Weighted ARank on Spike	WARR Result (k=1)
(0.5, 0.5)	2.5	3	3.5	Tom
(0.2, 0.8)	2.2	1.8	2.6	Jerry

Fig. 2. *WARR* query results for a bundle of p_1 and p_2 with different weights.

phone with some accessories (mobile battery, cell phones case, earphone, etc.); customers have to accept this product bundle if they want to obtain the new phone upon its release. Therefore, it is more accurate to add weights for different products in the bundle in the market analysis of interested consumers. In addition to e-commerce, user-product-based *ARR* queries can also be applied to other fields. Reference [6] uses *ARR* query with NBA player statistics to analyze preferences for a given NBA team (of several players). In business investment, start-up teams (of several members) would like to know which angel investor is most willing to invest in them. In these cases, adding weights to different roles on an NBA or start-up team is also reasonable.

The model is closer to reality if weights are assigned for the *ARR* query. Inspired by this, we generalize the *ARR* query and then propose a new query problem called weighted aggregate reverse rank query (*WARR*). *WARR* query uses weights to handle the rank value of each product so that it can find customers using their different perspectives on products in the bundle. In this extension, the summing *ARank* function in the *ARR* query becomes a specific situation in which the weights are equal to each other ($\alpha_i = \frac{1}{|\alpha|}, i = 1, 2, \ldots, |\alpha|$). More specifically, the weights $\alpha_i \in \alpha$ where $\sum_{i=1}^{|\alpha|} \alpha_i = 1$ correspond to the query products; note that this weighted preference is different from the user preference w on attributes, e.g., α_1 is the weighted value for q_1 in Q. These weights can adjust the rank of query products with a weighted *ARank* function (*WARank*): $WARank(w, Q, \alpha) = \sum \alpha_i \times rank(w, q_i), q_i \in Q, \alpha_i \in \alpha$. Sellers and data analysts from the manufacturer can use *WARR* query to analyze various marketing positions by testing product bundles with different weights.

Figure 2 shows a *WARR* query example for $\{p_1, p_2\}$ with two different weights: $\{0.5, 0.5\}$ and $\{0.2, 0.8\}$. As Figure 2(d) shows, Tom's weighted rank of $\{p_1, p_2\}$ is $3 \times 0.5 + 2 \times 0.5 = 2.5$ when $\alpha = \{0.5, 0.5\}$ and $3 \times 0.2 + 2 \times 0.8 = 2.2$ when $\alpha = \{0.2, 0.8\}$. As with the *ARR* query, Tom is also the result when the weights are equal. However, for weights $\{0.2, 0.8\}$, we want to find customers who consider p_2 to be the main reason to buy this bundle. In this case, Jerry's weighted aggregate rank value is $5 \times 0.2 + 1 \times 0.8 = 1.8$, which is the best rank among the three people; hence, *WARR* query returns Jerry as its result. In this query process, *WARR* helps manufacturers target Jerry, who treats p_2 as a main priority.

As a generalized version, *WARR* follows the previous work of *ARR*, which was a manufacturer-view query processing. Therefore, the weights for the query products in *WARR* query are assigned by the manufacturers. If we allow customers to set weights on the products, for each customer, we need to know her preferences for such a huge number of products and maintain them. Moreover, when a manufacturer releases a new product, it means a new vector be added to P in our model. However, it also means to add $|W|$ vectors to update the preference to all customers for allowing them to assign weights to products. In our model, it makes sense to let manufacturers assign weights to their products then issue a *WARR* query on these products. Specifically, to analyze the target customers with the variety considered on bundled products, a manufacturer can adjust different weights and issue different *WARR* queries, then *WARR* retrieves the target users under these weighted preferences. Using the example of Figure 2, assuming that the seller of the bookshop wants to use the book p_2 as the main product for sale, she sets the weight as (0.2, 0.8) corresponding to $\{p_1, p_2\}$. Then, *WARR* can help to return Jerry as the best target since his weighted aggregate rank is the best.

1.3 Challenges and Solutions

There are two challenges in efficiently processing a *WARR* query.

The first challenge is to solve *WARR* queries with the *bound-and-filter* framework. *WARR* query is a complicated processing; it requires to evaluate the rank of each query products with respect to all users and returns the top-k users. The basic Naive algorithm for *WARR* leads a huge computation with a complexity of $O(|P| \cdot |W| \cdot |Q|)$. The key point is to reduce the pairwise computation between P and W. The most relevant work is the *ARR* query in Ref. [6], which offers the solutions that bound query products Q with two dummy points and utilizes the spatial index R-tree to filter data with the divide and conquer methodology (The details are introduced in Section 4.1). We formalize the techniques in Ref. [6] as the *"bound-and-filter"* framework; it is an efficient strategy that reduces the computational complexity to $O(log|P| \cdot log|W|)$. *WARR* is a generation of the previous work *ARR* and has much more complicated processing. As the example in Figure 2 demonstrates, the value of the ranking changes with differing weights. Therefore, the techniques in Ref. [6], which was designed only for the products that are equal, cannot handle the weighted ranking relationship, hence, cannot be extended to solve *WARR* query effectively.

To this, we design sophisticated bounding methods and filtering strategies for the weighted ranking, and propose an extended filtering method (EFM) solution that is based on the bound-and-filter framework (Section 4). Specifically, for the bounding phase, we propose weighted aggregate rank bounds to bound Q safely. We propose a novel early stopping strategy, which takes into consideration the weights and helps provide more efficient filtering.

The second challenge is the optimization of the bounding phase in the bound-and-filter framework. The bound of the queries Q is the core of the efficient processing, since it determines the amount of the filtering data for both P and W in the filtering phase, hence, significantly affects the performance.

To this, we proposed an optimal bounding method (OBM) (Section 5), which finds the optimal bounds for an arbitrary Q then filters more data than EFM. We proved the optimal bounding with the theory of linear programming. It is important to note that the proposed optimal bounds are effective also in previous *ARR* query but it helps to enhance the performance remarkable in *WARR*.

1.4 Contributions

This article makes the following contributions:

— We define a new query, called weighted aggregate rank query (*WARR*), that extends the previous aggregate rank query by adding weights for different products in a bundle. With a variety of weights, *WARR* can analyze and target different types of potential customers for a given product bundle.

— We develop three solutions to process *WARR* queries, as existing approaches cannot be directly applied, called Straight Forward Method (SFM), Extended Filtering Method (EFM), and Optimal Bounding Method (OBM). SFM is a straightforward method that uses a spatial R-tree. EFM adapts the bound-and-filter framework of Ref. [6] to the additional weights in the *WARR* query. We study this filtering space in the bound-and-filter framework and propose an optimal bounding approach that is proven to find the tightest bound of Q. The OBM is based on this optimal bound in bound-and-filter framework. This optimal bounding strategy can also adjust into the previous *ARR* query.

— We conduct a thorough experimental evaluation of real-world and synthetic datasets to evaluate the efficacy of the proposed algorithms.

The remainder of this article is organized as follows: Section 2 summarizes related work. Section 3 formally defines the proposed *WARR* query and gives a straightforward solution. Two additional novel solutions are given in Sections 4 and 5. Experimental results are reported in Section 6, and Section 7 concludes the article.

2 RELATED WORK

Currently, in the user-product model, the ranking is an important technique for evaluating a product based on user preferences. In marketing analysis, such as identifying competing products or targeting potential customers, many variants of rank-aware queries have been widely researched.

Ranking query (top-k query). Many applications are designed for returning a limited set of ranked products on individual user preferences, the most basic of which is the top-k query. Here, we summarize some important work in ranking queries. Chang et al. [2] proposed the Onion technique to pre-process and index data points in layers with convex hulls for linear optimization queries; the onion-based index can help to filter data and compute efficiently. A tree-based index approach, which processes the top-k queries with a branch-and-bound methodology, has been studied in Ref. [20]. Fagin's algorithm [8] and the threshold algorithm (TA) [9] were proposed to compute top-k queries over multiple sources, where each source has only a subset of attributes. Other variants of the threshold-based algorithms for top-k queries were investigated in Refs [1], [3], and [14] . Reference [11] is an important study that describes and classifies top-k query processing techniques in relational databases.

Reverse rank query. The converse of rank queries, called reverse rank queries, have been studied extensively. Reverse rank queries evaluate the rank of a query product based on user preferences and retrieve the top k users. One of these, reverse top-k query, has been proposed in order to find users who treat a query product as their top k product [24, 25]. For an efficient reverse top-k process, Vlachou et al. [26] proposed a branch-and-bound algorithm using a tree-based method with boundary-based registration. Vlachou et al. [23, 27] have reported various applications of reverse top-k queries. However, Zhang et al. [32] indicated that reverse top-k query always returns an empty set for less-popular products. To ensure 100% coverage for any given query product, Ref. [32] proposed the reverse k-rank query to find the top-k user preferences with the highest rank for a given object among all users. Reference [7] gives a Grid-index algorithm for efficient processing these reverse rank queries with high-dimensional data. Reference [6] is the most related to our work; Dong et al. indicated that both reverse top-k and reverse k-rank queries were designed for only one product and cannot handle the product bundling. Similar to the cable television industry, which bundles channels, companies use product bundling as a common market strategy. Reference [6] defined an aggregate reverse rank query that finds the top-k users for a given bundled product, then proposes tree-based algorithms (Tree Pruning Method (TPM), Double Tree Method (DTM)) to process them efficiently.

