PAPER *Special Section on Data Engineering and Information Management*

# Efficient Methods for Aggregate Reverse Rank Queries

**Yuyang DONG**[†a)], **Hanxiong CHEN**[†b)], **Kazutaka FURUSE**[†c)], ***Nonmembers***, ***and*** **Hiroyuki KITAGAWA**[††d)], ***Member***

**SUMMARY** Given two data sets of user preferences and product attributes in addition to a set of query products, the aggregate reverse rank (ARR) query returns top-k users who regard the given query products as the highest aggregate rank than other users. ARR queries are designed to focus on product bundling in marketing. Manufacturers are mostly willing to bundle several products together for the purpose of maximizing benefits or inventory liquidation. This naturally leads to an increase in data on users and products. Thus, the problem of efficiently processing ARR queries become a big issue. In this paper, we reveal two limitations of the state-of-the-art solution to ARR query; that is, (a) It has poor efficiency when the distribution of the query set is dispersive. (b) It has to process a large portion user data. To address these limitations, we develop a cluster-and-process method and a sophisticated indexing strategy. From the theoretical analysis of the results and experimental comparisons, we conclude that our proposals have superior performance.

***key words:*** *similarity search, aggregate reverse rank queries, clustering method, cone*$^{+}$ *tree*

## 1. Introduction

In the user-product mode, there are two different datasets: user preferences and products. A top-$k$ query retrieves the top-$k$ products for a given user preference in this model. However, manufacturers also want to know the potential customers for their products. Therefore, reverse $k$-rank query [1] is proposed to obtain the top-$k$ user preferences for a given product. Because most manufacturers offer several products as part of product bundling, the aggregate reverse rank query (*ARR*) [2] responds to this requirement by retrieving the top-$k$ user preferences for a set of products. Not limited to shopping, the concept of ARR can be extended to a wider range of applications such as team (multiple members) reviewing and area (multiple businesses) reviewing.

An example of ARR query is shown in Fig. 1. There are five different books ($p_1$–$p_5$) scored on "price" and "ratings" in Table (b). Three preferences of users Tom, Jerry, and Spike are listed in Table (a), which are the weights for each attribute in the book. The score of a book under a user preference is the defined value of the inner product of the book attributes vector and user preference vector [1]–[3][*]. Now, assume that the book shop selects two bundles from books, say $\{p_1, p_2\}$ and $\{p_4, p_5\}$. The result of ARR query when $k = 1$ for them are shown in Table (c). The ARR query evaluates the aggregate rank (ARank) with the sum of each book's rank, e.g., $\{p_1, p_2\}$ ranks as $3 + 2 = 5$ based on Tom's preference. The ARR query returns Tom as the result because Tom thinks the books $\{p_1, p_2\}$ has the highest rank than the rest.

**(a) User preferences data and the ranks**

| | w[price] | w[rating] | Ranking |
|---|---|---|---|
| Tom | 0.8 | 0.2 | $p_3,p_2,p_1,p_4,p_5$ |
| Jerry | 0.3 | 0.7 | $p_2,p_5,p_3,p_4,p_1$ |
| Spike | 0.1 | 0.9 | $p_5,p_2,p_4,p_3,p_1$ |

**(b) Books data and their ranks on users.**

| | p[price] | p[rating] | Rank on Tom | Rank on Jerry | Rank on Spike |
|---|---|---|---|---|---|
| $p_1$ | 6 | 7 | 3rd | 5th | 5th |
| $p_2$ | 2 | 3 | 2nd | 1st | 2nd |
| $p_3$ | 1 | 6 | 1st | 3rd | 4th |
| $p_4$ | 7 | 5 | 4th | 4th | 3rd |
| $p_5$ | 8 | 2 | 5th | 2nd | 1st |

**(c) Bundled books and their ARR-1Rank**

| | ARank on Tom | ARank on Jerry | ARank on Spike | AR-1Rank |
|---|---|---|---|---|
| $p_1,p_2$ | 5 (3+2) | 6 (5+1) | 7 (5+2) | Tom |
| $p_4,p_5$ | 9 (4+5) | 6 (4+2) | 4 (3+1) | Spike |

**Fig. 1** The example

### 1.1 Definitions

The assumptions made on the product database, preferences database and the score function between them are the same as those made in the related research [1]–[4]. The querying problems are based on a User-Product model, which has two kinds of database: product data set $P$ and user's preference data set $W$. Each product $p \in P$ is a $d$-dimensional vector that contains $d$ non-negative scoring attributes. The product $p$ can be represented as a point $p = (p[1], p[2], \ldots, p[d])$, where $p[i]$ is the $i$th attribute value. The preference $w \in W$ is also a $d$-dimensional vector and $w[i]$ is a non-negative weight that affects the value of $p[i]$, where

Manuscript received June 20, 2017.
Manuscript revised November 6, 2017.
Manuscript publicized January 18, 2018.

[†]The authors are with the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
[††]The author is with the Center for Computational Sciences, University of Tsukuba, Tsukuba, 305–8577 Japan.
a) E-mail: touuyou@gmail.com
b) E-mail: chx@cs.tsukuba.ac.jp
c) E-mail: furuse@cs.tsukuba.ac.jp
d) E-mail: kitagawa@cs.tsukuba.ac.jp

DOI: 10.1587/transinf.2017DAP0007

[*]Without loss of generality, we assume that the minimum values are preferable.

$$\sum_{i=1}^{d} w[i] = 1.$$

The score of product $p$ w.r.t preference $w$ is defined as the inner product between $p$ and $w$, denoted by

$$f(w, p) = \sum_{i=1}^{d} w[i] \cdot p[i].$$

All products are ranked with their scores and we assume that the minimum score is preferable. Now, let $Q$ denote a query set containing a set of products.

Given a query product $q$, for a specific $w$, the rank of $q$ is defined as the number of products such that $f(w, p)$ is smaller than q's score $f(w, q)$; that is,

**Definition 1.** ($rank(w, q)$). *Given a product data set P, a preference w and a query q, the rank of q by w is given by*

$$rank(w, q) = |S|,$$

*where* $S \subseteq P$ *and* $\forall p_i \in S, f(w, p_i) < f(w, q) \wedge \forall p_j \in (P - S), f(w, p_j) \geq f(w, q)$.

