Grid-Index Algorithm for Reverse Rank Queries

Yuyang Dong
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
touuyou@gmail.com

Hanxiong Chen
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
chx@cs.tsukuba.ac.jp

Jeffrey Xu Yu
Department of System Engineering and Engineering Management
The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Kazutaka Furuse
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
furuse@cs.tsukuba.ac.jp

Hiroyuki Kitagawa
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
kitagawa@cs.tsukuba.ac.jp

ABSTRACT

In Rank-aware query processing, reverse rank queries have already attracted significant interests. Reverse rank queries can find matching customers for a given product based on individual customers' preference. The results are used in numerous real-life applications, such as market analysis and product placement. Efficient processing of reverse rank queries is challenging because it needs to consider the combination on the given data set of user preferences and the data set of products.

Currently, there are two typical reverse rank queries: Reverse top-k and reverse k-ranks. Both prefer top-ranking products and the most efficient algorithms for them have a common methodology that indexes and prunes the data set using R-trees. This kind of tree-based algorithms suffers the problem that their performance in high-dimensional data declines sharply while high-dimensional data are significant for real-life applications. In this paper, we propose an efficient scan algorithm, named Grid-index algorithm (GIR), for processing reverse rank queries efficiently. GIR algorithm uses an approximate values index to save computations in scanning and only requires a little memory cost. Our theoretical analysis guarantees the efficiency and the experimental results confirm that GIR has superior performance compared to tree-based methods in high-dimensional applications.

CCS Concepts

•Theory of computation → Database query processing and optimization (theory);

(a) User preferences and Top-2

user	w[smart]	w[rating]	Top-2
Tom	0.8	0.2	p3,p2
Jerry	0.3	0.7	p2,p5
Spike	0.9	0.1	p2,p3

(b) Cell phone database and R-Top2

cell phone	p[smart]	p[rating]	R-Top2
p1	0.6	0.7	null
p2	0.2	0.3	Tom,Jerry,Spike
p3	0.1	0.6	Tom,Spike
p4	0.7	0.5	null
p5	0.8	0.2	Jerry

(c) Cell phone ranks and R-1Rank

	Rank in Tom	Rank in Jerry	Rank in Spike	R-1Rank
p1	3	5	3	Tom
p2	2	1	2	Jerry
p3	1	3	1	Tom
p4	4	4	4	Tom
p5	5	2	5	Jerry

Figure 1: Example for *RTK* and *RKR* queries. (a): the top-2 cell phones appreciated by users. (b): the *RT*-2 of each phone. (c): the rank list and the *R*1-*R* of each phone.

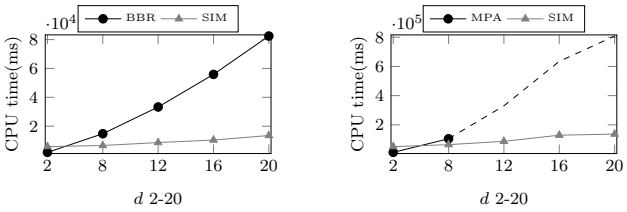
Keywords

Reverse Rank Queries; High-dimensional Data Querying;

1. INTRODUCTION

Top-k queries retrieve top-k products based on a given user preference. As a user-view model, top-k queries are widely used in many applications as shown in [3, 8]. Assuming that there is a dataset of user preferences, reverse rank queries (*RRQ*) have been proposed to retrieve the user preference that causes a given product to match the query condition. From the perspective of manufacturers, RRQ are essential to identify customers who may be interested in their products and to estimate the visibility of their fproducts based on different user preferences. Not limited to the field of product (user) recommendations for e-commerce, this concept of user-product can be extended to a wider range of applications, such as business reviewing, dating and job hunting.

Reverse top-k (*RTK*) [13, 14] and reverse k-ranks (*RKR*) [22] are two typical RRQ queries. Figure 1 shows an example of *RTK* and *RKR* queries. In this example, five different cell phones are scored on how "smart" they are and the "rating". Also, there is a preferences database for three users.



(a) BBR vs SIM. (b) MPA vs SIM.

Figure 2: Performance of tree-base algorithms (BBR, MPA) and Simple scan on varying d (2-20).

Symbol	Description
d	Data dimensionality
P	Data set of products (points)
W	Data set of weighting vectors
q	Query point
$f_w(p)$	The score of p based on w, $f_w(p) = \sum_{i=1}^{d}(w[i] \cdot q[i])$.
$p[i]$	Value of a point $p \in P$ on i'th dimension
$p^{(a)}$	Approximate index vector of a point p
$P^{(A)}$	Approximate index vectors set $\forall p \in P$
n	Number of partitions of value range
$Grid$	Grid-index
$L[f_w(p)]$	Lower bound of score of p on w
$U[f_w(p)]$	Upper bound of score of p on w
$q \prec_w p$	q precedes p based on w

Table 1: Notations and symbols

These preferences are based on a series of weights for each attribute. The score of a cell phone based on a user's preference is found by a weighted sum function that computes the inner product of the cell phone attributes vector and the user preferences vector. Without loss of generality, we assume that minimum values are preferable.

From the values in Figure 1, Tom's score for cell phone p_1 is $0.6 \times 0.8 + 0.7 \times 0.2 = 0.62$. All cell phones' scores are calculated in the same way and ranked. If a cell phone is in the top-k of a user's rank list, then the user is in the result of the *RTK* query for that specific cell phone. In Figure 1 (b), the *RT*-2 results for each cell phone are shown. We can see that p_2's *RT*-2 results are Tom, Jerry and Spike, meaning that all users consider p_2 as an element of their top-2 favorites. Notice that p_1 and p_4 have empty *RT*-2 result sets, which means that every user prefers at least two other phones. [22] believed that it was not useful to return an empty answer and proposed *RKR* query, which find the top-k user preferences whose rank for the given product is highest among all users. In Figure 1(c), p_1 is ranked 3rd by Tom, 5th by Jerry, and 3rd by Spike. In other words, Tom (Spike) ranks p_1 higher than other users, so he is in the answer of the *R1-R* of p_1.

1.1 Notations and Problem Definition

Each product p in the data set P is a d-dimensional vector, where each dimension is a numerical non-negative scoring attribute. p can be represented as a point $p = (p[1], ..., p[d])$, where $p[i]$ is an attribute value on ith dimension of p. The data set of preferences, W, is defined in a similar way. w is a user preference vector for products where $w \in W$, and $w[i]$ is the user defined weight value for the attribute on ith dimension, where $w[i] \geq 0$ and $\sum_{i=1}^{d} w[i] = 1$. The score is defined as an inner product of w and p, which is expressed as $f_w(p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. Notations are summarized in Table 1. The definitions of top-k query and of the two reverse rank queries [13, 22] are re-used here.

Definition 1. *(Top-k query): Given a positive integer k, a point set P and a user-defined weighting vector w, the resultant set $TOP_k(w)$ of the top-k query is a set of points such that $TOP_k(w) \subseteq P$, $|TOP_k(w)| = k$ and $\forall p_i, p_j$: $p_i \in TOP_k(w)$, $p_j \in P - TOP_k(w)$. Therefore, it holds that $f_w(p_i) \leq f_w(p_j)$.*

Definition 2. *(*RTK *query): Given a query point q and k, as well as P and W (dataset of points and weighting vectors respectively), a weighting vector $w_i \in W$ belongs to the reverse top-k result set of q, if and only if $\exists p \in TOP_k(w_i)$ such that $f_{w_i}(q) \leq f_{w_i}(p)$.*

Definition 3. *(*RKR *query): Given a query point q and k, as well as P and W, reverse k-ranks returns a set S, where $S \subseteq W$ and $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $rank(w_i, q) \leq rank(w_j, q)$.*

The $rank(w, q)$ is defined as the number of points with a smaller score than q for a given w.

