A Study on Rank-aware Query Processing for Multidimensional Data

March 2019

DONG YUYANG

A Study on Rank-aware Query Processing for Multidimensional Data

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2019

DONG YUYANG

Acknowledgments

Firstly, I would like to express my sincerest gratitude to my supervisor, Professor Hiroyuki Kitagawa, who has supported me throughout my thesis with his patience, motivation, enthusiasm, and immense knowledge. I would also like owe my deepest gratitude to my co-supervisor, Lecturer Hanxiong Chen, for making this research possible. His excellent guidance, caring, patience provide me an excellent environment for doing research.

Besides my supervisors, I would like to thank all teachers and students in DSE Lab and KDE Lab. Not only for their insightful comments and encouragement on my research, but also the helping in my ordinary life of study in Japan.

I would like to thank the following committee members of this doctoral dissertation: Professor Hiroyuki Kitagawa, Professor Toshiyuki Amagasa, Professor Kazuo Misue, Professor Tetsuji Satoh, Professor Mikio Yamamoto, and Lecturer Hanxiong Chen, for their insightful comments and encouragement. Their questions and comments greatly encouraged me to widen my research from various perspectives, and most importantly improve the quality of this dissertation

I would like to thank the Chronology of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) and Japan Student Services Organization (JASSO) for the scholarship support.

Finally, I would like to thank my parents for their unconditional support. I would also like to thank my dear wife, Li Xiang. She is a Ph.D. student of linguistics. She is my primary motivation that finishing my Ph.D. degree.

Abstract

Recently, information and communication technologies such as the 5G network are developing at high speed. Due to the increasing popularity of Internet of Things, SNS service and other web services, we witness exponential growth in the amounts of data on the web, and the growth will be more explosive in the future.

Query processing is considered to be one of the fundamental and indispensable processes in retrieving target data from such a big data source. In query processing, ranking is an important technique which shows the relative significance of candidate results. Rank-aware query processing evaluates source data with some similarity measure and produces ranking lists. It can be applied in many fields such as data mining, information retrieval, pattern recognition, and machine learning.

We study the rank-aware query processing on multidimensional data in two type of data: static marketing data and dynamic spatial keyword data. Three different query problems: (a) *Aggregate Reverse Rank Query*; (b) *Weighted Aggregate Reverse Rank Query* and (c) *Continuous Search on Dynamic Spatial Keyword Objects*, are formulated and efficient query processing methods are proposed in this dissertation.

For the static marketing data, we propose (a) *Aggregate Reverse Rank Query* that returns k users who favor the given set of products more than other users. Three different aggregate rank functions (SUM, MIN, MAX) are defined to target potential users in three normal views. To solve Aggregate Reverse Rank Query efficiently, we devise a novel bound-and-filter framework to deal with low-dimensional data. We also propose a grid index method for high-dimensional data. To generalize Aggregate Reverse Rank Query, we define (b) *Weighted Aggregate Reverse Rank Query*, which extends Aggregate Reverse Rank Query by adding weights for different inputted query products. With the help of weights, Weighted Aggregate Reverse Rank Query can handle any situations of users having different preferences to the inputted query products. We adapts the bound-and-filter solution to the Weighted Aggregate Reverse Rank Query. We also study a bounding approach to optimize the filtering space in the bound-and-filter framework.

For dynamic spatial keyword data we propose a problem named (c) *Continuous Search on Dynamic Spatial Keyword Objects*. We design a novel grid-based index to manage both dynamic objects and static queries. We also propose a novel strategy that refills one candidate object rather than reevaluating the entire top-k list. To take advantage of the cells in the grid-based index, we design a buffer named PCL (partial cell list). PCL balances the trade-off between search process and buffer maintenance to optimize efficiency.

For the above three query problems, we conduct sufficient experiments on both real-world data and synthetic data to test the performance compared to the baseline methods and the state-of-the-art methods. We also give theoretical analysis to confirm the superiorities of our proposed methods.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we first provide background on rank-aware query processing with multidimensional data. We then move on to describe the motivations and contributions of this dissertation. Finally, we outline the organization of this dissertation.