The aggregate reverse rank query [2] retrieves the top-$k$ $w$'s which produce $q$ better aggregate rank (ARank) than other $w's$,

**Definition 2.** (*aggregate reverse rank query,* ARR). *Given a product set P and a user preference set W, a positive integer k and a query set Q, the* ARR *query returns the set* $S$, $S \subseteq W$ *and* $|S| = k$ *such that* $\forall w_i \in S, \forall w_j \in (W - S)$, *the identity* $ARank(w_i, Q) \leq ARank(w_j, Q)$ *holds.*

Currently, three aggregate evaluation functions are considered for ARank.

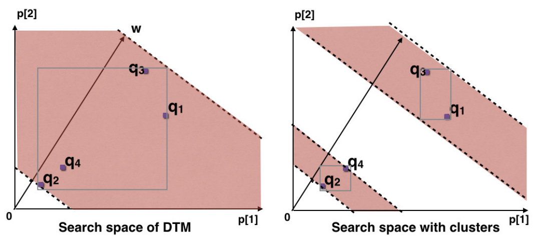
- **SUM** : $ARank_S(w, Q) = \sum_{q_i \in Q} rank(w, q_i)$
- **MAX** : $ARank_M(w, Q) = \underset{q_i \in Q}{\mathrm{Max}}(rank(w, q_i))$
- **MIN** : $ARank_m(w, Q) = \underset{q_i \in Q}{\mathrm{Min}}(rank(w, q_i))$ (1)

The **SUM** function treats each rank equally. By choosing either of **MAX** and **MIN**, it can be avoided that extremely different ranks composed to a worse ARank then that from relatively average ranks. In the rest of paper, we only discuss the processing of first **SUM** aggregate function since the other two functions also share the same technique of our proposal.

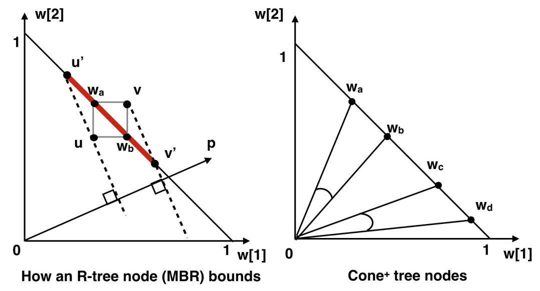
## 1.2 Motivation

The purpose of this research is to address the following two limitations of the state-of-the-art method employed to resolve ARR queries:

**1. The efficiency degrades when $Q$ is distributed widely.** As earlier defined, $Q$ is the query set of bundled products offered by a manufacturer. In practice, the attribute values of the products in $Q$ are not very close as they may not be in the same category. For example, in a product bundling of smartphones and earphones, each price may be quite different. Similarly, book shops always bundle attractive books with some unpopular books, which makes the values of the rating among these books dispersive. The state-of-the-art method for solving ARR queries in [2] is shown in Fig. 2 (a), where the search area is the area sandwiched between the two dashed lines (details are described in Sect. 2). In the worst case, when $Q$ is distributed as wide as the whole space, then the efficiency will degrade to a brute-force search.



(a) Search space of $P$.

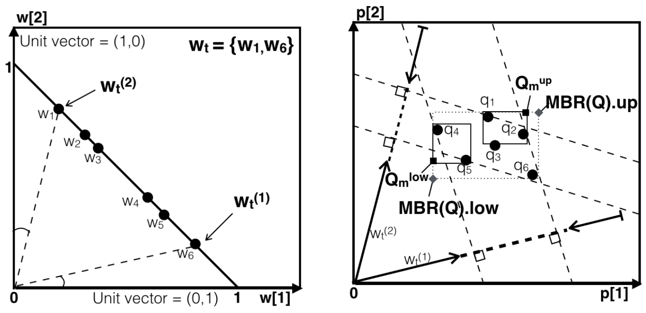


(b) Search space of $W$

**Fig. 2** The limitations of previous method (DTM) [2]: (a) The search space of $P$ spreads when the issued $Q$ is distributed wildly. (b) The search space of $W$ is over-enlarged from range $w \in [w_a, w_b]$ to range $w \in [u', v']$

**2. Only a few of user preferences data $w \in W$ are filtered.** We did a preliminary experiment to observe how many $w \in W$ are filtered with the synthetic data (UN, CL) and the real AMAZON data (refer to Sect. 5). Less than 16% of $w \in W$ are filtered for all data sets, which is not satisfactory because it means that even with an index, more than 84% of data have to be accessed. Furthermore, since the user data sets $W$ are always much larger than the product data $P$, it is very significant to enhance the filtering performance.

Our solution to address the above limitations is designing and combining the following techniques. Imaging that in Fig. 2 (a), $Q$ contains only two groups of queries locate densely around the left-low and right-up corners of the MBR of $Q$. Instead of finding the ARR against $Q$, it is much more efficient to treat the two groups one by one then compose the results. Motivated by this, we first propose a cluster-based method which efficiently processes the situation when $Q$ is distributed widely. To achieve a better filtering of user preferences data, we design a cone$^+$ tree to index $W$ which



(a) Find $W_t$. (b) Find $Q_m^{up}$ and $Q_m^{low}$.