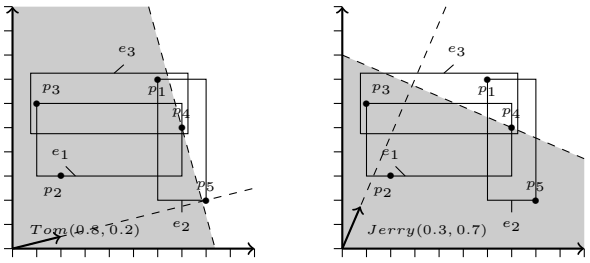
1.2 Motivation and Challenges

To the best of our knowledge, the most efficient algorithm for processing *RTK* is the Branch-and-Bound (BBR) algorithm [17], and the most efficient algorithm for *RKR* is the Marked-Pruning-Approach (MPA) algorithm [22]. Both algorithms use a tree-based methodology, which uses an R-tree to index the data set and prune unnecessary entries through the use of MBRs (Minimum Bounding Rectangles). However, as pointed out by [2, 4, 19], the use of R-tree or any other spatial indexes suffer from similar problems: When processing high-dimensional data sets, the performance declines to even worse than that of linear scan.

Figure 2 shows the comparison of performance between tree-based algorithms (BBR, MPA) and the simple scan (SIM, linear scan). According to the results, SIM outperforms these tree-based algorithms when processing RRQ in high dimensions. The reason for that inefficiency is that tree-based algorithms cannot divide data correctly in high dimensions, causing most of the MBRs to intersect with each other. Thus, even a small range query can overlap with a major proportion of the MBRs.

Figure 3 shows a geometric view of processing *RTK* queries. In this example, suppose that we treat p_4 as the query point q, then a line that crosses q is perpendicular to Tom's weight vector. The points in the gray area have a greater rank than q. Tree-based methodology filters entries that are entirely in the gray area and counts the number of points contained in filtered entries to record the rank of q. However, because q is within overlapping parts of MBRs, the tree-based algorithm cannot filter any parts of MBRs containing the Tom or Jerry's preferences. As a result, it has to go through most entries one by one and compute the scores. In these cases, traversal of the tree-based spatial index is not an efficient method.

For real-world applications, it is a natural requirement to process RRQ for high dimensional data (more than 3). Both the product's attributes and user's preferences are likely to be high-dimensional. For example, cell phones consumers care about many features, such as price, processor, storage, size, battery life, camera, etc. As another example, DIAN-



(a) Tom (b) Jerry

Figure 3: Tree-base methodology processing *RTK* and search space (gray).

PING [1], a Chinese business-reviewing website, ranks restaurants by users' reviews on overall rate, food flavor, cost, service, environment, waiting time, etc. Therefore, processing RRQ with a high-dimensional data set is a significant problem, and due to the so-called "curse of dimensionality", simple scan offers a better performance than R-tree to solve it.

Despite its performances advantages on high-dimensional queries, there are challenges in processing RRQ with the simple scan. RRQ are more complicated queries than simple similarity searches such as the top-k query or the nearest neighbor search, and the time complexity of a naive simple scan method is $O(|P| \times |W|)$. RRQ require that every combination between P and W is checked before obtaining an answer. And this incurs a large number of pairwise computations. A comparison of 10K cell phones and 10K user preferences would necessitate $10K \times 10K = 100M$ computations. As a result, the enormous computational requirements cause the CPU cost to outweigh the I/O cost, which is the opposite of what happen in normal situations. We hold a preliminary experiment to confirm this by measuring the elapsed time for reading different sizes of data, for processing RRQ queries and for the pairwise computations in the inner product. Table 2 shows that the time taken to read different sizes of data file is almost negligible in the RRQ processing. Rather, the major cost of processing RRQ is the pairwise computations. We also found that the proportion of pairwise computations in processing RRQ grew from about 50% in 6-dimensional data to 90% in 100-dimensional data. In conclusion, in contrast to the usual strategy of saving I/O cost in other simple similarity searches, saving CPU computations is the key to process high-dimensional RRQ efficiently.

For the above reasons, we develop an optimized version of the simple scan, called the Grid-index algorithm (GIR) which reduces the amount of multiplication of inner product in the processing. First, We pre-compute some approximate multiplication values and store them into a 2d array named Grid-index. Then we pre-process the data P and W and create the approximate vectors $P^{(A)}$ and $W^{(A)}$ which indicate the index. In the GIR algorithm, we first scan the approximate vectors $P^{(A)}$ and $W^{(A)}$, then use them with the Grid-index to assemble upper and lower bounds, which help to filter most data without multiplications. After the filtering, we only need to refine few remaining data. In the worst case, it costs the I/O time for reading the $P^{(A)}$ and $W^{(A)}$, which is much less than original data and insignificant as concluded above.

[1]http://www.dianping.com

Elapsed time(ms) \ Data size	1K	10K	100K
Reading data	5	26	146
Processing RRQ	240	9311	624318
−Pairwise computations	103	5321	352511

Table 2: Time cost for reading data and processing reverse rank queries with 6-dimensional data.

1.3 Contributions

The contributions of this paper are as follows:

- We elucidate that the simple scan is an appropriate way to process RRQ when processing high-dimensional data. We also demonstrate that CPU cost is the majority cost and that it is much larger than I/O processing. We are the first to conclude that a better approach for processing RRQ is to optimize the scan method.
- We propose a Grid-index, which uses pre-calculated score bounds to reduce multiplications in the simple scan. Based on Grid-index, we propose GIR algorithm which processes *RTK* and *RKR* queries more efficiently. Our method outperforms tree-based algorithms in almost all cases and all data sets, except for those in very low (less than 4) dimensional cases.
- We analyze the filter performance of tree-based algorithms and establish the GIR performance model. Theoretical analysis clarifies the limitation of the tree-based methods. The performance model of proposed GIR guarantees the efficiency of the Grid-index method is achieved at a negligible memory cost.

The rest of this paper is organized as follows: Section 2 summarizes the related work. Section 3 states the Grid-index concept and how to construct upper and lower bounds. In Section 4, we present the formal description of the GIR algorithm. Section 5 analyzes the performance of tree-based algorithms and gives a performance model for the Grid-index. Experimental results are shown in Section 6, and Section 7 concludes the paper.

2. RELATED WORK

For top-k queries, one possible approach to the top-k problem is the Onion technique [3]. This algorithm precomputes and stores convex hulls of data points in layers like an onion. The evaluation of a linear top-k query is accomplished by starting from the outermost layer and processing these layers inwardly. [8] proposed a system named PREFER that uses materialized views of top-k result sets that are very close to the scoring function in a query.

Reverse rank queries (RRQ) are the reverse version of the top-k queries. A typical query of RRQ is the reverse top-k query. [13, 14] introduced the reverse top-k query and presented two versions, namely monochromatic and bichromatic, and proposed a reverse top-k Threshold Algorithm (RTA). [5] indexed a dataset with a critical k-polygon for monochromatic reverse top-k queries in two dimensions. [17]

propose a tree-base, branch-and-bound (BBR) algorithm which is the state-of-the-art approach for reverse top-k query. BBR indexes both data sets P and W in two R-trees, and points and weighting vectors are pruned through the branch-and-bound methodology. For applications, reverse top-k query was used in [16] to identified the most influential products, and in [15] to monitor the popularity of locations based on user mobility.

However, the reverse top-k query has a limitation that returns an empty result for an unpopular product. [22] introduced the reverse k-ranks query to ensure that any product in the data set can find their potential customers. Then proposed a tree-base algorithm named MPA (Marked Pruning Approach), which uses a d-dimensional histogram to index W and an R-tree to index P. Dong et al. [7] indicated that both reverse top-k and reverse k-rank queries were designed for only one product and cannot handle the product bundling. So they defined an aggregate reverse rank query that finds the top-k users for multiple query products.

Other works also considered a given data point and aimed at finding the queries that have this data point in their result set, such as the reverse (k) nearest neighbor (RNN or RKNN) [10, 20] that finds points that consider the query point as the nearest neighbor. RKNN may looks similar to RRQ, but they are actually very different. RKNN evaluates relative L_p distance in one Euclid space with between two certain points. On the other hand, RRQ focus on the absolute ranking value over all products, and the ranking scores are found through inner products of user preferences and products, from two different data spaces.

For other reverse queries, the reverse furthest neighbor (RFN) [21] and its extension RKFN (reverse k furthest neighbor) [18] find points that consider a query point as their furthest neighbor. The reverse skyline query uses the advantages of products to find potential customers based on the dominance of competitors products [6, 11]. However, reverse skyline query uses a desirable product data to describe the preference of a user. But in the definition of RRQ, the preference is described as a weighting vector.