Other reverse and aggregate queries. In contrast to the nearest neighbor search in metric space, Korn and Muthukrishnan [12] proposed the reverse nearest neighbor (RNN) query. For reverse k-nearest neighbor (RKNN) queries, Yang et al. [29] carried out an in-depth investigation that analyzed and compared notable algorithms in Refs [4], [19], [21], [22], and [30]. RKNN differs from reverse rank queries because it evaluates the relative L_p distance (Euclidean distance) between two points in metric space. However, the reverse rank queries focus on absolute ranking among all objects and use the inner product function to compute scores. Additionally, all data are the same kind of points in one space in RKNN, while the user-product model of reverse rank queries demands that users and products are two datasets of different data spaces. Besides nearest neighbor, Yao et al. [31] proposed reverse furthest neighbor (RFN) queries to find points in which the query point is deemed to be the furthest neighbor. Wang et al. [28] extended the RFN to RkFN

Table 1. Symbols and Notation

Symbols	Description
P	Product dataset.
W	Preferences dataset.
Q	Query products.
α	Weights for Q.
d	Data dimensionality.
$f(w,p)$	The score of p based on w with inner product.
$p[i]$	Value of a product p in the ith dimension.
$H(w,q)$	The (d-1) dimensional hyper-plane perpendicular to w and cross q.
MBR	Minimum bounding rectangle.
ep (ew)	An MBR in Rtree of data set P (W).
$L[MBR], U[MBR]$	The lower-left and upper-right corners in an MBR.
Q_l (Q_u)	The set of $q_l^{(i)}$ ($q_u^{(i)}$) in all d dimensions.
Q^{low}, Q^{up}	Bounding of Q in Ref. [6].
$Q_{opt}^{low}, Q_{opt}^{up}$	The Optimal bounding of Q in this article.

queries for an arbitrary value of k and proposed an efficient filter in the search space. Reverse skyline query returns a user based on the dominance of competitors' products [5, 13]. The preference of a user is described as an ideal product point in this query. However, preferences are represented as weighting vectors in reverse rank query. Tao et al. [17, 18] developed aggregate nearest neighbor search (ANN) to retrieve points with the smallest aggregate distance to multiple query points in metric space.

3 WEIGHTED AGGREGATE REVERSE RANK QUERY

As previously mentioned, it is necessary to evaluate each query product with different weights. Here, we define the function of weighted aggregate rank (*WARank*), then propose a new query to extend the previous *ARR* query with the *WARank* function, namely a weighted aggregate reverse rank query.

Definition 4 (WARank(w,Q,α)) Given a dataset P, preference data w, query set Q, and weights α, where $\forall\alpha_i > 0$ and $\sum_{i=1}^{|\alpha|}\alpha_i = 1$, the $WARank$ of Q based on w is $WARank(w,Q,\alpha) = \sum\alpha_i \times rank(w,q_i), q_i \in Q, \alpha_i \in \alpha$.

Definition 5 (*weighted aggregate reverse rank query,* WARR). Given datasets P and W, positive integer k, and query product set Q, the *WARR* query returns set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $WARank(w_i,Q,\alpha) \leq WARank(w_j,Q,\alpha)$ holds.

3.1 Straightforward Filtering Method (SFM)

The naive *WARR* query processing algorithm calculates *WARank(w, Q, α)* for each preference $w \in W$ by comparing all scores of $f(w,p)$, $p \in P$ with all scores of $f(w,q)$, $q \in Q$. Hence, the naive algorithm is a triple-nested loop with complexity $O(|W|\cdot|P|\cdot|Q|)$. To overcome this inefficiency, we first explain an SFM that filters dataset P.

The geometric view of a 2-dimensional example that ranks a single q based on preference vector w ($rank(w,q)$) is shown in Figure 3. There is a $(d-1)$ dimensional hyperplane denoted by $H(w,q)$, which is a line in the 2-dimensional example of Figure 3 that crosses q and perpendicular to w. The $rank(w,q)$ is equal to the number of p_i enclosed in the half-space (the gray area) defined by the hyperplane $H(w,q)$. Many previous works [6, 24, 25, 32] have used a tree-base methodology

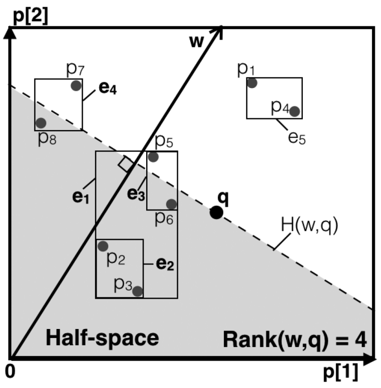


Fig. 3. Geometric view of the rank of q and a tree-based methodology.

ALGORITHM 1: Straightforward Filtering Method (SFM)

```
1: Let $T$ denote an array to record the $WARank$ with each $w$.
2: for each $w_i \in W$ do
3:     for each $q_j \in Q$ do
4:         Tree-based filtering and get $rank(w_i, q_j)$
5:         $T[w_i] \leftarrow T[w_i] + \alpha_j \cdot rank(w_i, q_j)$
6: return top-$k$ elements in $T$.
```

to find the number of points in half-space efficiently; in particular, R-trees are used to index and filter dataset P. An R-tree can group nearby points with a minimum bounding rectangle (MBR) to filter data at the lower-left and upper-right borders. For example, the upper-right border of MBR e_2 is in the half-space. Therefore, all points contained by e_2 should be counted for the rank of q. On the contrary, p_1 and p_4 should be discarded, since the lower-left border of MBR e_5 is not in the half-space. The MBRs are checked recursively while traversing the R-tree. We propose a straightforward method, called SFM, based on this technique. As Algorithm 1 shows, SFM uses the tree-based method on P to process *WARR* straightforwardly.

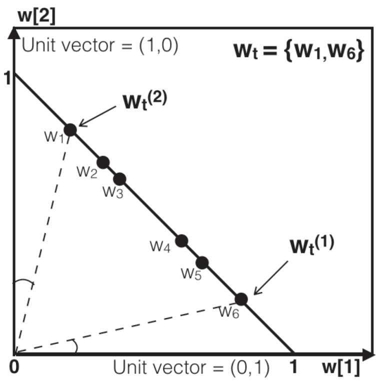
4 EXTENDED FILTERING METHOD (EFM)

The SFM solution sums the ranks for $q \in Q$ individually against each $w \in W$, which is inefficient, especially when the cardinality of W and Q are large. Reference [6] proposed an efficient bound-and-filter framework to bound Q to avoid checking every q, then filter W and P by implementing a tree-based method with Q's bounds. Unfortunately, this technique cannot handle *WARR* queries. Inspired by this point, we extend and adapt the bound-and-filter framework for the weights of ranks in *WARR* query.

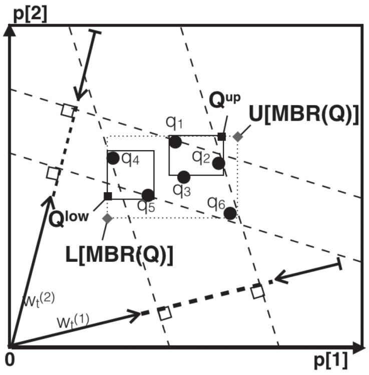
4.1 Bound-and-Filter Framework

Here, we describe the previous approach for ARR query briefly, the bound-and-filter framework. The full details can be found in Ref. [6]. There are two phases to this framework: *bounding* and *filtering*.

1. Bounding. In the bounding phase, a d-element subset of W is needed, denoted by $W_t = \{w_t^{(i)}\}_1^d$. The ith element of W_t is denoted as $w_t^{(i)}$ and is the most similar (in cosine similarity)



(a) Find W_t.



(b) Find Q^{up} and Q^{low}.