**Fig. 3** Bounding phase. (a) Finding the set $W_t$ of top-$w$ in each dimension. (b) Using $W_t$ to find the upper bound $Q_m^{up}$ and lower bound $Q_m^{low}$ of $Q$.

provides tighter bounds. As in Fig. 2 (b), $w$'s distribute on the line $l: w[1] + w[2] = 1$. Using MBR to index a range $[w_a, w_b]$ will enlarge the search range on $w$ to $[u', v']$ in processing a certain $p$. Where, $u'$ is the intersection of line $l$ with the line passing the left-low of the MBR and perpendicular to vector $p$ ($v'$ is similar). Instead, our proposal indexes exactly the range $[w_a, w_b]$ into a cone$^+$ tree node. (The details of discussion is in Sect. 4)

**Contribution**. This paper makes the following contributions:

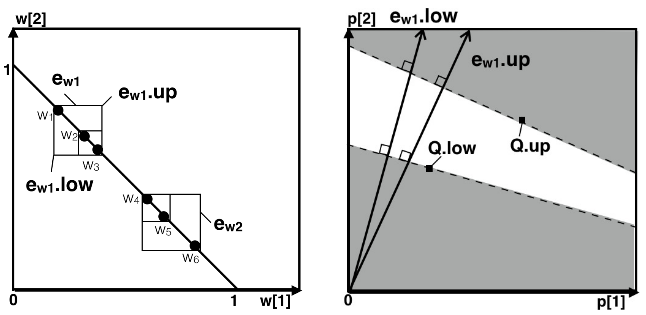
- We demonstrate two limitations of the state-of-the-art method of a widely distributed $Q$ and badly filtered $W$, either of which is a serious problem. To address these limitations, we proposed three solutions: (a) A clustering processing method (CPM) that divides $Q$ into clusters to filter data which cannot be filtered by previous methods and process them integrally to avoid redundant comparison. (b) A novel cone$^+$ tree based method (C$^+$TM) which can filter more $W$ data. (c) A combination method (CC$^+$M) of the above.
- Both the theoretical analysis and experimental results validated the efficiency of the proposed methods.

The rest of this paper is organized as follows: Section 2 describes the state-of-the-art method for ARR query. Section 3 and Sect. 4 introduce the proposed methods. The experimental results are discussed in Sect. 5. Section 6 summarizes related work and Sect. 7 concludes the paper.

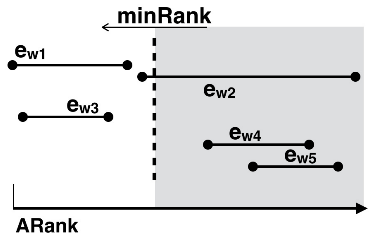
## 2. Bound-and-Filter Framework for Arr Query

In this section, we briefly describe the state-of-the-art solution, referred to in the literature as the *Double Tree Method* (DTM) [2], for ARR queries. DTM is based on a bound-and-filter structure and has two phases: *bounding* and *filtering*.

**1. Bounding.** Figure 3 shows the bounding phase. In this phase, two points $Q.up$ and $Q.low$ are created to bound $Q$. First, a $d$-elements subset of $W$, $W_t = \{w_t^{(i)}\}_1^d$ is built. The element $w_t^{(i)} \in W_t$ is the most similar (in terms of cosine similarity) to the unit vector ($e_i$) of the $i$th dimension. In



(a) R-tree W. (b) Filtering area.



(c) The *minRank*.

**Fig. 4** Filtering phase. (a) R-tree of $W$. (b) Filtering area defined by $e_{w1}.up/e_{w1}.low$ and $Q.up/Q.low$ (the gray parts). (c) Filtering data $W$ with rank bounds and the *minRank*.

other words, $w_t^{(i)} \in W$ and $\forall w \in W, cos(w_t^{(i)}, e_i) \geq cos(w, e_i)$). The two bounds of $Q$, denoted as $Q.up$ and $Q.low$, are found using the following rules:

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

$$Q_l = \{\arg\min_{q\in Q} f(w_t^{(i)}, q)\}_{i=1}^d \tag{3}$$

$$Q.up = \text{MBR}(Q_u).up \text{ and } Q.low = \text{MBR}(Q_l).low \tag{4}$$

where the MBR($Q_u$) is the minimum bounding rectangle of $Q_u$. The MBR($Q_u$).*up* and MBR($Q_u$).*low* are represented the left-low point and right-up point of this MBR, respectively.

**2. Filtering.** Notice that both $W$ and $P$ were pre-indexed independently with two R-trees. Figure 4 (b) shows the filtering phase. For an MBR $e_{w1}$ of R-tree, its upper bound $e_{w1}.up$ forms the upper border line with $Q.up$, while its lower bound $e_{w1}.low$ and $Q.low$ form the lower border line. The only data space of $P$ that must be computed is the area sandwiched between the two dashed border lines marked in gray in the figure. Figure 4 (c) shows how to filter $W$; here, a threshold value denoted by *minRank* is updated with the $k$th smallest rank during processing (this is because we only want the top-$k$ $w$'s such as $e_{w4}$ and $e_{w5}$). The MBRs of $w$'s whose lower bounds rank is greater than *minRank* will be filtered.

## 3. Clustering Processing Method (CPM)

In Sect. 1.2, we introduce the filtering space, which is determined by the previous bound-and-filter method DTM and becomes very small when $Q$ distributes widely. To address

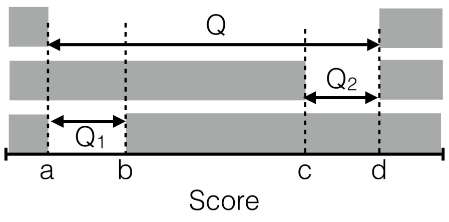


**Fig. 5** The score ranges against the whole $Q$, $Q_1$ and $Q_2$

this limitation, we propose a novel clustering processing method (CPM) which divides $Q$ into clusters and processes them separately so that they filter more data than the DTM does.

## 3.1 Counting Rank Separately

Regarding the situation where $Q$ distributes widely, we can divide $Q$ into clusters instead of treating all $q \in Q$ as a whole. We can then estimate the *ARank* by counting the rank against each cluster. Figure 5 shows the difference of the filtering (gray) area between the whole $Q$ and its clusters $Q_1$ and $Q_2$. There are 4 bounding score $a, b, c, d$, and let $\tilde{a}$ denote the number of points whose score less than $a$. We know that $\tilde{a} \cdot |Q|$ estimates the lower bound of the aggregate rank in [2]. Applying this result to the clusters, the lower bound ($LB$) becomes:

$$LB = \tilde{a} \cdot |Q_1| + \tilde{c} \cdot |Q_2| = \tilde{a} \cdot |Q| + (\tilde{c} - \tilde{a}) \cdot |Q_2| > \tilde{a} \cdot |Q| \quad (5)$$

Obviously, the bound becomes tighter if we estimate the rank separately against clustered $Q$ then sum them up.