For the space-partition tree-based structure, R*-tree [1], a variation on R-tree, improves pruning performance by reducing overlap in the tree construction. [9] used Hilbert space-filling curves to impose a linear ordering on the data rectangles in R-tree and improve the performance. [2] investigated and demonstrated the deficiencies of R-tree and R*-tree when dealing with high-dimensional data. As an improvement, a superior index structure named X-tree was proposed. X-tree uses a split algorithm to minimize overlap and utilizes the concept of super-nodes. In our opinion, X-tree can be seen as a middle approach between the R-tree and simple scan methods, because it uses the spatial tree structure to process the disjoint parts, and uses linear scan with the overlapping parts. For high-dimensional data, there are very few disjoint parts, causing there to be almost no advantage to the construction and look-up features of the X-tree.

It is well known that the overlapping nodes in high-dimensional space, is a shortcoming of tree structure. R. Weber *et al.* [19] proved that tree-based like [1, 2] is worse than linear scan in high-dimensional data and proposed a VAFILE filtering strategy. They divided the data space into buckets equally and use these buckets' upper and lower bounds to filter candidates. The goal of using VAFILE is to save I/O cost by scanning the bit-compressed file of buckets. However, we purpose to save the CPU computing in RRQ. [4] proposed a technique by "indexing the function" that pre-computing some key values of the L_p-distance function to avoid the expensive computing in high-dimensional nearest neighbour search.

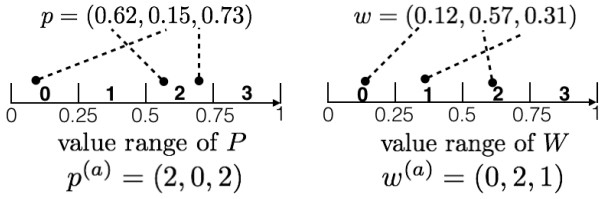


Figure 4: Equally dividing value range into 4 partitions, allocating real values into approximate intervals and getting the approximate vector $p^{(a)}$ and $w^{(a)}$.

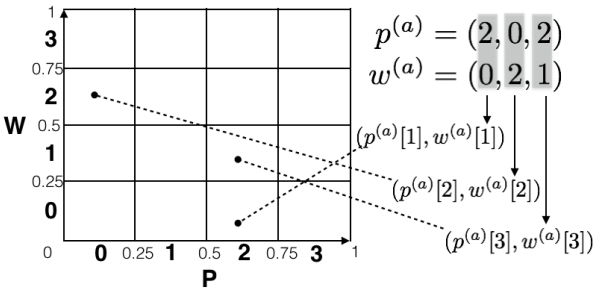


Figure 5: 4×4 Grids for points and weighting vectors, mapping $p^{(a)}$ and $w^{(a)}$ onto Grids.

3. GRID-INDEX

According the statement in Section 1.2, it stands to reason that using a simple scan with high-dimensional data is the most efficient approach. However, in this method, the multiplications of inner products take most of the processing time. We were inspired to study a method that could enhance the efficiency of the simple scan by avoiding multiplications for the inner product. In this section, we introduce the concept of Grid-index, which stores pre-calculated approximate multiplication values. The approximate values can form upper and lower bounds of a score and can be used in a filtering step for the simple scan approach.

3.1 Approximate Values in Grid-index

Concept of Grids. To confirm that the resultant score of the weighted sum function (inner product) is fair, all values in p must be in the same range, so must all values in w. We use this feature to allocate values into value ranges. As Figure 4 shows, in this example we partition the value range into 4 equal intervals. For the given $p = (0.62, 0.15, 0.73)$, the first attribute $p[1] = 0.62$ falls into the third partition $[0.5, 0.75]$. The second, $p[2] = 0.15$, falls into the first partition $[0, 0.25]$. We will store the partition numbers as an approximate vector, denoted as $p^{(a)}$ and $w^{(a)}$, so $p^{(a)} = (2, 0, 2)$ and $w^{(a)} = (0, 2, 1)$.

Since the inner product is the sum of pairwise multipli-

cations of $p[i]$ and $w[i]$, we combine the ranges of p and w to form the grids. Figure 5 illustrates the 4×4 grids in this example. We can map an arbitrary pair of $(p[i], w[i])$ onto a certain grid, and different $(p[i], w[i])$ pairs may share the same grid location. The purpose of mapping the pairs onto the grid is to use the grids' corners to estimate the score of $p[i] \cdot w[i]$. By taking advantage of values having the same range, these grids can be re-used for mapping all pairs $(p[i], w[i]),\ i \in [1, d],\ p \in P$ and $w \in W$.

Construction of Grid-index. Assume that we divide the value range of p and w into $n = 2^b$ partitions, and the position information of all elements in a vector are represented by a (n+1)-element vector α_p for points and α_w for weights. In the example of Figure 4, $\alpha_p = \alpha_w = (0, 0.25, 0.5, 0.75, 1)$. The Grid-index, denoted as $Grid$, is a 2-dimensional array and saves all multiplication results of all combinations between α_p and α_w:

$$Grid[i][j] = \alpha_p[i] \cdot \alpha_w[j],\ i, j \in [0, n] \tag{1}$$

Score Bounds and Precedence. According to the above Grid partition, we pre-store all approximate vectors for P and W, denoted as $P^{(A)}$ and $W^{(A)}$. The approximate vector $p^{(a)}$ for a given p is calculated by $p^{(a)}[i] = \lfloor p[i] \cdot n/r \rfloor$, where r is the range of $p[i]$'s attribute value. $w^{(a)}$ is calculated from w in the same way. Clearly, for a pair $(p[i], w[i])$ in the ith dimension, $Grid[p^{(a)}[i]][w^{(a)}[i]]$ is the lower bound and $Grid[p^{(a)}[i] + 1][w^{(a)}[i] + 1]$ is the upper bound. In the example, $p[1] = 0.62$, $w[1] = 0.12$ and $p^{(a)}[1] = 2$, $w^{(a)}[1] = 0$. Based on Equation (1), $Grid[2][0] = 0.5 \times 0$, $Grid[2+1][0+1] = 0.75 \times 0.25$, meaning $0.5 \times 0 \le p[1] \cdot w[1] \le 0.75 \times 0.25$.

For the inner product $f_w(p) = \sum_{i=1}^{d} p[i] \cdot w[i]$, based on properties of the inner product and features of the Grid-index, we know that:

$$L[f_w(p)] \le f_w(p) \le U[f_w(p)] \tag{2}$$

where $L[f_w(p)]$ and $U[f_w(p)]$, denoting the lower bound and the upper bound of $f_w(p)$, are given by

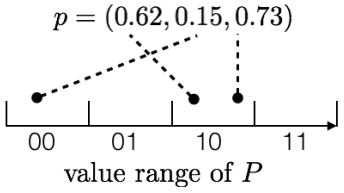
$$L[f_w(p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i]][w^{(a)}[i]] \tag{3}$$

$$U[f_w(p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i] + 1][w^{(a)}[i] + 1] \tag{4}$$

The relationship between p and q can be classified into three cases with the help of $L[f_w(p)]$ and $U[f_w(p)]$:

- Case 1 ($p \prec_w q$): If $U[f_w(p)] < f_w(q)$, p precedes q, p has a higher rank than q with w.
- Case 2 ($q \prec_w p$): If $L[f_w(p)] > f_w(q)$, q precedes p, p does not affect the rank of q with w.
- Case 3 ($p \asymp q$): Otherwise, p and q are incomparable, i.e., $L[f_w(p)] \le f_w(q) \le U[f_w(p)]$. The Grid-index cannot define whether p or q ranks higher with w.

Filtering Strategy. We scan the approximate vectors first, then use the Grid-index to obtain $L[f_w(p)]$ and $U[f_w(p)]$, and filter points that satisfy either Case 1 or Case 2 above. After scanning, if necessary, we carry out a refining phase, and compute the real score for all points in Case 3. Notice that throughout this process, we only calculated the sum and retrieved $L[f_w(p)]$ and $U[f_w(p)]$ of Equations (3) and (4). If a point p is in Case 1 or Case 2, we do not need to compute the real score $f_w(p)$, thus saving computational costs with multiplications to find the inner product.



p	0.62 0.15 0.73
$p^{(a)}$	2 0 2
bit	100010

Figure 6: 6-bit string for compressing the p to $p^{(a)}$.