1.1 Background

We live in the era of big data, and we can all easily see how big data processing techniques are changing our daily lives. Recently, information and communication technologies such as the 5G network are developing at high speed. Due to the increasing popularity of Internet of Things, SNS service and other web services, we witness exponential growth in the amounts of data on the web, and the growth will be more explosive in the future. For example, there were over 2.27 billion monthly active users on Facebook in 2017, creating 2.5 trillion posts [1]; Amazon had over 300 million users and had made over 178 billion sales in 2017 [2].

Making use of this big data evolves our daily lives. Now we can access limitless information anytime and everywhere via the internet from the source of big data. We can type our question on Google and get answers from the big data source in less than 1 second. Web services can also take advantage of the big data to enrich their business. According to people's preferences and behaviors on the web, web services can also quickly recommend related products, videos, restaurants, friends, and so on. All these convenient things are due to the development of rank-aware query processing technologies.

Rank-aware query processing (e.g., [45, 46, 48]) is considered to be one of the fundamental and indispensable processes in retrieving target data from such a big data source. It can be applied in many fields such as data mining, information retrieval, pattern recognition, and machine learning. In

[1] https://www.wordstream.com/blog/ws/2017/11/07/facebook-statistics

[2] https://expandedramblings.com/index.php/amazon-statistics/

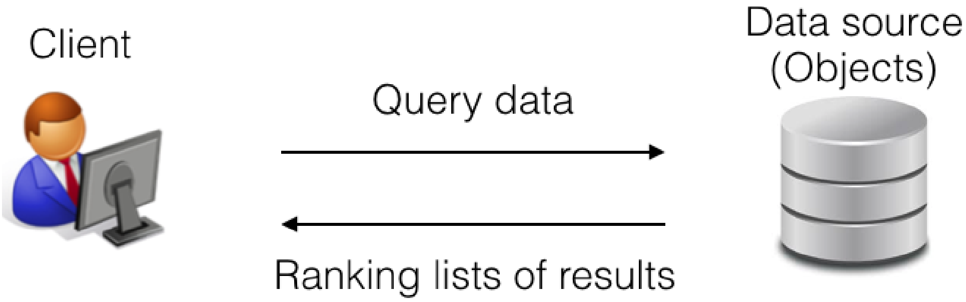


Figure 1.1: Rank-aware query processing.

rank-aware query processing, the real-world information always modeled as multidimensional data to react to the variety of attributes. For example, a product is represented as multidimensional data with its price, rating, size and so on; a tweet post is represented as multidimensional data with its context, date, geo-tag and so on.

Figure 1.1 shows the concept of rank-aware query processing. A client issues a rank-aware query by giving query data which describes what kind of data is needs. Then, it is required to compute similarities among source data (or we usually say objects) and the query data (Different similarity measure is introduced in Section 2.2). Finally, the ranking lists of candidate results are produced based on similarities and are returned to the client. It is worthy to note that ranking is an important technique which shows the relative significance of the candidate results. Therefore, we can easily select the candidate data with the priority of a ranking list. Usually, an integer value k is specified as the size of the result, in other words, the number of data to retrieve. For example, a k Nearest Neighbor query (e.g., [76]) is a basic rank-aware query processing problem that retrieves k closest data objects to the specific query data.

1.2 Motivations and contirbutions

Depending on the type of objects, the research of rank-aware query processing can divide into two parts: retrieving static data and retrieving dynamic data. In this dissertation, we track the challenges on both static data and dynamic data with three rank-aware query processing problems.

For static data, we focus on studying rank-aware query processing on marketing data. The marketing data always modeled as a static dataset since the attributes of products cannot be easily changed. We study the *Aggregate Reverse Rank Query* and *Weighted Aggregate Reverse Rank Query* on processing static marketing data. Regarding dynamic data, we focus on studying rank-aware query processing on GIS data. The moving objects in GIS data always update their attributes (i.e., location) in real-time. We study the *Continuous search on Dynamic Spatial Keyword Objects* on processing dynamic GIS data.