A simple implementation of this idea is to count the rank while searching the products R-tree for each cluster and summing them up finally. However, this process needs to access the R-tree index a number of times, which is equals to the number of clusters. We propose a more efficient algorithm named the Clustering Processing Method (CPM), which is also based on the bound-and-filter framework but counts rank for all clusters in only one R-tree traversal.

Algorithm 1 shows the CPM. In CPM, a *buffer* keeps the top-$k$ $w$'s for the result of the ARR query which, initially, is simply the first $k$ $w$'s $\in W$ and their aggregate ranks (Line 1). The value *minRank* is a threshold which changes according as the *buffer*. First, we divide $Q$ into clusters $Q_c$ (Line 2) and, during the bounding phase, we compute the bounds of each cluster (Line 3). Algorithm 2 is called to search with *RtreeP* and count the ARank with clusters $Q_c$ (Lines 7-9). According to the $flag$ returned by Algorithm 2, we update the *buffer* (Line 15) or add the children of $e_w$ for recursion (Lines 11 and 17).

The key point of CPM is to recursively count ARank with clusters $Q_c$ but not independently in Algorithm 2. We use an array of flags (*info*) to mark the state and avoid unnecessary processing; for example, the flag *info*["$Q_1''$"] is true means that the current $e_p$ has been filtered by $Q_1$ and we do not need to consider it again. Parent nodes in the R-tree pass *info* to their children (Line 15). The child node first confirms *info* (Line 3) to decide whether to skip processing (Lines 6 and 18), and updates *info* after it determines that it can filter new clusters (Lines 8 and 13).

**Algorithm 1** Cluster Processing Method (**CPM**)

```
Input: $P, W, Q$
Output: result set $heap$
1: Initialize $buffer$ to store the first $k$ w's in the $ARank(w, Q)$ sorted in
   ascending order.
2: $Qc \Leftarrow$ Clustering($Q$).
3: Get bounds for each $Q_i \in Qc$ // Bounding phase.
4: $heapW.enqueue(RtreeW.root)$ // Filtering phase start.
5: while $heapW$ is not empty do
6:    $e_w \Leftarrow heapW.dequeue()$
7:    $heapP.enqueue(RtreeP.root)$
8:    $minRank \Leftarrow$ the last rank in $buffer$.
9:    $flag \Leftarrow$ ARank-Cluster($heapP, e_w, Qc, minRank$) //Call Algo-
      rithm 2.
10:   if $flag = 0$ then
11:      $heapW.enqueue(e_w.children)$
12:   else
13:      if $flag = 1$ then
14:         if $e_w$ is a single vector then
15:            Update $buffer$ with $e_w$ and its ARank.
16:         else
17:            $heapW.enqueue(e_w.children)$
18: return $buffer$
```

**Algorithm 2** ARank-Cluster

```
Input: $heapP, e_w, Qc, minRank$
Output: include: 1; discard: -1; uncertain : 0;
1: $rnk \Leftarrow 0$
2: while $heapP$ is not empty do
3:    $\{e_p, Info\} \Leftarrow heapP.dequeue()$
4:    if $e_p$ is non-leaf node then
5:       for each $Q_i \in Qc$ do
6:          if $Info[``Q_i''] = false$ then
7:             if $e_p \prec Q_i$ then
8:                $Info[``Q_i''] \Leftarrow true$
9:                $rnk \Leftarrow rnk + e_p.size \times |Q_i|$
10:               if $rnk \geq minRank$ then
11:                  return -1
12:            else if $e_p \succ Q_i$ then
13:               $Info[``Q_i''] \Leftarrow true$
14:            else
15:               $heapP.enqueue(\{e_p.children, Info\})$
16:   if $e_p$ is a leaf node then
17:      for each $Q_i \in Qc$ do
18:         if $Info[``Q_i''] = false$ then
19:            Calculate $f(q_i, w)$ for each $q_i \in Q_i$ and update $rnk$.
20: if $rnk \leq minRank$ then
21:    return 1
22: else
23:    return 0
```

## 3.2 Clustering $Q$

The clustering method is called at Line 4 of the CPM algorithm to divide the query set $Q$ into clusters. In this research, we simply utilize x-means [5], which is a variation of the well-known k-means, as the clustering method. The reasons we choose x-means are: 1) it is not a wise idea to take times to analyze $Q$ and choose among clustering algorithms, so the simple x-means meets this need. 2) since $Q$ is unpredictable and without background knowledge, no other clustering algorithms performance generally better, and 3) x-means can divide $Q$ properly without inputting any parameter (e.g., the number of clusters). Nevertheless, it is still an important future work to find a specific strategy for clustering $Q$.

X-means is a heuristic clustering method, which determines the number of clusters by repeatedly attempting a 2-means subdivision and keeping the best result divisions. The Bayesian Information Criterion (BIC)[†] is used to make the subdivision decision and the lowest $BIC$ is preferred. In conclusion, the procedure of clustering $Q$ with x-means is: (a) Divide $Q$ into 2 clusters. (b) For each cluster, divide it into 2 sub-clusters. (c) if BIC decreases then repeat (b).

## 4. *Cone*$^+$ Tree and Methods

In this section, we address the second limitation in Sect. 1.2. Our solution is to develop the *cone*$^+$ tree index and search methods based on it.

### 4.1 The Over-Enlarged Bound by R-tree

The intrinsic reason for the second limitation pointed out in Sect. 1.2 is that the previous DTM indexes $W$ data with a spatial R-tree then utilizes the MBR to estimate the bound of the score. It is not an appropriate way since the bounding points in spatial MBR will over-enlarge the bound from the actual inner product value.

Figure 6 shows a 2-dimensional example for the over-enlarged bound by R-tree index. The points $w_1 \sim w_4$ are located on a line $L$ (plane in high dimensional space) where $\sum_{i=1}^{d} w[i] = 1$, based on the definitions in Sect. 1.1, We can see that R-tree groups data into their MBR and the right-up and left-low points are used to estimate the upper and lower score bounds, respectively. However, the diagonal crossing MBR.$up$ and MBR.$low$ always makes the largest angle with $L$; thus, for an arbitrary point $p$, the right-up and left-low points of MBR always enlarge (i.e., loosen) the bound of the score unnecessarily.