3.2 Compress the Approximate Vectors

Storing all approximate vectors incurs extra storage costs for data sets P and W. To compress this storage, each approximate vector can be presented by a bit-string describing the interval which its elements fall. Figure 6 shows an example where the approximate vector $p^{(a)}$ is saved as a 6-bit string (100010), because 2 bits are needed to define 4 partitions for each of the 3 dimensions. Generally, if we divide the value range into 2^b partitions, then a $(b \times d)$-bit string is needed to store an approximate vector. According to the analysis in Section 5.3, $b = 6$ is enough for a good filtering performance. Usually, the original data is a 64-bit float value, so the storage overhead by the compressed 6-bit data is less than 1/10 of the original data [2]. This kind of bit-string compressing technique is also used in [19].

Reading approximate vectors with bit-string binary compression only has half the time costs compared to regular I/O operations. However, the superiority of I/O cost can be ignored because the CPU cost is far greater than the I/O cost in RRQ, as discussed in Section 1.2.

It may be argued that it would be the most efficient to store all the scores of each p and w directly. In reality, storing that amount of data is impossible due to the immense cost. For example, assume that there are $10K$ products and $10K$ weight vectors. For Grid-index, 20K tuples are needed to store the approximate vectors, but it would take $10K \times 10K = 100M$ tuples to store all the scores. The storage overhead for storing all scores is thousands of times of the approximate vectors in the proposed Grid-index method.

4. THE GIR ALGORITHM

Next, we use the Grid-index methodology to propose two versions of Grid-indexing algorithm for *RTK* and *RKR* queries. The two algorithms can be implemented easily by using the GInTop-k function that efficiently obtains the rank of query point q on a certain input w.

4.1 GInTop-k Function Based on Grid-index

Algorithm 1 describes the GInTop-k function based on Grid-index. GInTop-k scans each approximate vector $p_j^{(a)} \in P^{(A)}$ – *Domin*. *Domin* is a global variable denoting a buffer

[2]When $n = 2^b$, then the storage cost for the approximate vectors are $|P^{(A)}| = \frac{b}{64}|P|$ and $|W^{(A)}| = \frac{b}{64}|W|$, if P and W's attributes are float values.

Algorithm 1 Grid-index checking q's rank (GInTop-k)

Require: $P^{(A)}, w_i^{(a)}, q, k, Grid$, $Domin$
Ensure: -1: discard w_i, rnk: include w_i
1: $Cand \leftarrow \emptyset$
2: $rnk \leftarrow Domin.size$
3: **for** each $p_j^{(a)} \in P^{(A)} - Domin$ **do**
4: Calculate $U[f_w(p_j)]$ by Eq. (4)
5: **if** $U[f_{w_i}(p_j)] \leq f_{w_i}(q)$ **then**
6: rnk ++ // $(p_j \prec_w q)$
7: **if** $p_j \prec q$ **then**
8: $Domin \leftarrow Domin \cup \{p_j\}$
9: **if** $rnk \geq k$ **then**
10: **return** -1
11: **else**
12: Calculate $L[f_w(p_j)]$ by Eq. (3)
13: **if** $L[f_{w_i}(p_j)] \leq f_{w_i}(q) \leq U[f_{w_i}(p_j)]$ **then**
14: $Cand \leftarrow Cand \cup \{p_j\}$ // $(p_j \asymp q)$
15: Refine $Cand$: compare real score and updating rnk.
16: **if** $rnk \geq k$ **then**
17: **return** -1
18: **else**
19: **return** rnk

recording dominating points. If p is in $Domin$, then every attribute of a point p is smaller than the corresponding attribute in q ($\forall p[i], i \in (0, d) : p[i] < q[i]$). Once q is given, points in $Domin$ always rank better than q. During scanning, the number of points that rank better than q are counted by rnk, which is initialized by the size of $Domin$ (line 2). The upper bound for the score is obtained using Grid-index (line 4). If the upper bound is smaller than the score of q (Case 1, line 5), then p_j must have a better rank than q for the weighting vector w_i, hence rnk increases by 1 (line 6). Anytime p_j is found dominating q, denoted by $p_j \prec q$, p_j will be appended to $Domin$ (line 7-8). Whenever rnk reaches k (line 9), there are at least k points that rank better than q, thus the current w_i is not a result of RTK of q (-1 is returned). Otherwise, we get a lower bound from the Grid-index (line 12). If $q's$ score is between the lower bound and upper bound of p_j's score (in Case 3, line 13), then p_j is added to $Cand$ for further refinement(line 14). After scanning $P^{(A)}$, if the algorithm did not return a decision, then a refinement step is necessary to establish (line 15). We check the original data of the points held in $Cand$ and refine rnk in the same way, terminating immediately when it reaches k.

Computing $f_w(p)$ requires d multiplication operations and d addition operations. However, to find $U[f_w(p)]$ and $L[f_w(p)]$, it is only necessary to carry out d addition operations. Therefore, our approach will save d times of multiplication if $U[f_w(p)] \leq f_w(q)$ and the algorithm uses the branch at lines 5-10. When $U[f_w(p)] \geq f_w(q)$, our approach requires another d addition operations to find $L[f_w(p)]$, that is, an equivalent amount of additions to replace the multiplication operations in the evaluation of $f_w(p)$. In conclusion, using this method will save computational cost if any point can be filtered by the Grid-index. Section 5.3 proves that a low cost Grid-index can be used to filter over 99% of points.

4.2 Grid-index Algorithm

Now we introduce how Grid-index is applied to RRQ. Algorithm 2 and Algorithm 3 give the implementation of RTK and RKR.

For each approximate vector of $w_i^{(a)} \in W^{(A)}$, Algorithm 2 receives the result of filtering performed by GInTop-k (line 4). If the current w_i ranks q in its top-k, then w_i will be added into the result set of $RTOPk(q)$ (Line 5-6). If there exists more than k dominating points of q, the algorithm returns an empty set because q cannot be part of the top-k for any weighting vector w (line 7-8).

Unlike RTK, a heap structure of size k, denoted by $heap$, and a value $minRank$ are introduced in Algorithm 3 for processing the RKR. For each $w_i^{(a)} \in W^{(A)}$, function GInTop-k is called first, $minRank$ is passed to GInTop-k and used for filtering (line 5). If q ranks in the top-$minRank$ (line 6), we insert w_i and rnk into the $heap$. The last rank of $heap$ is pushed out after it holds more than k elements (line 7). Meanwhile, $minRank$ is updated by the current last rank of $heap$ (line 8). This ensures a self-refined bound and keeps the current k best results from W in $heap$. Finally, when the algorithm terminates, $heap$ is returned as the result set.

Algorithm 2 Grid-index Reverse top-k (GIRTop-k)

Input: $P^{(A)}, W^{(A)}, q, k$
Output: RTK result set $RTOPk(q)$
1: create $Grid$ (Grid-index)
2: $Domin \leftarrow \{\emptyset\}$
3: **for** each $w_i^{(a)} \in W^{(A)}$ **do**
4: $rnk \leftarrow$ GInTop-$k(P^{(A)}, w_i^{(a)}, q, k, Grid, Domin)$
5: **if** $rnk \neq -1$ **then**
6: $RTOPk(q) \leftarrow RTOPk(q) \cup \{w_i\}$
7: **if** $Domin.size \geq k$ **then**
8: **return** $\{\emptyset\}$
9: **return** $RTOPk(q)$

Algorithm 3 Grid-index Reverse k-ranks (GIRk-Rank)

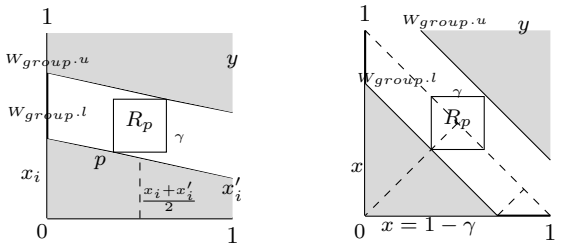
Input: $P^{(A)}, W^{(A)}, q, k$
Output: $heap = RKR$ result set
1: create $Grid$ (Grid-index)
2: $heap \leftarrow \{\emptyset\}$, $Domin \leftarrow \{\emptyset\}$
3: $minRank \leftarrow \infty$
4: **for** each $w_i^{(a)} \in W^{(A)}$ **do**
5: $rnk \leftarrow$ GInTop-$k(P^{(A)}, w_i^{(a)}, q,$ $minRank,$ $Grid,$ $Domin)$
6: **if** $rnk \neq -1$ **then**
7: $heap.insert$ $(w_i,$ $rnk)$
8: $minRank \leftarrow heap$'s last rank.
9: **return** $heap$

5. PERFORMANCE ANALYSIS

In this section, we first analyze the weakness of tree-based algorithms for RRQ. We then build a cost model for Grid-index that finds the ideal number of grids ($n \times n$), guaranteeing that specified filtering performance.