1.2.1 Aggregate Reverse Rank Query

For marketing data, many research papers [25, 47, 68, 82, 85, 86, 106] formalized a user product model (details in Section 2.1) to represent the products and users' preferences in marketing. Finding products based on a user is a basic application of the search engine or recommendation system. This is a user-view problem that was formalized and named as top-k preference query [9, 25, 43, 47, 82]. In contrast to the above top-k preference query, many applications such as marketing analysis or advertisement systems require a problem that sits at the manufacturer-view level, for example, targeting users for a newly released product. [68, 85, 86, 106] studied this problem and named it reverse ranking query.

However, in the manufacturer-view, besides targeting users for a single product, there is also a huge demand for targeting users for a set of products. Product bundling (or package selling) is an important marketing strategy that bundles multiple products and selling them together. For example, the cable television industry often bundles various channels into a single tier to expand the channel market.

Unfortunately, the existing research only focuses on targeting users for a single product and their approaches cannot be used for product bundling cases. For this reason, we propose a problem named aggregate reverse rank query which finds users for product bundling in Chapter 4. Three different aggregate rank functions (SUM, MIN, MAX) are defined to target potential users. To solve aggregate reverse rank query efficiently, we devise a novel bound-and-filter framework to with low-dimensional data. We also propose a grid index method for high-dimensional data. In the bound-and-filter framework, queries are bounded to calculate an approximate aggregate rank value efficiently, then we use tree-based structure to filter data in processing. With high-dimensional data, we propose a grid index method which uses pre-calculated score bounds to reduce multiplications in the simple scan.

1.2.2 Weighted Aggregate Reverse Rank Query

Aggregate Reverse Rank query is an essential tool for finding potential customers for a given product bundle. However, it has a limitation that it can only deal with a part of scenario in which ranks are evaluated in the SUM, MIN and MAX aggregate functions. Therefore, it is more accurate to add weights for different products in the bundle and evaluate them with a weighted function.

To generalize the query problem of the aggregate reverse rank query, we define a weighted aggregate rank query that extends the previous aggregate rank query by adding weights to different inputted query points. With the help of weights, weighted aggregate reverse rank query can handle situations where users have different preferences to the inputted query products.

We develop three solutions called SFM, EFM and OBM. SFM is a straightforward method that uses a spatial R-tree. EFM adapts the bound-and-filter framework to the additional weights. We also study the filtering space in the bound-and-filter framework and propose an optimal bounding approach that is proven to find the tightest bound of Q. The OBM is a solution based on this optimal bound.

1.2.3 Continuous Search on Dynamic Spatial Keyword Objects

For GIS data, a spatial keyword model (details in Section 2.1) is formalized for the data that has both location attribute and textual attribute such as geo-tag tweets. Many research on continuous spatial keyword queries [13, 14, 17, 53, 57, 60, 90, 91] has been studied for continuous searching users for a business.

Although different types of monitor systems have been studied, they only consider the cases of a user with a dynamic location attribute and a static keyword attribute. However, it is more realistic to set a dynamic spatial keyword data like a person changes the content on his/her cellphone (keyword attribute) while moving around (location attribute). It is also a huge demand for monitoring systems to response to the dynamic interested keywords from users.

Based on this, we define a novel query process problem, which continuously searches for the top-k dynamic spatial keyword objects in Chapter 6. We also propose a solution system for efficient processing the problem of dynamic spatial keyword search. In this system, to overcome the challenge of indexing, we design a novel grid-based index to manage both dynamic objects and static queries. The grid-based index can support rapid and economical updates of dynamic objects. In addition, queries are indexed with a sophisticated strategy of influential circles. Queries affected by a dynamic object can be quickly identified. For the second challenge of the top-k reevaluation, we propose a novel strategy that refills one candidate object rather than reevaluating the entire top-k list. To take advantage of the cells in the grid-based index, we design a buffer named PCL (partial cell list). PCL balances the trade-off between search process and buffer maintenance to optimize efficiency.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 outlines the basic concepts and definitions of rank-aware query processing on multidimensional data, as well as data modeling and data indexing. Chapter 3 describes the related works and highlights the position of the works in this dissertation. Chapter 4 introduces the proposed aggregate reverse rank query and efficient solutions on both low dimensional data and high dimensional data. Chapter 5 proposes a weighted aggregate reverse rank query and offers solutions. Chapter 6 proposes a continuous spatial keyword search on dynamic objects, as well as an efficient solution system. Chapter 7 gives a summary for this dissertation and remarks the future works.