Inspired by above, we aim at bounding $W$ with a tighter way, which is preferable to group $w's$ with their directions. Therefore, we propose a *cone*$^+$ tree index to pre-index the user data $W$. The *cone*$^+$ tree is a variant of the cone-tree method [6] and, like the latter, it groups data by cosine similarity; in addition, *cone*$^+$ tree stores the boundary points of each node and uses them to calculate the precise score bounds in processing.

### 4.2 Cone Tree and *Cone*$^+$ Tree

Cone tree [6] is a binary construction tree. Every node in the tree is indexed with a center and encloses all points, which are close to this center up to cosine similarity. The node splits into two, left and right, child nodes if it has more

[†]https://en.wikipedia.org/wiki/Bayesian_information_criterion

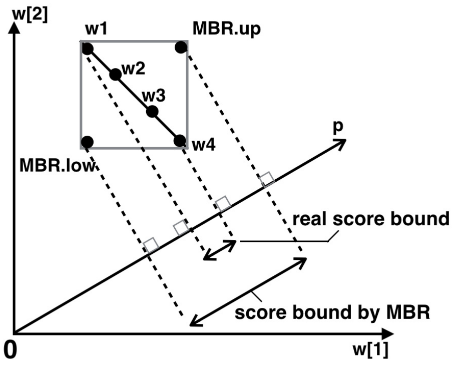


**Fig. 6** The difference of score bounds created by MBR and real score bounds.

points than a set threshold value $M_n$. The tree is built hierarchically by splitting itself until the points are fewer than $M_n$.

In [6], a cone tree was proposed, where the score bounds were based on a cone used to search for the maximum inner product value under the assumption that the length (i.e., norm) of a query is irrelevant to the maximum inner product result. In other words, it is enough to consider only the directions in the cone. Unfortunately, since our problem is different, this assumption on the cone tree and the ways maximum bounding was achieved in [6] does not hold in ARR queries.

From Fig. 6, we can know that the actual bounds of the set of $w$'s are always found from the boundary points. We put forward the following Lemma 1.

**Lemma 1.** *Given a set of preference $B$ and a product $p$, the boundary points of $B$ is $B.boundar$. The preference $w \in B$ which makes the maximum (minimum) score of $f(w,p)$ must be contained in $B.boundar$.*

*Proof.* By contradiction. Assume that $\exists w_a \in B$ and $w_a \notin B.boundar$, where $\forall w_b \in B\ f(w_a,p) \geq f(w_b,p)$. Since the inner product is a monotone function and all values are positive, so $\exists w_a[i] > w_b[i], i \in [1,d]$ and $w_a$ should be a boundary point in $B.boundar$. This leads to the contradiction. □

In the geometric view, the inner product $f(w,p)$ is the length of the projection of $w$ onto $p$. Therefore, both the shortest and the longest projection of a set of $w$'s come from the boundary points of the set.

We took advantage of Lemma 1 and proposed a *cone*$^+$ tree which keeps the boundary points for each node. Therefore the precise score bounds can be computed directly. Figure 7 shows the example of *cone*$^+$ tree. Algorithm 3 and 4 show the construction of *cone*$^+$ tree. The indexed boundary points are the points containing the maximum value on a single dimension (Algorithm 4, Line 2).

### 4.3 *Cone*$^+$ Tree Method (C$^+$TM) and Combined Method (CC$^+$M)

When processing ARR with *cone*$^+$ tree and R-tree in the

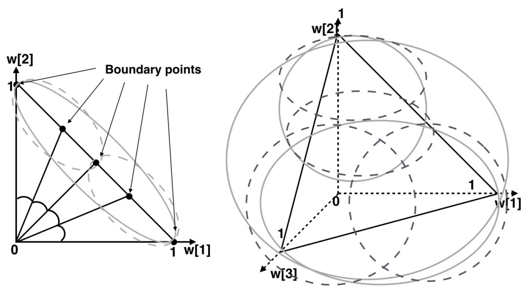


**Fig. 7** Images of 2-dimensional (left) and 3-dimensional (rigth) *cone*$^+$ tree, respectively.

```
Algorithm 3 Cone+TreeSplit(Data)
Input: points set, $Data$
Output: two centering points of children, $a, b$ ;
1: Select a random point $x \in Data$.
2: $a \Leftarrow \arg\max_{x' \in Data} cosineSim(x, x')$
3: $b \Leftarrow \arg\max_{x' \in Data} cosineSim(a, x')$
4: return {a,b}
```

```
Algorithm 4 BuildCone+Tree(Data)
Input: points set, $Data$
Output: $cone^+$ tree, $tree$ ;
1: $tree.data \Leftarrow Data$
2: $tree.boundary \Leftarrow \{w \in Data : \arg\max_{i \in [0,d)} w[i]\}$
3: if $|Data| \leq M_n$ then
4:     return $tree$
5: else
6:     $\{a, b\} \Leftarrow Cone^+TreeSplit(Data)$
7:     $left \Leftarrow \{p \in Data : cosineSim(p, a) > cosineSim(p, b)\}$
8:     $tree.leftChild \Leftarrow BuildCone^+Tree(left)$
9:     $tree.rightChild \Leftarrow BuildCone^+Tree(Data - left)$
10:    return $tree$
```

filtering phase, we can compute the bounds between a *cone*$^+$ and an MBR by the following Theorem.

**Theorem 1.** *(The bounds with cone*$^+$ *and MBR): Given a set of w's in a cone*$^+$ *node* $c_w$, *a set of points in an MBR* $e_p$. $\forall w \in c_w$, $\forall p \in e_p$, $f(w, p)$ *is upper bounded by* $Max_{w_b \in c_w.boundar}\{f(w_b, e_p.up)\}$. *Similarly, it is lower bounded by* $Min_{w_b \in c_w.boundar}\{f(w_b, e_p.low)\}$.