5.1 The Difficulty of Space-division in High Dimensional Data

We first observe the influence of the number of divisions through a space-division index. According to [22], MPA



(a) Trapezoidal prism (b) Tetrahedron

Figure 7: Two kinds of Filtering areas (gray) of R-tree.

uses a d-dimensional histogram to group all weighting vectors W into *buckets*. Each dimension is partitioned into c equal-width intervals, in total, there are c^d *buckets*. As [22] suggests, $c = 5$, If $|W| = 100K$ with the 3-dimensional data, W is grouped in $5^3 = 125$ *buckets*. However, if $d = 10$, then there are $5^{10} \approx 9$ million *buckets*. It is not logical to filter only 100K weight vectors by testing the upper and lower bounds of such a huge number of *buckets*. In this case, scanning one by one would be more efficient.

5.2 Analysis of R-tree Filtering Performance

We test some range queries (within 1% area of the data space) over different d with an R-tree and observe the MBRs. Table 3 shows the average value of accessed MBRs' attributes. Not surprisingly, when $d > 6$, all (100%) of MBRs overlap in the query range, which means that all entries will be accessed during processing. As mentioned in Section 1.2, it is a shortcoming of tree-based algorithms that the MBRs will always overlap with each other when the data is high-dimensional.

Besides the shortcoming from the tree-based index itself, we also found that the filterable space of RRQ with tree-based methodology reduces as the dimensionality increases. This conclusion is supported by the following estimation.

Consider a tree-based algorithm that constructs an R-tree for the products P and assume that R_p is a MBR of this R-tree. In query processing, for each group of w's (denoted as W_{group}), points within R_p are checked. The upper and lower bounds of $f_{W_{group}}(R_p)$ are determined by the borders of W_{group} and R_p. As Figure 7 shows, The gray area is the safely filtered space. The shape of the gray area can be a hyper-prism, a hyper-tetra or a combination of the two. It means that in some of the dimensions (denoted as g) the area will be a triangle, while a trapezoid in others. Assume that the two kinds of shapes are separated clearly; then the proportion of filtered values can be obtained by measuring the volume:

$$Vol = Vol_{TetraX} \cdot Vol_{PrismX} + Vol_{TetraY} \cdot Vol_{PrismY} \quad (5)$$

To give an analytical result, we assume that R_p is in the centroid, so the two filtering areas are equal ($Vol_{TetraX} = Vol_{TetraY}$). Then the volume becomes

$$Vol = 2 \cdot Vol_{Tetra} \cdot Vol_{Prism} \quad (6)$$

Firstly, the volume of hyper-tetra is: [3]

$$Vol_{Tetra} = \frac{1}{g!}(\prod_{i=1}^{g} x_i) = \frac{1}{g!}(1-\gamma)^g \quad (7)$$

then, the volume of the hyper-prism (the area in Figure 7 (a)) is:

$$S_i = \frac{1}{2}(x_i + x'_i) \cdot H \leq (\frac{1-\gamma}{2}) \leq \frac{1}{2} \quad (8)$$

where $H = 1$ is the length of the side. Imagine a 3 dimensional trapezoidal prism in the figure, the volume is:

$$Vol_{Prism3d} = \frac{1}{3}(S_1 + S_2 + \sqrt{S_1 S_2}) \cdot H \leq \frac{1}{2} \quad (9)$$

This result holds for higher dimensional trapezoidal prisms. Consequently, the maximum volume gives the filtered area.

$$Vol_{max} = 2 \cdot \frac{1}{g!}(1-\gamma)^g \cdot \frac{1}{2} = \frac{1}{g!}(1-\gamma)^g \quad (10)$$

It is reasonable to assume that in half of the dimensions the filtered area is hyper-tetra in shape. We will consider a dataset of $d = 10$, $g = 5$, according to Equation (10), R-tree based methods can only filter at most $\frac{1}{5!}$ =0.8% of the data space.

This clearly shows that the space filtered by R-trees in RRQ becomes very small when encountering high-dimensional data. For all points in the space which can not be filtered, each $w[i] \cdot p[i]$ must be calculated and compared with that of the query point.

5.3 The Performance Model of Grid-index

To build a model of our Grid-index, we make the following assumption about the d-dimensional point data set: Values in all dimensions are independent of each other, and the sub-score in each dimension ($w[i] \cdot p[i]$) follows a uniform distribution. Both value ranges of P and W are divided into n partitions for the Grid-index.

Let the probability of a score S falling into a certain interval $(a,\ b)$ be $Prob(a < S < b)$, where $(a,\ b)$ is created by Grid-index. Data points with scores outside of $(a,\ b)$ can be filtered. We denote the filtering performance F by:

$$F(a,\ b) = 1 - Prob(a < S < b). \quad (11)$$

For example, if the probability of a point falling in an interval is 5%, then we say that the filter performance is 95%.

Obviously, $F(a,\ b)$ from Grid-index depends on the density of the grids ($n \times n$). More partitions n lead to smaller $Prob(a < S < b)$ and better filtering performance. However, larger n requires more memory, so it is important to find a suitable n that balances these factors. For this purpose, we first establish specific score properties and then define the relationship between F and n.

For the case of one dimension, dividing the range into equally n^2 partitions, the probability of a point p's score falling into a certain interval is obviously:

$$Prob(\frac{k}{n^2} < w \cdot p < \frac{k+1}{n^2}) = \frac{1}{n^2},\ k = 1, 2, ..., n^2. \quad (12)$$

[3]Recall that the area of a right triangle is $s = \frac{x_1 x_2}{2}$, and a tetrahedron has volume $v = \frac{x_3 s}{3} = \frac{x_1 x_2 x_3}{3 \cdot 2}$. if for (d-1) dim, the volume is $V_{d-1} \sim cx^{d-1}$ then $V_d = \int V_{d-1} dx \sim \frac{cx^d}{d}$.

Dimensionality	3	6	9	12	15	18	21	24
#MBR	1501	1480	1470	1470	1439	1479	1458	1456
diagonal length	4057.7	11744.3	19559.1	23807.9	31010.9	33717.1	36979.2	40515
Shape*	24.9	13.8	8.9	6.4	4.8	4.6	4.7	4.4
Overlaps in Query(1%)	30%	99.8%	100%	100%	100%	100%	100%	100%
Volume	$2.89 \times e^{9}$	$1.39 \times e^{21}$	$3.65 \times e^{33}$	$1.72 \times e^{45}$	$1.08 \times e^{58}$	$5.31 \times e^{69}$	$2.16 \times e^{81}$	$2.28 \times e^{93}$

* Shape is the ratio of the longest edge against the shortest one of an MBR.

Table 3: Observation of accessed MBRs of R-tree in query. 100K points indexed in R-tree, each MBR has 100 entries.

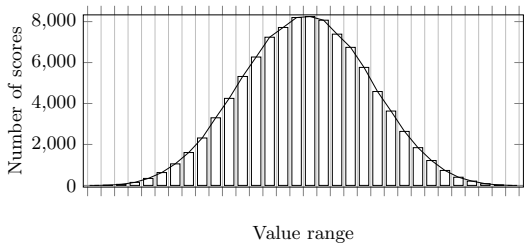


Figure 8: Grid-index scores distribution in dimension $d = 4$, partitions $n = 4$, $|P| = 100$K, $|W| = 100$K.