Chapter 2

Preliminaries

In this chapter, we first introduce a formal definition of multidimensional data, and two data models studied in this dissertation. Then, we introduce different similarity functions, as well as the different types of query processing. We also introduce some popular indexing structures used in this dissertation.

2.1 Data models

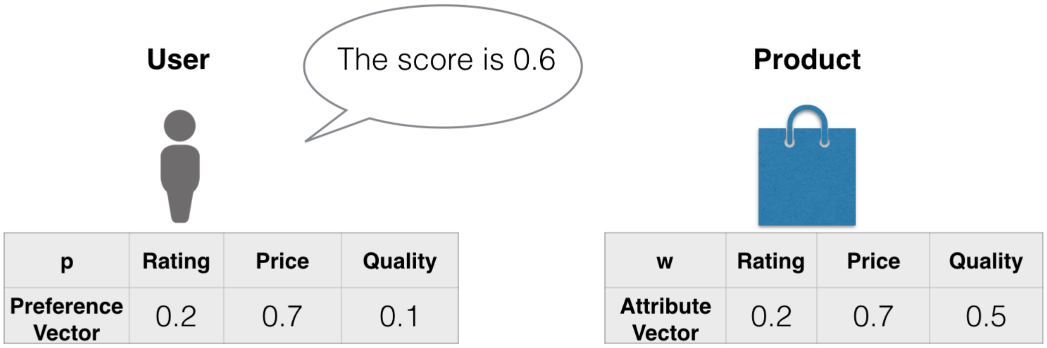
2.1.1 Multidimensional data

In general, any data (real-world information) with multiple numerical attributes can be described as a multidimensional vector that can be viewed as a point or object in a multidimensional space. In this dissertation, we assume that a d-dimensional data p is composed of d nonnegative numerical attributes, i.e., $p = (p[1], p[2], p[3], ..., p[d])$. Here, d is an integer value that represents the dimensionality. $p[i]$ refers to the attribute value of p in the ith dimension, where $i \in \{1, 2, 3, ..., d\}$.

2.1.2 User product model

Many research papers [9, 25, 43, 47, 68, 82, 85, 86, 106] formalized the user product model to represent the products and users' preferences in marketing. In addition to E-commerce, the concept of the user product model can also be applied to a wide range of applications such as dating and job-hunting services.

There are two types of data sets P and W in the user product model. P represents a set of products, and W represents a set of users. Both P and W are in a d-dimensional space. Each product in the product dataset $p \in P$ is a d-dimensional vector that contains d nonnegative values. p is represented as: $p = (p[1],\ p[2],\ ...\ ,\ p[d])$ where $p[i]$ is the attribute value of p in the ith dimension. The preference $w \in W$ is also a d-dimensional weighting vector, and $w[i]$ is a nonnegative



p	Rating	Price	Quality
Preference Vector	0.2	0.7	0.1

w	Rating	Price	Quality
Attribute Vector	0.2	0.7	0.5

Figure 2.1: An example of user product model.

weight that evaluates the ith attribute of products, where $\sum_{i=1}^{d} w[i] = 1$. Many research papers carry out a weighted sum function to calculate the score of a product p to a preference w by $f(w,p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. Figure 2.2 shows an example of the user product model.

2.1.3 Spatial keyword model

The spatial keyword model is formalized for the data that has both geo-tag and textual contents. GPS devices (e.g. smartphone) generate huge spatial keyword data such as geo-tag tweets, blogs, and reviews. This kind of data is named as spatial keyword data. Many location-based services require querying the spatial keyword data.