For a query set $Q$ and an MBR $e_p$ of points, the relationship between them on a $w$'s *cone*$^+$ can be inferred from the following.

$$\begin{cases} e_p \prec Q : \\ \quad Max_{w_b \in c_w.boundar}\{f(w_b, e_p.up)\} < \\ \quad Min_{w_b \in c_w.boundar}\{f(w_b, Q.low)\} \\ e_p \succ Q : \\ \quad Min_{w_b \in c_w.boundar}\{f(w_b, e_p.low)\} > \\ \quad Max_{w_b \in c_w.boundar}\{f(w_b, Q.up)\} \\ \text{Unknow} : otherwise \end{cases} \tag{6}$$

The *Cone*$^+$ Tree Method (C$^+$TM) can be implemented easily by indexing $W$ in the *cone*$^+$ tree and using Eq. (6) to apply the bound-and-filter framework described in Sect. 2. We can also combine the features of the two methods in this paper, i.e., the (CC$^+$M) method, which can be implemented by using cone$^+$ tree with $W$ and replacing Lines 7 and 12 in Algorithm 1 with the above Eq. (6).

## 5. Experiment

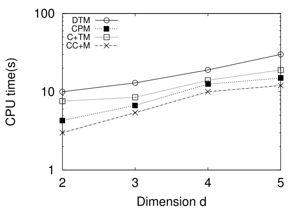
We present the experimental evaluation of the previous DTM [2], the proposed CPM and C$^+$TM, and the method CC$^+$M which combines the features of both CPM and C$^+$TM. All algorithms were implemented in C++, and the experiments were run in-memory on a Mac with 2.6 GHz Intel Core i7, 16 GB RAM.

**Synthetic data**. The synthetic data sets for both $P$ and $W$ are uniform (UN), clustered (CL) and anti-correlated (AC) with an attribute value range of $[0, 1)$. $Q$ is randomly select from $P$. The details of the generation can be found in [1]–[3].

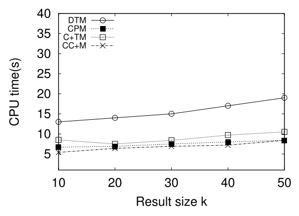
**Real data**. We also have two real data sets: NBA[†] and AMAZON[††]. NBA data set contains 20,960 tuples of players in the NBA from 1949 to 2009. We extracted 5-tuples to evaluate a player with his points, rebounds, assists, blocks, and steals from this NBA statistics. The NBA data is treated as $P$, and we generate user data $W$ with UN. AMAZON is the metadata of products and reviews from the famous AMAZON.com. This metadata has 208,321 user reviews to the products in the category of Movies and TV, in which product bundling often occurs. Each user and product had at least five reviews. For each product in the metadata, we extracted the value on "Price" and "salesRank" as a 2-dimensional vector to represented the data $p \in P$. For a $w \in W$, we computed the average value on "Price" and "salesRank" of the products which the user bought. In both case of NBA and AMAZON data, $Q$ is randomly selected from $P$.

**Experimental results for synthetic data**. Figures 8, 9 and 10 show 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 100K tuples. $Q$ had five query points, and we wanted to find the five best preferences ($k = 5$) for this $Q$. Figures 8 (a), 9 (a), 10 (a) show the runtime on varying dimensionality. CPM and C$^+$TM boost the performance at least 1.5 times in all cases. Figures 8 (b), 9 (b), 10 (b) show that all methods are stable with $k$ from 10 to 50, this is because $k$ is far smaller then the data size. The combined method CC$^+$M takes advantages of them and be the best. We can also see that all methods have better performance in CL data than others, it is because that the CL data can be indexed (R-tree and Cone$^+$ tree) in tighter bounds. We varied the $|Q|$ size to test the performance of clustered processing in CPM, the results are shown in Figs. 8 (c), 9 (c), 10 (c). We also observed the percentage of $W$ been filtered.

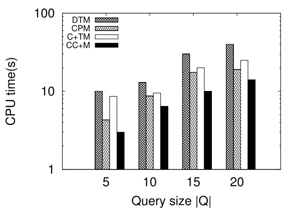
[†]NBA: http://www.databasebasketball.com.
[††]AMAZON: http://jmcauley.ucsd.edu/data/amazon/.



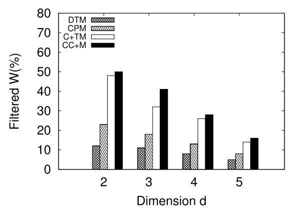
(a) Varying $d$.



(b) Varying $k$.

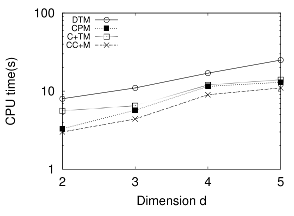


(c) Varying $|Q|$.

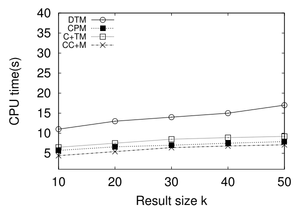


(d) Varying $d$.

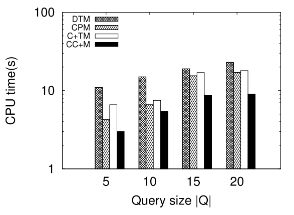
**Fig. 8** Comparison results on UN data, $|P| = |W| = 100K$, $k = 10$.



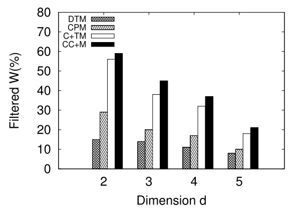
(a) Varying $d$.



(b) Varying $k$.

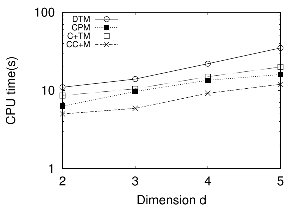


(c) Varying $|Q|$.

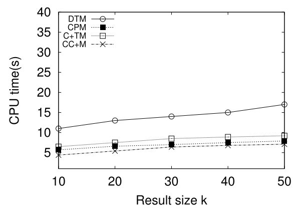


(d) Varying $d$.