Now, we want to estimate the probability of p's score ($\sum_{i=1}^{d} w[i] \cdot p[i]$) falling in a score range obtained by Grid-index. For the discrete d dimension case:

$$Prob(\sum_{i=1}^{d}(w[i] \cdot p[i]) = s) \tag{13}$$

This probability can be found by the so called "Dice Problems": Rolling d n^2-sided dice and find the probability of obtaining s score. In this problem, a n^2-sided die corresponds to the score range of a single dimension which is equally partitioned in n^2 parts by Grid-index. The number of dice corresponds to the number of dimensions d, and the scores by rolling d dice becomes the point's score.

The number of ways obtaining score s is the coefficient of x^s in:

$$t(x) = (x^1 + x^2 + ... + x^{n^2})^d \tag{14}$$

By [12], the probability of obtaining s score on d n-sided dice is

$$Prob(s, d, n) = \frac{1}{n^{2d}} \sum_{k=0}^{\lfloor (s-d)/n^2 \rfloor} (-1)^k \binom{d}{k} \binom{s - n^2 k - 1}{d - 1} \tag{15}$$

The filtering performance of Grid-index can be presented by $1 - Prob(s, d, n)$. However, it is difficult to analyse the relationship between n and the filtering performance by Equation (15). On the other hand, we found that the distribution of scores approaches a normal distribution, even in low dimensional cases, such as 4. Figure 8 shows the observation of distribution of scores computed by Grid-index with $n = 4$ partitions, and the dimension $d = 4$. This encourages us to approximate the feature by normal distribution.

For a point p, $p[i] \cdot w[i]$ obeys a uniform distribution with range [0, r), average value μ and standard deviation σ, where

$$\mu = \frac{1}{2}r \qquad \sigma = \frac{1}{2\sqrt{3}}r \tag{16}$$

The average score value of a point p is

$$\overline{p \cdot w} = \frac{1}{d}\sum_{i=1}^{d}(p[i] \cdot w[i]) \tag{17}$$

By the central limit theorem, we have the following approximation when d is sufficiently large.

LEMMA 1. *(Score Distribution). The following random variable*

$$Z = \frac{\sqrt{d}}{\sigma}(\overline{p \cdot w} - \mu) \tag{18}$$

follows the standard normal distribution (SND). In other words, $Z \sim N(0, 1)$, where μ and σ are as in Equation (16).

Note that $d \cdot \overline{p \cdot w}$ is the score of point p. Representing it by a random variable S, S follows a normal distribution with mean $\mu' = \mu d$ and standard deviation $\sigma' = \sigma\sqrt{d}$. By Equation (16),

$$\mu' = \frac{1}{2}rd \qquad \sigma' = \frac{\sqrt{d}}{2\sqrt{3}}r \tag{19}$$

From Lemma 1 and (11), we may now estimate the filtering performance.

LEMMA 2. *(Filtering performance). The filtering performance of Grid-index, F, is given by*

$$\begin{aligned} F(x, x + \Delta) &= 1 - Prob(x < S < x + \Delta) \\ &= 1 - \int_{x}^{x+\Delta} f(x)dx \end{aligned} \tag{20}$$

where

$$f(x) = \frac{1}{\sigma'\sqrt{2\pi}} exp(-\frac{(x - \mu')^2}{2\sigma'^2}) \tag{21}$$

is the probability density function of $N(\mu',\ \sigma')$.

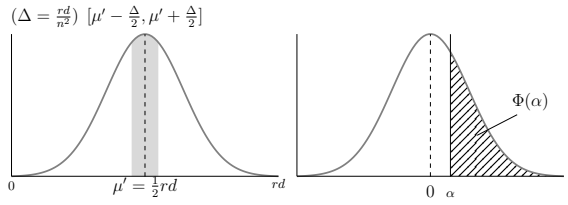
It is difficult to calculate the integral, but by rewriting Z in Lemma 1, The above equation can be:

$$Z = \frac{d \cdot \overline{p \cdot w} - \mu d}{\sigma\sqrt{d}} = \frac{S - \mu'}{\sigma'} \tag{22}$$

we can map S to $Z \sim N(0,\ 1)$ and need only to look up the SND table.

We are now ready to estimate the filtering performance of the Grid-index methodology. Recall that the score of a point is the sum of d addends. The score's range in each dimension is $[0, r)$, and it is equally divided into n^2 partitions. Thus, the value range computed by Grid-index of a d-dimensional points corresponds to range Δ:

$$\Delta = \frac{r}{n^2}d \tag{23}$$



(a) (b)

Figure 9: (a): The normal distribution of point scores $N(\mu', \sigma')$ and the largest probability interval (gray). (b): $\Phi(\cdot)$ of the SND showing $1 - \int_{-\alpha}^{\alpha} \cdot = 2\Phi(\alpha)$.

Our purpose is to find the number of partitions n which guarantees a certain filtering performance F in Lemma 2. To do this, it is sufficient to show the worst case. By Lemma 2, scores that fall within the interval illustrated by the gray part in Figure 9(a) which is located on either side of μ, have the largest probability and thus gives the worst F. Concentrating on this worst interval $[\mu' - \frac{\Delta}{2}, \mu' + \frac{\Delta}{2}]$, by Equation (22) and Equation (19), we find that $S_\Delta = \mu' \pm \frac{\Delta}{2}$ corresponds to

$$Z_\Delta = \frac{S_\Delta - \mu'}{\sigma'} = \frac{\mu' \pm \frac{\Delta}{2} - \mu'}{\sigma'} = \pm\frac{\sqrt{3d}}{n^2} \quad (24)$$

From Lemma 1, $Z \sim N(0, 1)$, the filtering performance in the worst case can be given by

$$F(x, x+\Delta) > F_{worst}(x, x+\Delta) = 1 - \int_{\mu' - \frac{\Delta}{2}}^{\mu' + \frac{\Delta}{2}} f(x)dx = 2\Phi(\frac{\sqrt{3d}}{n^2}) \quad (25)$$

where $\Phi(\cdot)$ is the area shown in Figure 9 (b).

The above discussion leads to the following result.

THEOREM 1. *Given $\epsilon < 1$, the filtering performance of n partitions is guaranteed to be above 1- ϵ in Grid-index such that*

$$n > \sqrt{\frac{2\sqrt{3d}}{\delta}} \quad (26)$$

where δ is determined by looking up the SND table at $(1 - \epsilon)/2$, that is,

$$\Phi(\frac{\delta}{2}) = \frac{1 - \epsilon}{2} \quad (27)$$

PROOF. *By Equation (26), $\frac{\delta}{2} > \frac{\sqrt{3d}}{n^2}$. Since Φ is a monotonically decreasing function (Figure 9), $\Phi(\frac{\sqrt{3d}}{n^2}) > \Phi(\frac{\delta}{2})$. Combining Equation (25) and Lemma 2, we have $F > 2\Phi(\frac{\delta}{2}) = 1 - \epsilon$* □

Example. To ensure that Grid-index filters out over 99% data, we set $\epsilon = 1\%$ ($\frac{(1-\epsilon)}{2} = 0.495$), thus the filtering performance is guaranteed to be better than $F_{worst}(\delta) = 99\%$. Looking up this value in the SND table, we have $\Phi(0.0125) = 0.495$, hence, $\delta = 0.025$. By Theorem 1, the sufficient number of partitions n is calculated by

$$\frac{\sqrt{3d}}{n^2} < \delta = 0.0125 \quad \longrightarrow n > \sqrt{\frac{2\sqrt{3d}}{\delta}} = \sqrt{80\sqrt{3d}} \quad (28)$$

W \ P	Uniform	Normal	Exponential
Uniform	99.3%	98.3%	99.0%
Normal	98.8%	96.5%	98.7%
Exponential	99.2%	97.5%	98.9%

Table 4: Filtering performance of Grid-index with different distributions. $|P| = 100K$, $|W| = 100K$, $d = 6$, $n = 32$

.

Parameter	Values
Data dimensionality d	$2 \sim 50$, **6**
Distribution of data set P	**UN**,CL,AC,RE
Data set cardinality $\|P\|$	50K,**100K**,1M,2M,5M
Distribution of data set W	**UN**,CL,RE
Data set cardinality $\|W\|$	50K,**100K**,1M,2M,5M
Experiment times	1000
Number of clusters	$\sqrt[3]{\|P\|}$, $\sqrt[3]{\|W\|}$
Variance σ_W^2,σ_P^2	0.1^2
Number of grids, n^2	4^2,8^2,16^2,32^2,64^2,$\mathbf{128^2}$
k (top-k and k-ranks)	**100**,200,300,400,500

Table 5: Experimental parameters and default values(in bold) .