Fig. 4. Bounding phase. (a) Finding the W_t set of top-w in each dimension. (b) Using W_t to find the upper bound Q^{up} and lower bound Q^{low} of Q.

to the orthonormal basis vector (e_i) of the ith dimension ($w_t^{(i)} \in W$ and $\forall w \in W, cos(w_t^{(i)}, e_i) \geq cos(w, e_i)$). As Figure 4(a) shows, $w_t^{(1)} = w_6$ because w_6 is the closest vector to the orthonormal basis vector (0,1). In same way, $w_t^{(2)} = w_1$, therefore $W_t = \{w_6, w_1\}$. The two bounds of Q, denoted as Q^{up} and Q^{low}, are found using the following rules:

$$Q_u = \{\arg\max_{q \in Q} f(w_t^{(i)}, q)\}_{i=1}^{d} \tag{1}$$

$$Q_l = \{\arg\min_{q \in Q} f(w_t^{(i)}, q)\}_{i=1}^{d} \tag{2}$$

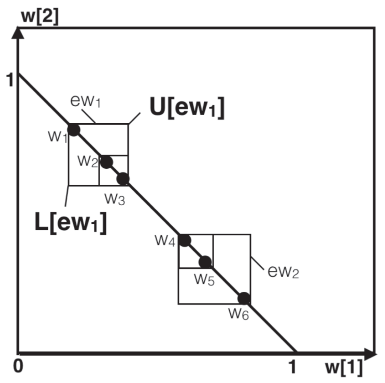
$$Q^{up} = U[MBR(Q_u)] \; and \; Q^{low} = L[MBR(Q_l)] \tag{3}$$

In the geometric view shown in Figure 4(b), q_1 and q_2 are the first points touched (maximum value) while sliding perpendicular lines of $w_t^{(2)}$ and $w_t^{(1)}$ from infinity to 0 in parallel. Then, Q^{up} defined as the right-up border of MBR of {q_1, q_2}, denoted as $U[MBR(Q_u)]$ in Equation (3). Q^{low} is similarly found by sliding the perpendicular lines from 0 to infinity. Compared with the basic bounds from $MBR(Q)$, the bounds formed by Q^{up} and Q^{low} are tighter.

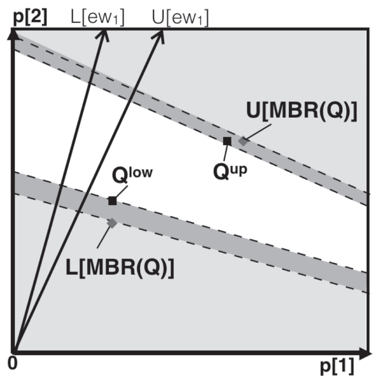
2. Filtering. In the filtering phase, both W and P are indexed independently with an R-tree, as in $R\text{-}tree_p$ in Figure 3 and $R\text{-}tree_w$ in Figure 5(a). As Figure 5(b) shows, for an MBR ew_1 of $R\text{-}tree_w$, its upper bound $U[ew_1]$ and Q^{up} form the upper hyperplane $H(U[ew_1], Q^{up})$, while its lower bound $L[ew_1]$ and Q^{low} form the lower $H(L[ew_1], Q^{low})$. The only data space of P that must be computed is the space sandwiched between the two hyperplanes; hence, the gray area in the figure is the filtering space. For every ew_i or single w in the leaf node of $R\text{-}tree_w$, Q^{up} and Q^{low} can filter more data than $MBR(Q)$ since the bounding is tighter. Figure 5(c) shows how W is filtered; a threshold value denoted as *minRank* is updated with the kth smallest rank while processing, since we only want the top-k w's. The MBRs of w's whose lower bounds rank greater than *minRank* will be filtered, such as ew_4 and ew_5.

4.2 Re-Building the Bound-and-Filter Framework in WARR Query

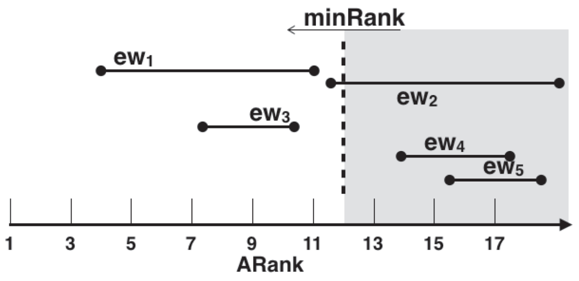
Next, we start to introduce a solution for the WARR query, which re-builds the bound-and-filter framework.



(a) Index W with R-tree.



(b) Filtering space (gray area).



(c) *minRank* and rank bounds of MBRs.

Fig. 5. Filtering phase. (a) R-tree of W. (b) Comparison of the filtering space of P between basic MBR bounding and Q^{up} (Q^{low}) bounding. (c) Filtering data W with rank bounds and *minRank*.

The intrinsic reason why the bound-and-filter framework in Ref. [6] cannot solve *WARR* queries is that the filtering phase uses the rank of Q^{low} (Q^{up}) to indicate the lower (upper) bound of $ARank(w, Q)$. In particular:

$$rank(L[ew], Q^{low}) \cdot |Q| \leq ARank(w, Q) \leq rank(U[ew], Q^{up}) \cdot |Q|. \tag{4}$$

If the lower bound of ew is still greater than the kth smallest w's *ARank* that have been checked, no $w \in ew$ is in the top-k w's and ew should be discarded.

Obviously, this does not work for *WARR* rank query, which has a series of weights for Q. In order to bound the $WARank(w, Q, \alpha)$ for an arbitrary α, we multiply the left side of Inequality (4) by the minimum value in α, denoted as α_{min}, and multiply the right side by the maximum value, α_{max}. This adaptation bounds the weighted aggregate rank as Inequality (5) based on the following lemma:

$$\alpha_{min} \cdot rank(L[ew], Q^{low}) \cdot |Q| \leq WARank(w, Q) \leq \alpha_{max} \cdot rank(U[ew], Q^{up}) \cdot |Q|. \tag{5}$$

Lemma 1 (Weighted aggregate rank bounds of Q for ew). *Given a set of query points Q, an MBR of w's, and a set of weights α for Q, the lower bound of* WARank(w, Q, α) *is* $|Q| \cdot rank(L[ew], Q^{low}) \cdot \alpha_{min}$ *and the upper bound of* WARank(w, Q, α) *is* $|Q| \cdot rank(U[ew], Q^{up}) \cdot \alpha_{max}$*, where α_{min} and α_{max} are the minimum and maximum values in α.*

Proof. $\forall q_i \in Q$, $\forall w_i \in ew$, it holds that $f(w_i, q_i) \geq f(L[ew], Q^{low})$; hence, $rank(w, q_i) \geq rank(L[ew], Q.low)$. By definition, because α_{min} is the minimum value in α, $WARank(w, Q, \alpha) = \sum$

$rank(w, q_i) \cdot \alpha_i \geq |Q| \cdot rank(L[ew], Q.low) \cdot \alpha_{min}$. Similarly, $|Q| \cdot rank(U[ew], Q.up) \cdot \alpha_{max}$ is the upper bound of $WARank(w, Q, \alpha)$. □

4.3 Early Stopping Strategy

To further enhance the performance, we propose a novel early stopping strategy that reduces the computations while processing *WARR* queries.

If the lower bound of rank given by Inequality (5) cannot filter the e_w directly, it is necessary to check the weighted rank for all $w \in e_w$ and $q \in Q$. To reduce computations, we can reuse the value of $rank(w, Q^{low})$, which has been calculated in Lemma 1, to figure out the exact value of weighted rank. The correctness of the re-using is introduced and proved in the following lemma:

Lemma 2 (Correctness of computed weighted aggregate rank with Q^{low}). *Given a set of query points Q, Q^{low}, w, and a set of weights α, the weighted aggregate rank of Q in w is equal to* $rank(w, Q^{low}) \cdot |Q| + \sum(rank(w, q_i) - rank(w, Q^{low})) \cdot \alpha_i$

Proof. $rank(w, Q^{low}) \cdot |Q| + \sum(rank(w, q_i) - rank(w, Q^{low})) \cdot \alpha_i = rank(w, Q^{low}) \cdot |Q| + \sum rank(w, q_i) \cdot \alpha_i - rank(w, Q^{low})) \cdot |Q| \cdot \sum \alpha_i$. Because $\sum \alpha_i = 1$, the result is $\sum rank(w, q_i) \cdot \alpha_i$, which is the $WARank(w, Q, \alpha)$. □

Before the final computation in Lemma 2, we can first check Q with $L[ew]$ and $U[ew]$ to determine further bounds before getting through all $w \in ew$. While processing, if the current rank becomes greater than the threshold *minRank*, we can early stop this process to avoid further checking. The details are given in Corollary 3:

Corollary 3. *Based on Lemmas 1 and 2, it can be inferred that:* $|Q| \cdot rank(L[ew], Q^{low}) \cdot \alpha_{min} \leq rank(w, Q^{low}) \cdot |Q| + \sum(rank(L[ew], q_i) - rank(L[ew], Q^{low})) \cdot \alpha_i \leq rank(w, Q^{low}) \cdot |Q| + \sum(rank(w, q_i) - rank(w, Q^{low})) \cdot \alpha_i$. *The further upper bound is calculated in a similar manner.*

Finally, Corollary 3 and Lemma 2 form an early stopping strategy that can terminate the algorithm and avoid unnecessary computation when checking the weighted rank for all $w \in e_w$ and $q \in Q$.