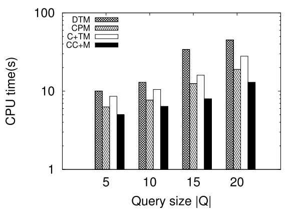
**Fig. 9** Comparison results on CL data, $|P| = |W| = 100K$, $k = 10$.



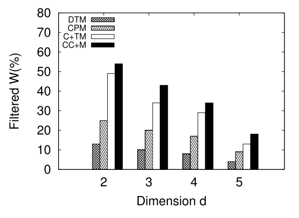
(a) Varying $d$.



(b) Varying $k$.

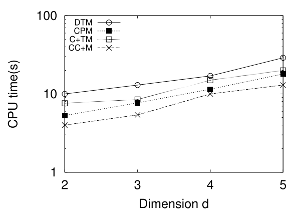


(c) Varying $|Q|$.

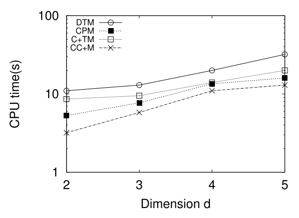


(d) Varying $d$.

**Fig. 10** Comparison results AC data, $|P| = |W| = 100K$, $|Q| = 5$, $k = 10$.

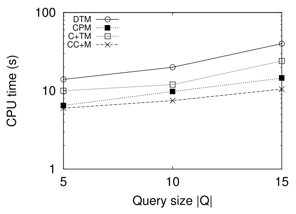


(a) P: UN, W: AC.

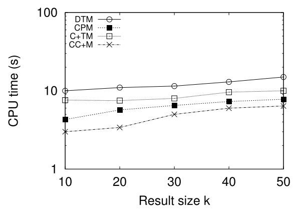


(b) P: AC, W: CL.

**Fig. 11** Comparison results on data with different distribution, $|P| = |W| = 100K$, $k = 10$, $|Q| = 5$.

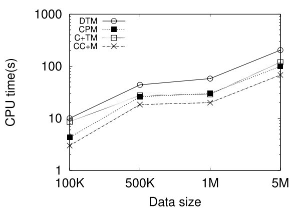


(a) NBA, Varying $|Q|$.

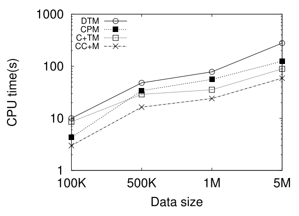


(b) AMAZON, Varying $k$.

**Fig. 12** Comparison results on Real data (NBA, AMAZON).



(a) Varying $|P|$.



(b) Varying $|W|$.

**Fig. 13** Scalability on varying $|P|$ and $|W|$ on UN data, $k = 10$, $|Q| = 5$, $d = 3$.

Figures 8 (d), 9 (d), 10 (d) react the superior filtering of the proposed cone$^+$ tree structure. Figure 11 shows the results on different distribution between $P$ and $W$. In same, proposed methods have better performance.

**Experimental results for real data**. Figure 12 (a) shows the results with the NBA data set on varying $Q$. We selected five, ten, and fifteen players from the same team as $Q$ to deal with a practical query "find users who like an NBA team." As we expected, CC$^+$M is the fastest method. Figure 12 (a) shows the results with the AMAZON data set on varying $k$, it is a good demonstration that our proposals have outstanding performance in marketing applications.

**Scalability**. Figure 13 shows the scalable property for varying $|P|$ and $|W|$. As well as the previous DTM, the proposed CPM, C$^+$TM and CC$^+$M are all scalable, linear and they outperform DTM. A key benefit from the proposed cone$^+$ tree, C$^+$TM and CC$^+$M is that it can be stable even with the a large $|W|$.

**Precision**. We also test the quality of the ARR query with AMAZON data. We randomly selected two products then execute ARR query with three different aggregate functions in Eq. (1). Figure 14 reports the precision of ARR query with the percentage of users in ARR query results who

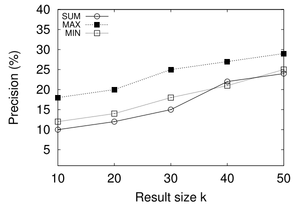


**Fig. 14** The precision of ARR query on AMAZON data with SUM, MIN and MAX aggregate functions.

really have bought either of the selected product. Nevertheless, we can know that the precision depends on the data and the aggregate function, so it is also an important issue that to adjust a suitable aggregate function based on specific data.

## 6. Related Work

**Basic top-*k* query**. The most basic query processing is the top-*k* query. In the user-product model, when given a user preference, the top-*k* query returns *k* products with minimal ranking scores. The research [7] is an important investigation that describes and classifies top-k query processing techniques in relational databases.

**Reverse rank queries (RRQ)**. Conversely, finding potential users for a given product is an equally important field in the user-product model. We call these kinds of queries as the reverse rank queries (RRQ). An example of the RRQ is the reverse top-*k* [4] query, which has been proposed to evaluate the impact of a product based on the preferences of users who treat it as a top-*k* product. For an efficient reverse top-*k* process, Vlachou et al. [3] proposed a branch-and-bound (BBR) algorithm using a boundary-based registration and a tree-based processing. Vlachou et al. In order to resolve the reverse query for some less-popular objects, Zhang et al. [1] proposed another kind of RRQ: the reverse k-rank query, which can find the top-*k* user preferences with the highest rank for a given object among all users. The most relevant research related to this work is the aggregate reverse rank query [2]. Dong et al. [8] also proposed a Grid-index algorithm, which is focused on efficiently processing RRQ with high-dimensional data.

**Reverse queries in spatial data**. For the spatial query problem, there also exist many reverse queries. Given a data point, queries are performed to find result sets containing this data point. Korn et al [9] proposed the reverse nearest neighbour (RNN) query. On the other end, Yao et al. [10] proposed the reverse furthest neighbor (RFN) query to find points where the query point is deemed as the furthest neighbor. The extension of the RNN, which is the reverse *k* nearest neighbor (RKNN), has equally been researched on sufficiently. We remark that Yang et al. [11] analyzed and compared notable algorithms from [12]–[16]. In another direction, Dellis and Seeger introduced the reverse skyline query, which uses the advantages of products to find potential customers based on the dominance of competitors' products [17], [18]. The preference of each user is described as a data point representing the desirable product.