If $d = 20$ then $n = 32$ satisfies Equation (28) hence a 32×32 Grid-index is enough for filtering over 99% data. The necessary memory is less than 8 K ($32 \times 32 \times 8$) Bytes.

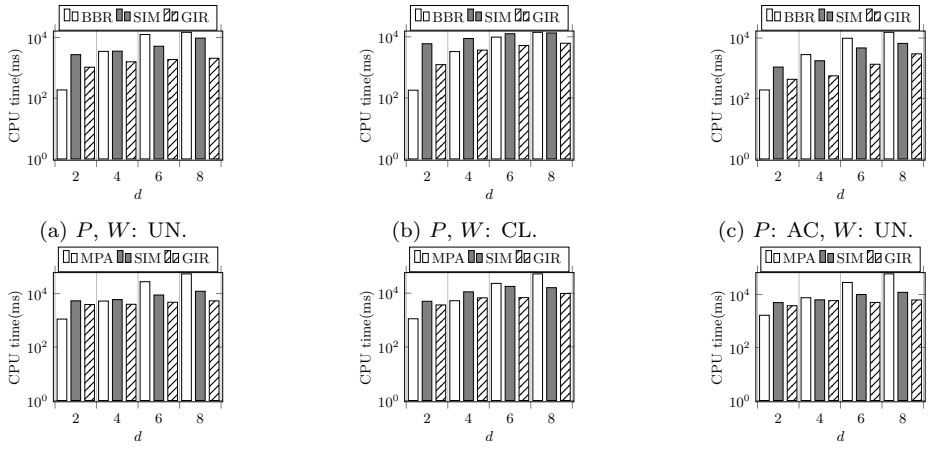
Theorem 1 is still true when $w[i] \cdot p[i]$ follows other distributions. The only difference is that a new μ_i and $\frac{\sigma_i}{\sqrt{d}}$ would have to be estimated, which would lead to a different partition n. We observed the filtering performance on some typical distributions, including the normal distribution ($\sigma = 10\%$) and exponential distribution ($\lambda = 2$). The filtering power of the Grid-index is shown in Table 4. Different σ between these distributions lead to slight differences in filtering power. But the filtering power is always efficient.

6. EXPERIMENT

In this section, we present the experimental evaluation. All algorithms are implemented in C++ and experiments are run on a Mac with a 2.6 GHz Intel Core i5 processor, 8GB RAM, 500GB flash storage space. We pre-read the R-tree, data sets P and W, approximated vectors P_A and W_A and the Grid-index into memory. According to Table 2, the I/O time is not relevant, so we focus on comparing our work only in terms of CPU processing time.

6.1 Experimental Setup

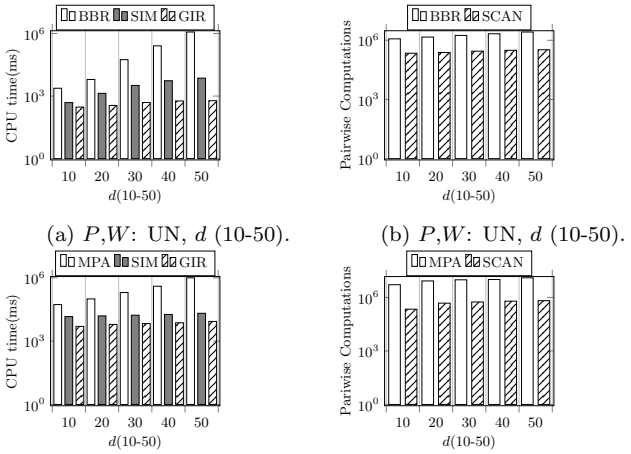
Data sets. For data set P, both real data (RE) and synthetic data sets are employed. Synthetic data sets are uniform (UN), anti-correlated (AC), and clustered (CL), with an attribute value range of $[0, 10K)$. The details on generating UN, AC, and CL data are in related research [13, 17]. To create weighting vectors W, there is additional UN and CL data that is generated in the same way. There are three real data sets, HOUSE, COLOR and DIANPING. HOUSE (Household) consists of 201,760 6-d tuples, representing the distribution percentages of an American family's annual payment on gas, electricity, water, heating, insurance and property tax. COLOR consists of 68,040 9-d tuples and describes features of images in the HSV color space. HOUSE and COLOR were also used in related works [13, 17]. DI-



(a) P, W: UN. (b) P, W: CL. (c) P: AC, W: UN.

(d) P, W: UN. (e) P, W: CL. (f) P: AC, W: UN.

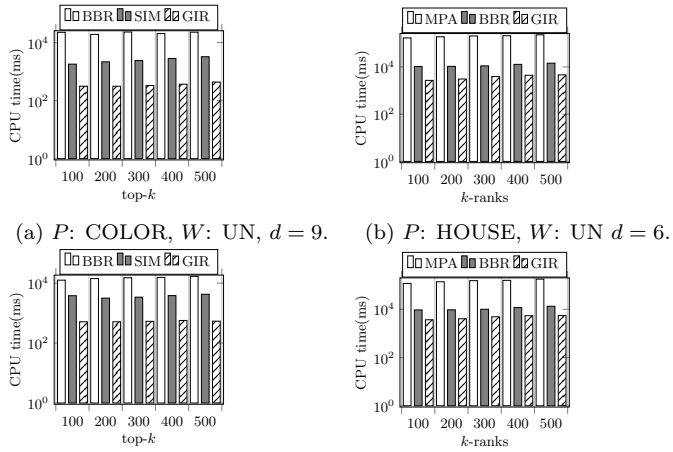
Figure 10: GIR vs BBR (a, b, c) for *RTK*, GIR vs MPA (d, e, f) for *RKR*. Performance on synthetic data with varying d (2-8), $|P| = |W| = 100$K, top-$k = 100$, k-ranks = 100, $n = 32$.



(a) P,W: UN, d (10-50). (b) P,W: UN, d (10-50).

(c) P,W: UN, d (10-50). (d) P,W: UN, d (10-50).

Figure 11: Performance on synthetic data with high dimensional d $(10-50)$, $|P| = |W| = 100$K, top-$k = 100$, k-ranks = 100, $n = 32$.



(a) P: COLOR, W: UN, $d = 9$. (b) P: HOUSE, W: UN $d = 6$.

(c) P, W: DIANPING, $d = 6$. (d) P, W: DIANPING, $d = 6$.

Figure 12: GIR vs Tree-base with RE data on varying "k", for *RTK* and *RKR* queries. $n = 32$.

ANPING is a 6-d real world data set from a famous Chinese online business-reviewing website. It includes 3,605,300 reviews by 510,071 users on 209,132 restaurants about rate, food flavor, cost, service, environment and waiting time. We use the average scores of the reviews by the same user as his/her preference (w), and the average scores of the reviews on a restaurant as its attributes (p). RRQ can be anticipated to help to find target users for these restaurants.

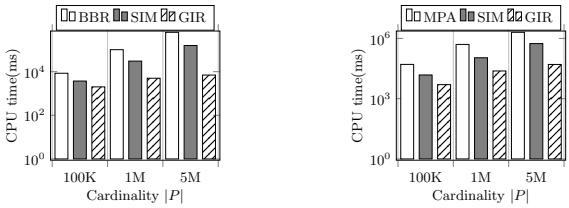
Algorithms. We implemented BBR, MPA and Simple Scan algorithms (SIM). In BBR [17], both data sets P and W are indexed by R-tree, points and weighting vectors are pruned through the branch-and-bound methodology. MPA [22] uses an R-tree to index P and a d-dimensional histogram to group W in order to avoid checking every weighting vector. In SIM, for each w, all points in P are scanned and used to compute the score. SIM also maintains a *Domin* buffer to avoid unnecessary computing and terminates when current rank does not satisfy the conditions for *RTK* or *RKR*. In conclusion, the only difference between SIM and GIR is that SIM computes a score for each p and w directly rather than using Grid-index for filtering.

Parameters. Parameters are shown in Table 5 where the default values are $d = 6$, $|P|$=100K, $|W|$=100K, k=100, the number of Grids is 32^2, and both P and W are UN data.