4.4 EFM Algorithm

We proposed the EFM algorithm based on Lemma 1, Lemma 2, and Corollary 3. Algorithm 3 shows the details of EFM. A k-element *buffer* keeps the top-k w's and is initialized to store the first k w's $\in W$ and their weighted aggregate ranks (Line 1). *Lrank* is the counter that records the lower bound rank. *minRank* is the threshold value. First, in the bounding phase, we determine the bounds Q^{low} and Q^{up} of Q (Line 4). Then, $heap_w$ and $heap_p$ help to traverse the $R\text{-}tree_p$ and $R\text{-}tree_w$. For each *ew* obtained from $heap_w$ (Line 6), we traverse $R\text{-}tree_p$ (Lines 10–20). If an *ep* located below the lower hyperplane $H(\text{L[ew]}, Q^{low})$, we count the number of $p \in ep$ and update *Lrank* using Lemma 1 (Lines 12–14). We stop and check the next $ep \in heap_p$ when *Lrank* becomes greater than *minRank* (Lines 15–16). If *ep* is in the sandwiched space, it will be added into *Cand* for future processing (Lines 17–18). If *ep* covers the upper or lower hyperplane, its children are added into $heap_p$ (Lines 19–20). After processing all MBRs $\in heap_p$, if *Lrank* is less than *minRank*, we first check all $q \in Q$ based on Corollary 3 (Lines 21–23). When *Lrank* is still smaller than *minRank*, if *ew* is a single w, we compute the exact *WARank* based on Lemma 2 and decide whether to update *buffer* and *minRank* (Lines 24–29). Otherwise, we add the children of *ew* into $heap_w$ for the next regression (Lines 30–31). Finally, the algorithm returns *buffer*, which is the result of the *WARR* query.

ALGORITHM 2: Extended Filtering Method (EFM)

```
1: Initialize *buffer* to store the first $k$ w's and the $WARank(w, Q, \alpha)$.
2: $Lrank \Leftarrow 0$
3: $minRank \Leftarrow$ the last rank in *buffer*.
4: Bounding phase: get $Q^{low}$ and $Q^{up}$
5: $heap_w.enqueue(R\text{-}tree_w.root)$
6: **while** $heap_w$ is not empty **do**
7:    $ew \Leftarrow heap_w.dequeue$
8:    $Cand \Leftarrow \emptyset$
9:    $heap_p.enqueue(R\text{-}tree_p.root)$
10:   **while** $heap_p$ is not empty **do**
11:      $ep \Leftarrow heap_p.dequeue$
12:      **if** $ep$ is located below the lower hyperplane **then**
13:         // Lemma 1.
14:         $Lrank \Leftarrow Lrank + ep.size \cdot |Q| \cdot \alpha_{min}$
15:         **if** $Lrank \geq minRank$ **then**
16:            Continue
17:      **if** $ep$ in sandwiched space **then**
18:         $Cand \Leftarrow Cand \cup ep$
19:      **if** $ep$ covers the upper or lower hyperplane **then**
20:         $heap_p.enqueue(ep.children)$
21:   **if** $Lrank \leq minRank$ **then**
22:      // Corollary 3.
23:      Compute further bounds and update $Lrank$ and $Cand$.
24:      **if** $Lrank \leq minRank$ **then**
25:         **if** $ew$ is a single $w$ **then**
26:            // Lemma 2.
27:            Compute exact weighted rank $WARank$.
28:            **if** $WARank \leq minRank$ **then**
29:               Update *buffer* and $minRank$.
30:         **else**
31:            $heap_w.enqueue(ew.children)$
32: **return** *buffer*
```

5 OPTIMAL BOUNDING METHOD (OBM)

As Figure 5 shows, the key point of efficiency in the bound-and-filter framework is the bounding phase, because the tightness of the bounds of Q determines the effectiveness of filtering both P and W. The score of the bounds $f(ew, Q^{up})$ and $f(ew, Q^{low})$ determine the amount of data in P that can be filtered. Moreover, as mentioned in Section 4, the higher the lower hyperplane $H(ew, Q.low)$ is located, the higher the value of the lower rank bound will be, and the more data from W will be filtered. According to Figure 5(c), if Q could be bounded more tightly, ew_2 might be filtered directly, without further computation. In conclusion, tightening the bounds of Q is significant to the performance.

In this section, we propose the optimal bounds of Q. We utilize the theory of linear programming to prove the optimization. We propose an OBM for *WARR* query based on these optimal bounds.

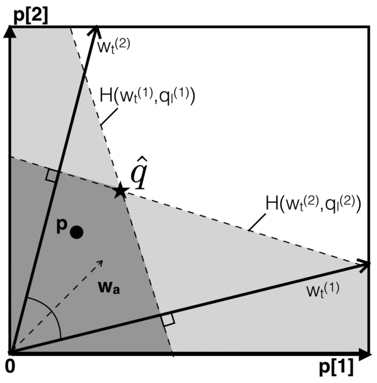


Fig. 6. The half-spaces of $H(w_t^{(i)}, q_l^{(i)})$, $i = 1, 2, \ldots, d$. The intersection point is the optimal lower bound of Q for an arbitrary $w \in W$.

5.1 The Optimal Bounds for Q

An arbitrary preference $w_a \in W$ can be represented by the linear combination of $w_t^{(i)}$ with coefficient γ_i as follows:

$$w_a = \sum_{i=1}^{d} \gamma_i w_t^{(i)},\ where\ \gamma_i \geq 0\ and\ \sum_{i=1}^{d} \gamma_i = 1, \tag{6}$$

where $\sum_{i=1}^{d} \gamma_i = 1$ is guaranteed by $\sum_{i=1}^{d} w_a[i] = 1$, as shown below.

$$\begin{aligned} 1 = \sum_{j=1}^{d} w_a[j] &= \sum_{j=1}^{d} \sum_{i=1}^{d} \gamma_i w_t^{(i)}[j] \\ &= \sum_{i=1}^{d} \gamma_i \sum_{j=1}^{d} w_t^{(i)}[j] \\ &= \sum_{i=1}^{d} \gamma_i \times 1 \end{aligned} \tag{7}$$

Before formally stating and proving the d-dimensional case, we explain the lower bound optimum with 2-dimensional data. (The process for the upper bound can be illustrated in exactly the same way.) In Figure 6, there are top preferences $W_t = \{w_t^{(1)}, w_t^{(2)}\}$ of all dimensions, and two corresponding perpendicular lines (hyperplanes) of minimum values for $f(w_t^{(i)}, q_l^{(i)})$, $i = 1, 2, \ldots, d$ are determined.

The construction of the hyperplanes indicates that p has a lower score than $q_l^{(i)}$ on $w_t^{(i)}$ when p is located in the half-space below $H(w_t^{(i)}, q_l^{(i)})$. In Figure 6, where p is located in the overlap area (dark gray) of the two half-spaces, $p \cdot w_t^i \leq q_l^{(i)} \cdot w_t^{(i)}$ for $i = 1, 2, \ldots, d$. According to Equation (6), the score of p based on an arbitrary w_a, $f(w_a, p) = p \cdot w_a$, can be presented as follows:

$$p \cdot w_a = p \cdot \left(\gamma_1 w_t^{(1)} + (1 - \gamma_1) w_t^{(2)}\right) \leq \gamma_1 q_l^{(1)} \cdot w_t^{(1)} + (1 - \gamma_1) q_l^{(2)} \cdot w_t^{(2)} \tag{8}$$

Let the intersection point of $H(w_t^{(i)}, q_l^{(i)})$ be $\hat{q}$. $\hat{q}$ locates on both $H(w_t^{(i)}, q_l^{(i)}), for\ i = 1, 2, \ldots, d$, meaning that

$$\begin{cases} \hat{q} \cdot w_t^{(1)} & = q_l^{(1)} \cdot w_t^{(1)} \qquad (a) \\ \hat{q} \cdot w_t^{(2)} & = q_l^{(2)} \cdot w_t^{(2)} \qquad (b) \end{cases}$$

Replacing the *r.h.s* of Equation (8) with the *l.h.s* of $\gamma_1 \times (a) + (1 - \gamma_1) \times (b)$ gives

$$p \cdot w_a \leq \gamma_1 \hat{q} \cdot w_t^{(1)} + (1 - \gamma_1)\hat{q} \cdot w_t^{(2)} = \hat{q} \cdot w_a \tag{9}$$

By the theory of linear programming, it is easy to know that $p \cdot w$ takes the maximum value at $\hat{q}$. In other words, the score of point $\hat{q}$ is optimal.