## 7. Conclusion

In this paper, we pointed out two serious limitations of the state-of-the-art method of aggregate reverse rank query and proposed two methods CPM and C$^+$TM to solve them. On the one hand, CPM divides a query set into clusters and processes them to overcome the low efficiency caused by bad filtering when considering the query set as a whole. On the other hand, C$^+$TM uses a novel index *cone*$^+$ tree to avoid enlarging the score bound. Furthermore, we also proposed a method CC$^+$M that utilizes the advantages of both of the above solutions. The experimental results on both synthetic data and real data showed that CC$^+$M has the best efficiency.

As future works, we are finding ways to reduce the computational cost of constructing the cone+ tree. This cost is currently linear to data size. We also want to propose a strategy of clustering for ARR problem to achieve a better efficacy. Moreover, we are considering a wider definition of ARR query with more than proposed three aggregate ARank functions.

## Acknowledgements

This research was partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of RIKEN, Japan.

### References

[1] Z. Zhang, C. Jin, and Q. Kang, "Reverse k-ranks query," Proc. VLDB Endow., vol.7, no.10, pp.785–796, 2014.

[2] Y. Dong, H. Chen, K. Furuse, and H. Kitagawa, "Aggregate reverse rank queries," Database and Expert Systems Applications, Lecture Notes in Computer Science, vol.9828, pp.87–101, Springer International Publishing, Cham, 2016.

[3] A. Vlachou, C. Doulkeridis, and e. Yannis Kotidis, "Branch-and-bound algorithm for reverse top-k queries.," SIGMOD Conference, pp.481–492, 2013.

[4] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag, "Reverse top-k queries," ICDE 2010, pp.365–376, 2010.

[5] D. Pelleg and A.W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000, pp.727–734, 2000.

[6] P. Ram and A.G. Gray, "Maximum inner-product search using cone trees," The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, Aug. 12-16, 2012, pp.931–939, 2012.

[7] I.F. Ilyas, G. Beskales, and M.A. Soliman, "A survey of top-*k* query processing techniques in relational database systems," ACM Comput. Surv., vol.40, no.4, pp.1–58, 2008.

[8] Y. Dong, H. Chen, J.X. Yu, K. Furuse, and H. Kitagawa, "Grid-index algorithm for reverse rank queries," Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017, pp.306–317, 2017.

[9] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," Proc. 2000 ACM SIGMOD, vol.29, no.2, pp.201–212, 2000.

[10] B. Yao, F. Li, and P. Kumar, "Reverse furthest neighbors in spatial databases," Proc. 25th ICDE, pp.664–675, 2009.

[11] S. Yang, M.A. Cheema, X. Lin, and W. Wang, "Reverse k nearest neighbors query processing: Experiments and analysis," Proc. VLDB Endow., vol.8, no.5, pp.605–616, 2015.
[12] I. Stanoi, D. Agrawal, and A. El Abbadi, "Reverse nearest neighbor queries for dynamic databases," ACM SIGMOD Workshop, pp.44–53, 2000.
[13] Y. Tao, D. Papadias, and X. Lian, "Reverse knn search in arbitrary dimensionality," Proc. 13th International Conference on VLDB, pp.744–755, 2004.
[14] Y. Tao, D. Papadias, X. Lian, and X. Xiao, "Multidimensional reverse *k* NN search," The VLDB Journal, vol.16, no.3, pp.293–316, 2007.
[15] M.A. Cheema, X. Lin, W. Zhang, and Y. Zhang, "Influence zone: Efficiently processing reverse k nearest neighbors queries," Proc. 27th ICDE 2011, pp.577–588, 2011.
[16] S. Yang, M.A. Cheema, X. Lin, and Y. Zhang, "SLICE: reviving regions-based pruning for reverse k nearest neighbors queries," IEEE 30th International Conference on Data Engineering, ICDE, pp.760–771, 2014.
[17] X. Lian and L. Chen, "Monochromatic and bichromatic reverse skyline search over uncertain databases," Proc. ACM SIGMOD, pp.213–226, 2008.
[18] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," Proc. 33rd International Conference on VLDB, pp.291–302, 2007.

**Yuyang Dong** received the B.E. degree in computer science, from China University of Mining and Technology, Beijing, China, in 2013. In 2016, he received the M.E. degree in computer science, from the University of Tsukuba, Japan. He is currently a Ph.D. candidate at the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba. His research interests include query processing and optimization, spatial algorithm and streaming processing. He is a student member of the Database Society of Japan (DBSJ).

**Hanxiong Chen** received the B.S. degree from Zhongshan University, Guangdong, China, in 1985, the M.S. degree and the Ph.D. degree in computer science, from the University of Tsukuba, Japan, in 1990 and 1993, respectively. He is currently an assistant professor in the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interests include data engineering, knowledge discovery, data mining and information retrieval. He is a member of ACM, IEEE-CS and IPSJ.

**Kazutaka Furuse** received his B.E., M.E. and Ph.D. degrees in computer science, from the University of Tsukuba, in 1988, 1990, and 1993, respectively. He was an R&D Engineer in RICOH Software Research Center from 1993 to 1996. Then he joined the University of Ibaraki as a research associate until 1999. He is currently an associate professor in the Faculty of Engineering, Information and Systems, the University of Tsukuba. His current research interests include link analysis, scalable query processing, indexing mechanisms, and architecture of advanced database systems.

**Hiroyuki Kitagawa** received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science, all from the University of Tokyo. He is currently a full professor at Center for Computational Sciences and Center for Artificial Intelligence Research, University of Tsukuba. His research interests include data integration, databases, data mining, and information retrieval. He served as President of the Database Society of Japan from 2014 to 2016. He is an IEICE Fellow, an IPSJ Fellow, a member of ACM and IEEE, and an Associate Member of the Science Council of Japan.