Metrics. We did each experiment over 1000 times, and present the average value. The query point q is randomly selected from P. Besides the query execution time required by each algorithm, we also observe the number of pairwise computations and the percentage of accessed data.

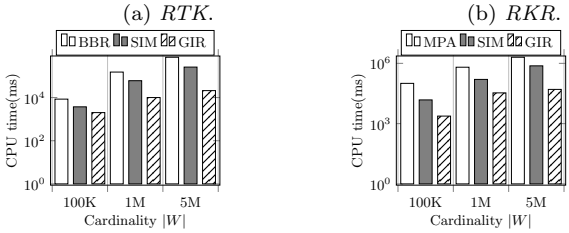
6.2 Experimental Results

Synthesis data with varying d. Figure 10 shows the performance of P (UN, AC, CL) and W (UN, CL) on synthetic data sets, with $|P|$=100K and $|W|$=100K, k=100, n = 32. Figures 10a, 10b and 10c show the CPU time and cost comparisons for *RTK* in low dimensions (2 to 8). GIR outperforms BBR in all distributions (UN,CL,AC) when data



(a) *RTK*.

(b) *RKR*.



(c) *RTK*.

(d) *RKR*.

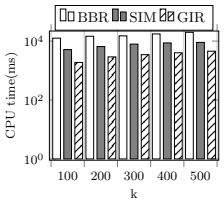
Figure 13: Scalability for all algorithms with varying $|P|$ (a,b) and $|W|$ (c,d), top-k = 100, k-ranks = 100, n = 32, d = 6.

has over 4 dimensions. SIM outperforms BBR when data has more than 6 dimensions, with the exception of CL data, since R-tree can group and prune more points when the data set is clustered. GIR always exceeds SIM at least 2 times because GIR uses score bounds from Grid-index to skip most data without doing multiplications. The results of *RKR* are shown in Figures 10d, 10e, 10f, GIR outperforms simple scan SIM at all times and outperforms the tree-based MPA with 4 to 8-dimensional data.

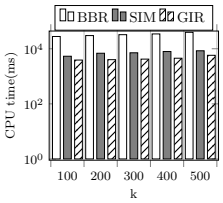
In high-dimensions (10-50), as shown in Figures 11a and 11c, the query time taken by tree-based method increases rapidly for the two reasons we presented in Sections 1.2 and 5.2: overlapping MBRs and little space to prune. Figure 11b, 11d present the number of pairwise computations for all algorithms, both BBR and MPA use more computations than the simple scan. Notice that the computation numbers for GIR and SIM are equal and are both titled "SCAN" in the figures. On the other hand, GIR is the most stable method and only grows slightly. This confirms that GIR is only slightly affected by increasing dimensionality.

Real data with varying "k". For the performance of these algorithms on real data sets (RE) with varying k. Notice that k has a different meaning, it is a query condition in *RTK* and a result size in *RKR*. Figures 12a, 12b show the results from data set HOUSE and COLOR, and data set W is generated as UN data. We process COLOR with *RTK* and HOUSE with *RKR*. Clearly, GIR is consistently superior to tree-based algorithms (BBR and MPA) and SIM, though all are stable for various k values. For the DIANPING dataset, P and W contain the average scores vectors from the reviews of users and restaurants. We peform *RTK* and *RKR* queries on DIANPING data and the Figures 12c and 12d show the comparison results. As we expected, the GIR algorithm is the most efficient for this real-world application data set.

Scalability with varying $|P|$ and $|W|$. According Figure 13, as the cardinality of data set increases P (Figures 13a and 13b) or W (Figures 13c and 13d), GIR becomes significantly superior to tree-based algorithms (BBR, MPA) and SIM. n = 32 is sufficient to filter more than 99% of points for a 6-d dataset based our Theorem 1. Thus, the

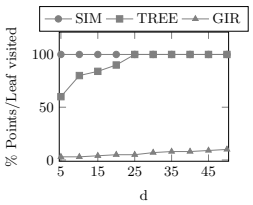


(a) *RTK*.

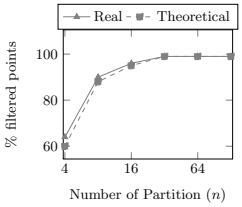


(b) *RKR*.

Figure 14: GIR vs Tree-base with varying k, for *RTK* and *RTK* queries. P, W: UN, n = 32, d = 6.



(a) Visited data, n = 32.



(b) Filtered data, d = 20.

Figure 15: (a) Visited data for all algorithms on varying d, (b) Filtering data (%) of Grid-index on varying n. $|P|$ = 100K, $|W|$ = 100K. P, W: UN.

CPU cost increased only slightly as the scale increased.

Effect on "k". Figures 12, 14 also show the performance changes when k increases from 100 to 500. All algorithms are insensitive to k because $k \ll |P|$ and $k \ll |W|$.

Accessed data points. Figure 15a shows the percentage of visited data in the leaf nodes of the R-tree and original data points on UN data. As predicted by our analysis, R-tree degenerated to a simple scan through all leaf nodes with high-dimensional data. However, GIR accesses a relatively small amount of data after filtering with Grid-index.

Effect on value range partitions n. Figure 15b shows the percent of 20-d data which can be filtered with Grid-index with various Grid numbers ($n \times n$). We created Grid-index with different n from 4 to 128 and observed the filtering of data points. The results confirm the analytical result guaranteed by Theorem 1. n = 32 is enough to guarantee a high Grid-index efficiency.

7. CONCLUSION

Reverse rank queries are useful in many applications. In marketing analysis, they can be used to help manufacturers recognize their consumer base by matching their product features with user preferences. The state-of-the-art approaches for both reverse top-k (BBR) and reverse k-ranks (MPA) are tree-based algorithms, and are not designed to deal with high-dimensional data. In this paper, we proposed the Grid-index and the GIR algorithm to overcome the cost of high-dimensional computing when processing reverse rank queries. Theoretical analysis and experimental results confirmed the efficiency of the proposed algorithm when compared to the tree-based algorithms especially in high-dimensional cases.

In future work, there are two extensions for GIR algorithm. The first is to find a heuristic method to adapt GIR to different data distributions by using non-equal-width Grid-index. This is easy to implement by merging and splitting

some grids of the equal-width Grid-index based on the distributions of the given P and W. The challenging point is the model of filtering performance with varied distributions in different dimensions. The second extension is to do optimization when the user preferences data $w \in W$ has many zero entry, i.e., when W is sparse. Since in practice, a user is normally interested in a few attributes of the products.

Acknowledgement

This work was partly supported by JSPS KAKENHI Grant-in-Aid for Scientific Research (B) (Grant number: 26280037).

8. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 322–331, 1990.

[2] S. Berchtold, D. A. Keim, and H. Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39, 1996.

[3] Y. Chang, L. D. Bergman, V. Castelli, C. Li, M. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 391–402, 2000.

[4] H. Chen, J. Liu, K. Furuse, J. X. Yu, and N. Ohbo. Indexing expensive functions for efficient multi-dimensional similarity search. *Knowl. Inf. Syst.*, 27(2):165–192, 2011.

[5] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top-k queries in two dimensions. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part I*, pages 201–208, 2013.

[6] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 291–302, 2007.

[7] Y. Dong, H. Chen, K. Furuse, and H. Kitagawa. Aggregate reverse rank queries. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, pages 87–101, 2016.

[8] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 259–270, 2001.

[9] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 500–509, 1994.

[10] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 201–212, 2000.

[11] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 213–226, 2008.

[12] J. V. Uspensky. *Introduction to Mathematical Probability.* New York: McGraw-Hill, 1937.

[13] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 365–376, 2010.

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

[15] A. Vlachou, C. Doulkeridis, and K. Nørvåg. Monitoring reverse top-k queries over mobile devices. In *Proceedings of the Tenth ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE 2011, Athens, Greece, June 12, 2011*, pages 17–24, 2011.

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

[17] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 481–492, 2013.

[18] S. Wang, M. A. Cheema, X. Lin, Y. Zhang, and D. Liu. Efficiently computing reverse k furthest neighbors. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1110–1121, 2016.

[19] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205, 1998.

[20] S. Yang, M. A. Cheema, X. Lin, and Y. Zhang. SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 760–771, 2014.

[21] B. Yao, F. Li, and P. Kumar. Reverse furthest neighbors in spatial databases. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 664–675, 2009.

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