Based on this discussion, we can lay out the formal conclusion that shows that the optimal bounds of Q are the intersection points.[3]

THEOREM 1 (THE OPTIMAL BOUNDS OF Q). *Given Q_u and Q_l from Q and W_t from W, let Q_{opt}^{low} be the intersection point(s) of all hyperplanes $\{H(w_t^{(i)}, q_l^{(i)}) | i = 1, 2, \ldots, d\}$, and Q_{opt}^{up} be the intersection point(s) of all hyperplanes $\{H(w_t^{(i)}, q_u^{(i)}) | i = 1, 2, \ldots, d\}$, respectively. Then, Q_{opt}^{low} and Q_{opt}^{up} are the optimal lower and upper bounds of Q, respectively.*

5.2 Proof of the Optimal Bounds (Theorem 1)

As in the discussion for the 2-dimensional case, we only give the proof for the lower bound Q_{opt}^{low}, since $Q.up$ can be proved in the same way.

PROOF. (Q_{opt}^{low} of Theorem 1):

Assume that the point $\hat{q}$ can bound Q with an arbitrary $w_a \in W$. Because the larger value of $f(w_a, \hat{q})$ filters more data, we want to find the $\hat{q}$ that maximizes $w_a \cdot \hat{q}$. The problem can be converted to a linear programming problem with the standard form as follows:

$$\boxed{\begin{array}{ll} \texttt{Maximize:} & \hat{q} \cdot w_a \\ & \\ \texttt{Subject to:} & \hat{q} \cdot w_t^{(i)} \leq f(w_t^{(i)}, q_l^{(i)}) \\ & \hat{q}[i] \geq 0, \quad i = 1, 2, \ldots, d \end{array}}$$

By the theory of linear programming, the optimal lower bound is the intersection point(s) of all hyperplanes $\{H(w_t^{(i)}, q_l^{(i)}) | i = 1, 2, \ldots, d\}$. Generalizing the Equations (8.a) and (8.b) to d-dimensional case, the intersection point(s) $\hat{q}$ found by solving the following simultaneous Equations (10) and (11) is the optimum solution of the above problem and hence the lower bound. In other words, like Equation (9), the score of a p under all the hyperplanes based on an arbitrary w_a satisfies $p \cdot w_a \leq \hat{q} \cdot w_a$.

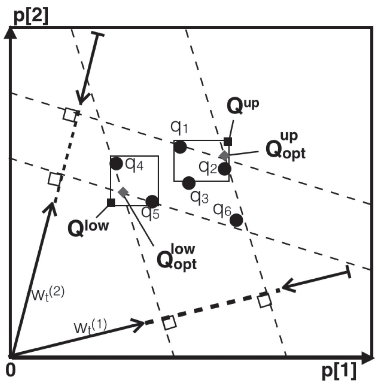
$$A \cdot \hat{q} = c, \tag{10}$$

where

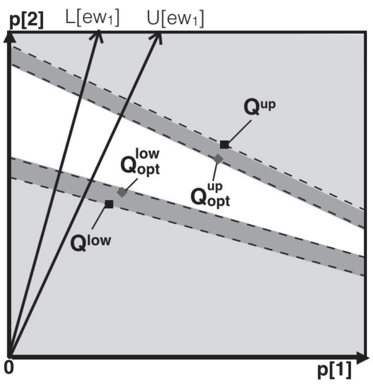
$$A = \begin{bmatrix} w_t^{(1)}[1] & \cdots & w_t^{(1)}[d] \\ \vdots & \ddots & \vdots \\ w_t^{(d)}[1] & \cdots & w_t^{(d)}[d] \end{bmatrix} and\ c = \begin{bmatrix} f(w_t^{(1)}, q_l^1) \\ \vdots \\ f(w_t^{(d)}, q_l^d) \end{bmatrix} \tag{11}$$

□

[3] If there is more than one point in the solution, any one of them can be the bound since they all have the same score.



(a) The bounding phase.



(b) The filtering phase (gray area).

Fig. 7. Bound-and-filter in OBM. (a) Finding the optimal bounds Q_{opt}^{low} and Q_{opt}^{up}. (b) Comparing the filtering space of the previous $Q^{up}(Q^{low})$ and optimal bounds.

Table 2. The Complexities of the Methods

Algorithm	Index	CPU cost	I/O cost
NAIVE	None	$O(\|P\| \cdot \|W\| \cdot \|Q\|)$	$\|P\| + \|W\|$
SFM	RtreeP	$O(\|W\| \cdot \log \|P\|)$	$\log \|P\| + \|W\|$
EFM	RtreeP, RtreeW	$O(\log \|W\| \cdot \log \|P\|)$	$\log \|W\| + \log \|P\|$
OBM	RtreeP, RtreeW	$O(\log \|W\| \cdot \log \|P\|)$	$\log \|W\| + \log \|P\|$

5.3 OBM Algorithm

Since computing Q_{opt}^{low} and Q_{opt}^{up} is equivalent to solving the linear equations of Equation (11) and finding $\hat{q}$, Gaussian elimination [4] is an easy and low-cost method for doing so. The total complexity of Gaussian elimination is approximate to $O(d^3)$, where d is the dimensionality of the data and indicates the number of linear equations. Therefore, the complexity of the bounding phase in OBM is $O(d \cdot |Q| + d^3)$. Notice that the complexity of finding Q^{up} and Q^{low} is $O(d \cdot |Q|)$ according to Ref. [6] and the cube of dimensionality is still a very small value. The cost of bounding Q is negligible to the whole bound-and-filter algorithm, since both $|Q|$ and d are far smaller than the cardinality of W and P.

We take advantage of this optimal bound and propose the OBM method. The bounding phase of OBM is described in Algorithm 3. The filtering phase of OBM is to replace the Q^{low} and Q^{up} in EFM with the best bounds Q_{opt}^{low} and Q_{opt}^{up}. Furthermore, Lemmas 1 and 2 and Corollary 3 also hold to the optimal bounds. Figure 7 shows the optimal bounds and filtering space of OBM. In Figure 7(a), it is obvious that Q_{opt}^{low} and Q_{opt}^{up} bound Q more tightly than the previous Q^{low} and Q^{up}. Figure 7(b) also shows that more data from P can be filtered than in EFM.

Table 2 summarizes the space and time complexities for NAIVE and the proposed SFM, EFM, and OBM. NAIVE has the highest time complexity $O(|P| \cdot |W|)$ but no need for extra index storage. SFM uses R-tree to index P so it costs $O(|W| \cdot \log |P|)$. EFM and OBM are based on a bound-and-filter framework with two R-trees and have the complexities of $O(\log |W| \cdot \log |P|)$. The difference is that the optimal bounding strategy makes OBM better than EFM.

[4]https://en.wikipedia.org/wiki/Gaussian_elimination.

ALGORITHM 3: Optimal Bounding

```
1: $W_t$ has been found offline
2: $A_l$ and $A_u$ are matrixes for storing hyperplane equations.
3: **for** each $w_t^{(i)} \in W_t$ **do**
4:     $q_l^i \leftarrow argmax(f(w_t^{(i)}, q)), q \in Q$
5:     $Q_l \leftarrow Q_l \cup \{q_l^i\}$
6:     $A_l \leftarrow A_l \cup \{w_t^{(i)} \cup \{f(w_t^{(i)}, q_l^i)\}\}$
7:     $q_u^i \leftarrow argmin(f(w_t^{(i)}, q)), q \in Q$
8:     $Q_u \leftarrow Q_u \cup \{q_u^i\}$
9:     $A_u \leftarrow A_u \cup \{w_t^{(i)} \cup \{f(w_t^{(i)}, q_u^i)\}\}$
10: $Q_{opt}^{low} \leftarrow GuassianElimination(A_l)$
11: $Q_{opt}^{up} \leftarrow GuassianElimination(A_u)$
12: **return** $\{Q_{opt}^{low}, Q_{opt}^{up}\}$
```

6 EXPERIMENT

In this section, we present an extensive experimental evaluation of the NAIVE and proposed SFM, EFM, and OBM algorithms for *WARR* query. All algorithms were implemented in C++ and the experiments were run on a Mac with a 2.6GHz Intel Core i7 CPU, 16GB RAM, and 256G flash storage.

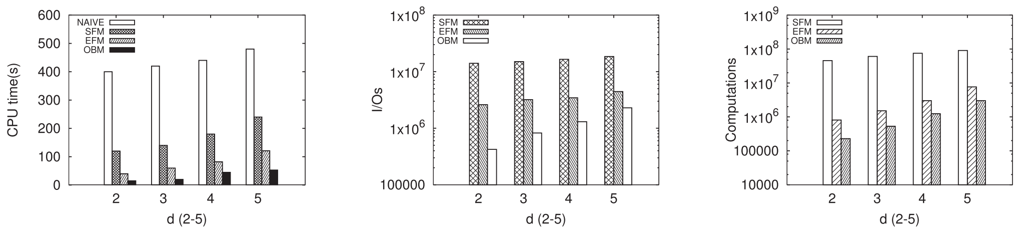
6.1 Datasets and Metrics

Product dataset *P*: We used both synthetic and real-world data for *P*:

— *Synthetic datasets*: The synthetic datasets are uniform (UN), clustered (CL), and anti-correlated (AC). The attribute value range of each dimension is [0,1). For the UN dataset, all attribute values are generated independently and following a uniform distribution. The AC dataset is generated by selecting a plane perpendicular to the diagonal of the data space using a normal distribution; we generate attribute values in this plane and follow a uniform distribution. For the CL dataset, first, the cluster centroids are selected randomly and follow a uniform distribution. Then, each attribute is generated with the normal distribution. We use the centroid values as the mean and 0.1 as variance. All of the above distributions were used in related work of other reverse rank queries [6, 24, 25, 32].

— *Real-world datasets*: We also use two real datasets, NBA[5] and Amazon.[6] The NBA dataset contains 20,960 tuples of box scores of basketball players in NBA seasons from 1949 to 2009. We extracted 5 tuples to evaluate a player using points, rebounds, assists, blocks, and steals statistics. The NBA dataset was also used in the *ARR* query in Ref. [6]. Another real-world dataset is the metadata of products from Amazon.com, a well-known online retailer. This metadata contains 1,689,188 user reviews on 208,321 tuples of products in the categories of Movies and TV, in which product bundling is common. Each user provides at least five reviews, and each product is reviewed by at least five users. All the values were normalized based on the definition. We extracted price and sales rank from the metadata as

[5]NBA: http://www.databasebasketball.com.
[6]Amazon: http://jmcauley.ucsd.edu/data/amazon/.



(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 8. Comparison results of varying d (2-5) on UN data P, W:UN, $|P| = 20$K, $|W| = 200$K, all with $|Q| = 5$, $k = 10$.

2-dimensional vectors that represent a product. The Amazon data are also used in other research, such as Refs [15] and [16].

User preference dataset W: For dataset W, we also have the synthetic datasets, UN and CL, which were generated in the same manner as the P datasets. For the real data of Amazon, for a specific user $w \in W$, we computed the average value on "Price" and "salesRank" of the products, which the user bought, then assemble these values as a 2-dimensional vector that represents this user's preference.

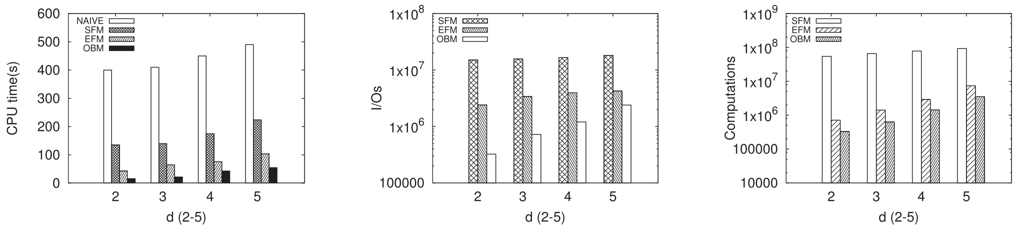
Query products Q: For the query products Q, we have two strategies to generate the queries. The first is to select a clustered subset from the product data P. In particular, we select a product in dataset P randomly, then find its m nearest neighbor in P, where m is the pre-defined cardinality of Q. We use this strategy as the default Q since it is a common situation that the products are always similar in a product bundling in the real-life applications (i.e., bundled books, clothes, and games). On the other hand, we also test the performance on the Q, which is randomly selected from P simply (uniform); this test is for the situation that products in a bundle are not in a cluster (i.e., a mobile phone has a different price compared to its charging cable and case).

Weights α: The weights α corresponding to Q are generated randomly.

Efficiency metrics: We use three metrics to observe the efficiency of all algorithms. (a) The query execution time (CPU time) required by each algorithm; (b) the I/O cost. I/O is estimated by checking accessed nodes in $R\text{-}tree_p$ and $R\text{-}tree_w$. We also observe (c) the number of pairwise computations between P and W, which is a statistic that clearly shows the performance of each algorithm. We present average values over 1,000 queries in all cases.

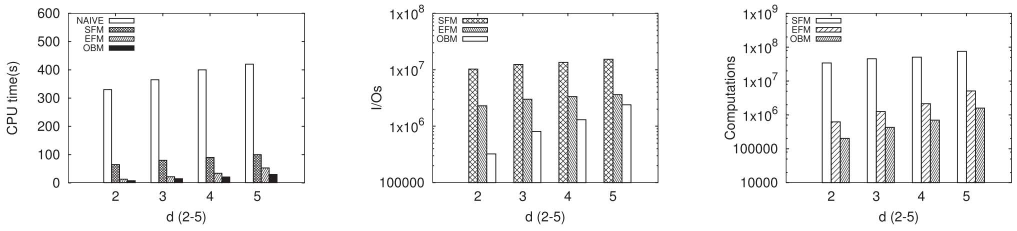
6.2 Experimental Results

Synthetic data: Figure 8 presents the comparative performance of all algorithms on UN data of both P and W for varying dimensionality d. The cardinality of $|P| = 20$K, $|W| = 200$K, k = 10, and $|Q| = 5$. According to the execution time results shown in Figure 8(a), our three proposed methods are significantly faster than the NAIVE algorithm. The EFM and OBM methods, which use the bound-and-filter framework with two R-trees, are superior to SFM because they avoid checking each $w \in W$. OBM is the most efficient, 2–3 times faster than EFM with the help of its optimal bounding strategy. We also found that the performance of SFM, EFM, and OBM decrease as dimensionality increases; in higher dimensional space, query Q intersects more MBRs of the R-tree and the tree-based algorithms traverse deeper layers of the tree-structure for filtering data. Figure 8(b) shows the I/O cost of the proposed algorithms. EFM and OBM are better than SFM, since they only access a part of W with the $R\text{-}tree_w$ while SFM needs to check every w. OBM has a lower I/O cost than EFM due to the optimal bounds on Q in OBM, which allows it to filter more



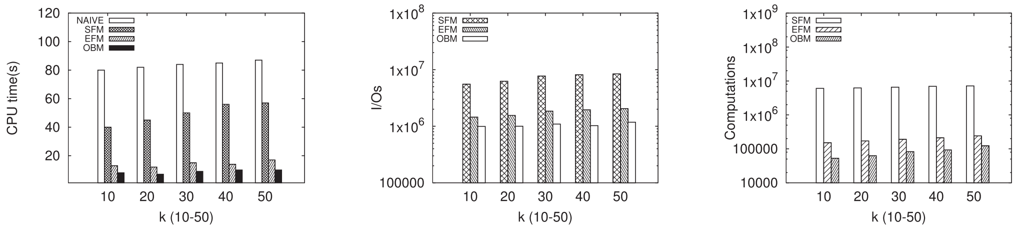
(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 9. Comparison results of varying d (2-5) on AC data P, W: UN, $|P| = 20K$, $|W| = 200K$, all with $|Q| = 5$, $k = 10$.



(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 10. Comparison results of varying d (2-5) on CL data P and W, $|P| = 20K$, $|W| = 200K$, all with $|Q| = 5$, $k = 10$.



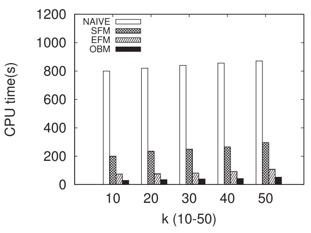
(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 11. Comparison results of varying k (10-50) on NBA data, $|P| = 20{,}960$, $|W|$: UN, $|W| = 100K$, all with $|Q| = 5$, $d = 5$.

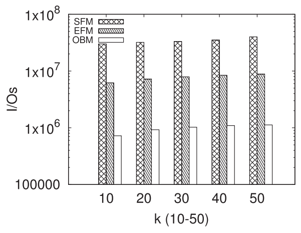
data than EFM does, as was proved in Theorem 1. The observation of pairwise computations is shown in Figure 8(c), which is an insight view of all algorithms. OBM makes the fewest pairwise computations because it can filter the most data among all the algorithms; this also proves that OBM requires the least computation time when processing queries.

The comparison results of AC and CL data in the same setting as the UN experiment are shown in Figures 9 and 10, respectively. Similarly to the results of the UN data, OBM is the most efficient method; not only is it the fastest algorithm, but it also has the lowest I/O cost and number of pairwise computations. We found that the performance of EFM and OBM on CL data are better than on UN data, since the bound-and-filter framework can filter more MBRs in $R\text{-}tree_p$ and $R\text{-}tree_w$ when P and W are clustered.

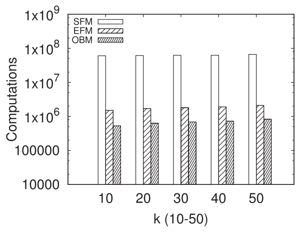
Real-world NBA data. Figure 11 shows the CPU time and I/O cost for all algorithms on NBA data, with varying k. Clearly, OBM is more efficient than others and has lower I/O cost and fewer



(a) CPU time.

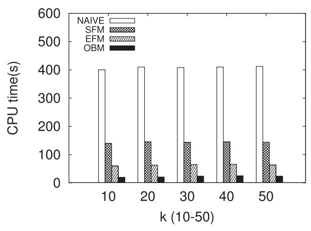


(b) I/Os cost.

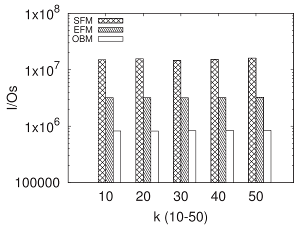


(c) Pairwise computations.

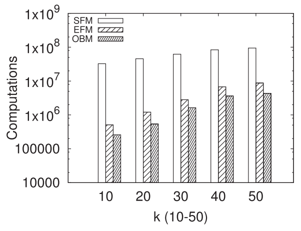
Fig. 12. Comparison results of varying k (10-50) on AMAZON data, $|P|$ = 208,321, $|W|$ = 1,689,188, all with $|Q| = 5$, $d = 2$.



(a) CPU time.



(b) I/Os cost.



(c) Pairwise computations.

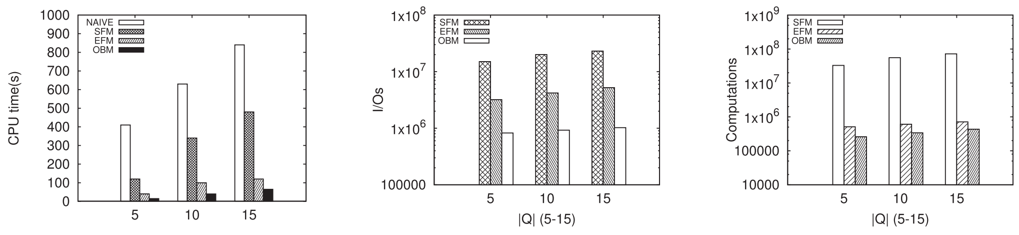
Fig. 13. Comparison results of varying k (10-50) on UN data P and W, $|P|$ = 20K, $|W|$ = 200K, all with $|Q|$ = 5, $d = 3$.

pairwise computations. The NBA data were also used in *ARR* query in Ref. [6] to answer the question "Who loves a given basketball team more than other people do?" Every player on a basketball team has his responsibility; e.g., the center and power forward defend and take rebounds, while the point guard and score guard need to pass and score. In addition to verifying *WARR* query's efficiency, we also tested its practical applicability. We set a query team that was good at defense, with (a) equal weights (*ARR* query) and (b) large weights on the center and power forward (*WARR* query). The results from the *WARR* query all have greater preferences for the rebound and block attributes; this means that *WARR* query returns a correct set of people, those who prefer defensive teams. For the above observation, we conclude that *WARR* is more reasonable than the *ARR* query.

Real-world Amazon data. The Amazon dataset contains e-commerce data. Comparison results of CPU time, I/O cost, and pairwise computing times are shown in Figure 12 with varying k on UN data from W. We randomly selected five movies or TV programs from the Amazon data as a product bundle query set. OBM maintains its efficiency in execution time and I/O cost, as can be seen in Figure 12(a) and (b). This is a very strong result that demonstrates the efficiency of OBM in practical marketing applications.

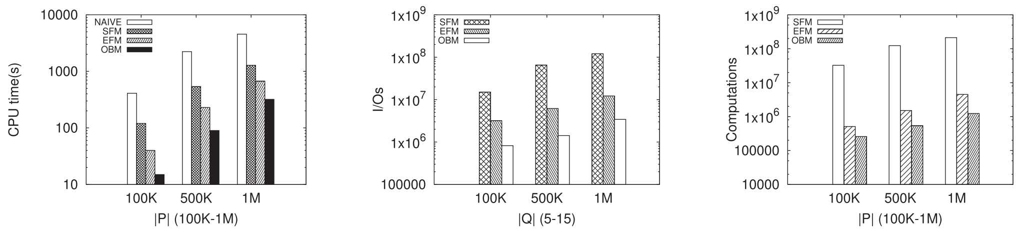
Effect of varying k. Performance results when varying k on 3-dimensional UN data with $|Q|$ = 5, $|P|$ = 20K, and $|W|$ = 200K, are shown in Figure 13. All algorithms are insensitive to k. First, k is far smaller than $|W|$ and $|P|$. Second, k is the number of results of w for *WARR* query, so the value of k does not affect performance very much for any algorithms, even though the NAIVE and SFM algorithms check all $w \in W$. EFM and OBM keep a k-element ascending *buffer* while processing, so they are only concerned with the last element with *minRank* rather than all k candidates in the *buffer*.

Effect on varying $|Q|$. For the varying $|Q|$ in Figure 14, because the number of products in a product bundle is not generally large, we test $|Q|$ from 5 to 15. EFM and OBM are insensitive to



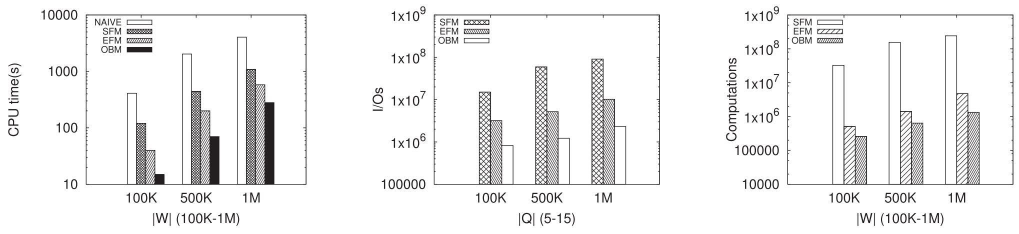
(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 14. Comparison results of varying $|Q|$ (5-25) on UN data P and W, $|P|$ = 20K, $|W|$ = 200K, all with k = 10, d = 3.



(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 15. Scalability of varying P (100K-1M) on UN data P and W, $|P|$ =100K, all with k = 10, d = 3, $|Q|$ = 5.



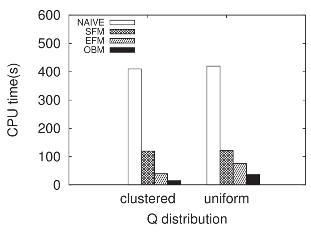
(a) CPU time. (b) I/Os cost. (c) Pairwise computations.

Fig. 16. Scalability of varying W (100K-1M) on UN data P and W, $|W|$ = 100K, all with k = 10, d = 3, $|Q|$ = 5.

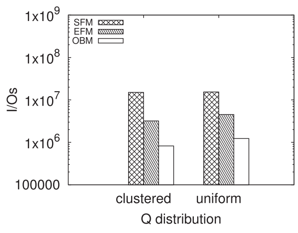
$|Q|$ based on these results because they bound Q in advance. However, the efficiency of the NAIVE and *SFM* algorithms decrease as $|Q|$ increases, as they check every q.

Scalability with varying $|P|$. Figure 15 shows the performance of all algorithms when increasing the cardinality of dataset P. We show the results of $|P| = 100K, 500K, 1M$, with $|W| = 100K$. The scalability of all algorithms are with respect to $|P|$, and OBM maintains the advantage over other algorithms. Because SFM, EFM, and OBM all use R-trees to index P, the execution time and data accessed grow faster than linearly with P. This is clearly shown in Figure 15(a), (b), and (c).

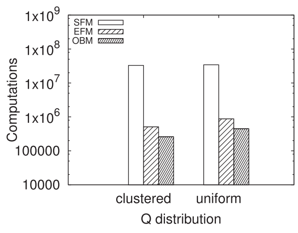
Scalability with varying $|W|$. We also test the scalability of all algorithms for varying $|W|$. We show the results of CPU time and I/O cost with the setting of $|W| = 100K, 500K, 1M$, with $|P| = 100K$, $d = 3$, $|Q| = 5$, and k = 5. These results differ with those of varying $|P|$ in Figure 16; in this case, only EFM and OBM maintain their growth with increasing $|W|$. OBM is still the most efficient algorithm. Figure 16(c) gives the insight view of processing with the number of pairwise computations.



(a) CPU time.



(b) I/Os cost.



(c) Pairwise computations.

Fig. 17. Comparison results of different distribution on Q, $|P| = 20$K, $|W| = 200$K, all with $k = 10$, $d = 3$, $|Q| = 5$.

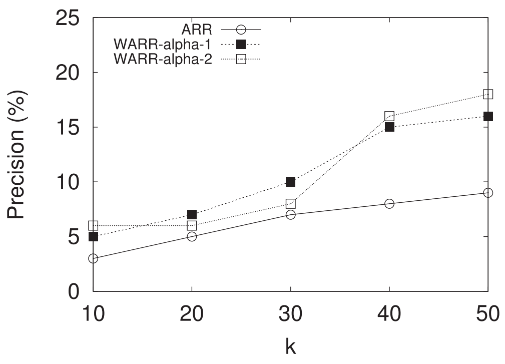


Fig. 18. Precision of *ARR* and *WARR* on Amazon data.

Effect on the distribution of Q. Figure 17 shows the comparison results on clustered Q and uniform Q. We can see that the performances in NAIVE and SFM are not changed with the clustered Q since they process queries in Q independently. On the other hand, the clustered Q has a better performance in the bound-and-filter based methods EFM and OBM. This is because the uniform Q may select a large distribution of products, then loose the bound of Q and compute more data than a tighter bound in a cluster.

6.3 Effectiveness

We test the effectiveness of *WARR* with AMAZON metadata and reviews data in Section 6.1, in comparison with previous *ARR* [6]. The details are as follows:

—Product (P): Price, sales rank, and rating are three attributes of a product. Price and sales rank are from the metadata, which is also used in our experiments of performance comparison. The rating of a specific product is computed as the average value of the reviews on this product.
—User preference (W): The user preference is also a three-dimensional vector, which has values on (price, sales rank, rating), corresponding to the attributes of a product. For a user, the value of price and sales rank are computed as the average value of the products she has bought, and the value of the rating is the average value of her reviews.
—Query bundled products (Q): We first select a product p from P randomly, then find the "bought together" product of the selected p, and use these two products as a product bundle.

We issued two types of queries of *WARR* and *ARR* with 100 randomly selected Q, and recorded 50 results for each Q (i.e., $k = 10 \sim 50$). The precision is defined as the proportion of the users who have bought all products of Q. We set $\alpha_0 = (0.5, 0.5)$ as the *ARR* query, which treats everything equally. On the other hand, we set $\alpha_1 = (0.75, 0.25)$ and $\alpha_2 = (0.25, 0.75)$, which means that either would be the appreciative one. Figure 18 reports the precision of *ARR* and *WARR* with α_1 and α_2. According to these results, *WARR*'s precision is better than *ARR* in all situations. Of course, the precision depends on α. Nevertheless, people always evaluate products unequally when they consider buying a bundled products, and *WARR* enables it to adjust the weights reflexing such request.

7 CONCLUSION

In this study, we proposed a general, weighted aggregate reverse rank (*WARR*) query. To *WARR*, aggregate reverse rank (*ARR*) query is only a simple, special case in which all query points are treated with equal importance. *WARR* query can be critical in various applications, such as finding potential customers and analyzing marketing via different views for a set of products. We proposed three solutions for solving *WARR* query efficiently. SFM is a straightforward way to use tree-based methods for reducing the computation of product data. The EFM adapts the previous bound-and-filter framework and is made able to solve *WARR* queries by filtering the pairwise computation from both product and preferences data. To optimize the bound, we designed a new bounding strategy, then developed and implemented an OBM. We theoretically proved the optimum of the bounds in OBM and compared the performance of the above three methods with both synthetic and real data. The results show that OBM is the most efficient of these algorithms.

In future work, we first plan to make use of *WARR* query to implement an application tool for e-commerce marketing analysis. We also plan to investigate how to set appropriate weights in *WARR* query.

REFERENCES

[1] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2007. Best position algorithms for top-k queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 495–506.

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

[3] Surajit Chaudhuri and Luis Gravano. 1999. Evaluating top-k selection queries. In *Proceedings of 25th International Conference on Very Large Data Bases(VLDB'99)*. 397–410.

[4] Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2011. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *Proceedings of the 27th ICDE*. 577–588. DOI: http://dx.doi.org/10.1109/ICDE.2011.5767904

[5] Evangelos Dellis and Bernhard Seeger. 2007. Efficient computation of reverse skyline queries. In *Proceedings of the 33rd International Conference on VLDB*. 291–302.

[6] Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, and Hiroyuki Kitagawa. 2016. Aggregate reverse rank queries. In *Proceedings of the 27th International Conference on Database and Expert Systems Applications, Part II*. 87–101. DOI: http://dx.doi.org/10.1007/978-3-319-44406-2_8

[7] Yuyang Dong, Hanxiong Chen, Jeffrey Xu Yu, Kazutaka Furuse, and Hiroyuki Kitagawa. 2017. Grid-index algorithm for reverse rank queries. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT'17)*. 306–317. DOI: http://dx.doi.org/10.5441/002/edbt.2017.28

[8] Ronald Fagin. 1999. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.* 58, 1 (1999), 83–99. DOI: http://dx.doi.org/10.1006/jcss.1998.1600

[9] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003), 614–656. DOI: http://dx.doi.org/10.1016/S0022-0000(03)00026-6

[10] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A System for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD*. 259–270. DOI: http://dx.doi.org/10.1145/375663.375690

[11] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008), 11:1–11:58. DOI: http://dx.doi.org/10.1145/1391729.1391730

[12] Flip Korn and S. Muthukrishnan. 2000. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD*. 201–212. DOI: http://dx.doi.org/10.1145/342009.335415

[13] Xiang Lian and Lei Chen. 2008. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proceedings of the ACM SIGMOD*. 213–226. DOI: http://dx.doi.org/10.1145/1376616.1376641

[14] Amélie Marian, Nicolas Bruno, and Luis Gravano. 2004. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.* 29, 2 (2004), 319–362. DOI: http://dx.doi.org/10.1145/1005566.1005569

[15] Julian J. McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794. DOI: http://dx.doi.org/10.1145/2783258.2783381

[16] Julian J. McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 43–52. DOI: http://dx.doi.org/10.1145/2766462.2767755

[17] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. 2004. Group nearest neighbor queries. In *Proceedings of the 20th International Conference on Data Engineering(ICDE'04)*. 301–312. DOI: http://dx.doi.org/10.1109/ICDE.2004.1320006

[18] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. 2005. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.* 30, 2 (2005), 529–576. DOI: http://dx.doi.org/10.1145/1071610.1071616

[19] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. 2000. Reverse nearest neighbor queries for dynamic databases. In *Proceedings of the ACM SIGMOD Workshop*. 44–53.

[20] Yufei Tao, Vagelis Hristidis, Dimitris Papadias, and Yannis Papakonstantinou. 2007. Branch-and-bound processing of ranked queries. *Inf. Syst.* 32, 3 (2007), 424–445. DOI: http://dx.doi.org/10.1016/j.is.2005.12.001

[21] Yufei Tao, Dimitris Papadias, and Xiang Lian. 2004. Reverse kNN search in arbitrary dimensionality. In *Proceedings of the 13th International Conference on VLDB*. 744–755.

[22] Yufei Tao, Dimitris Papadias, Xiang Lian, and Xiaokui Xiao. 2007. Multidimensional reverse kNN search. *VLDB J.* 16, 3 (2007), 293–316. DOI: http://dx.doi.org/10.1007/s00778-005-0168-2

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

[24] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. 2010. Reverse top-k queries. In *Proceedings of the 26th International Conference on Data Engineering (ICDE'10)*. 365–376. DOI: http://dx.doi.org/10.1109/ICDE.2010.5447890

[25] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. 2011. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.* (2011), 1215–1229.

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

[27] Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. 2011. Monitoring reverse top-k queries over mobile devices. In *MobiDE*. 17–24.

[28] Shenlu Wang, Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, and Dongxi Liu. 2016. Efficiently computing reverse k furthest neighbors. In *Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE'16)*. 1110–1121. DOI: http://dx.doi.org/10.1109/ICDE.2016.7498317

[29] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. 2015. Reverse k nearest neighbors query processing: Experiments and analysis. *PVLDB* 8, 5 (2015), 605–616.

[30] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Ying Zhang. 2014. SLICE: Reviving regions-based pruning for reverse k nearest neighbors queries. In *Proceedings of the IEEE 30th International Conference on Data Engineering, ICDE*. 760–771. DOI: http://dx.doi.org/10.1109/ICDE.2014.6816698

[31] Bin Yao, Feifei Li, and Piyush Kumar. 2009. Reverse furthest neighbors in spatial databases. In *Proceedings of the 25th ICDE*. 664–675. DOI: http://dx.doi.org/10.1109/ICDE.2009.62

[32] Zhao Zhang, Cheqing Jin, and Qiangqiang Kang. 2014. Reverse k-ranks query. *PVLDB* 7, 10 (2014), 785–796.

Received May 2017; revised January 2018; accepted May 2018