Let O be a set of spatial keyword objects (users) and Q be a set of business or POIs (query points). Each spatial keyword object $o \in O$ is defined as $O = (o.\rho, o.\psi, o.t)$. $o.\rho$ is the location of a user with a 2-d coordinates (x, y) in geography space. $o.\psi$ is a set of keywords and $o.t$ represents a timestamp. Similarly, each query point $q \in Q$ is defined as a $q = (q.\rho, q.\psi, q.\eta)$. where $q.\rho$ is the location, $q.\psi$ is a set of keywords, and $q.\eta$ is the query condition.

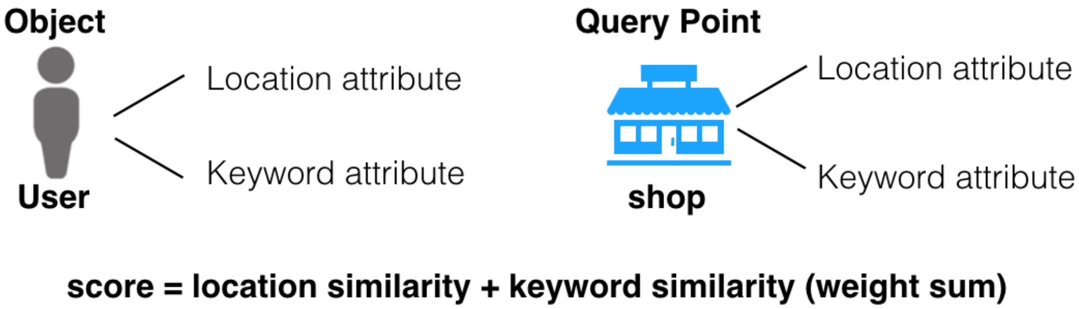


Figure 2.2: An example of spatial keyword model.

2.2 Similarities

A rank-aware query processing requires to calculate the similarity between query data and data source. Assume that there are two multidimensional data represented as vectors $p = (p[1], p[2], p[3], ..., p[d])$ and $q = (q[1], q[2], q[3], ..., q[d])$, we introduce how to calculate the similarity between them with different similarity measures.

2.2.1 Euclidean distance

Euclidean distance (or Euclidean metric) is the straight-line distance between two points. In Euclidean d-space (d is the dimensionality), the distance from p to q, denoted as $dist(p, q)$ is:

$$dist(p, q) = \sqrt{\sum_{i=1}^{d} (q[i] - p[i])^2} \tag{2.1}$$

For the similarity view, we need to convert the distance to $1 - dist(p, q)$ to accord with the intuition of similarity that the larger value means more similar.

2.2.2 Cosine similarity

The cosine similarity between two vectors (or two documents) is a measure calculating the cosine of the angle between them. The vectors usually represent the word count (tf-idf) of each document. In a d-dimensional vector space, the cosine similarity between p and q, denoted as $cos(p, q)$ is:

$$cos(p, q) = \frac{p \cdot q}{||p||||q||} = \frac{\sum_{i=1}^{d} p[i]q[i]}{\sqrt{\sum_{i=1}^{d} p[i]^2}\sqrt{\sum_{i=1}^{d} q[i]^2}} \tag{2.2}$$

2.2.3 Jaccard similarity

Jaccard similarity is a term coined by Paul Jaccard, measures similarities between two sets. It is defined as the size of the intersection divided by the size of the union of two sets. For two sets P and Q, the Jaccard similarity $jac(P, Q)$ is:

$$jac(P, Q) = \frac{|P \cap Q|}{|P \cup Q|} = \frac{|P \cap Q|}{|P| + |Q| - |P \cap Q|} \tag{2.3}$$

2.2.4 Dot product (inner product, weighted sum function)

For two vectors p and q, the dot product is equal to the length of the projection from one to the other. A simple weighted function is equal to the dot product (inner product) of two vectors, where one of the vectors is the data we want to evaluate, and the other is the weights.

$$f(p, q) = p \cdot q = \sum_{i=1}^{d} p[i]q[i] \tag{2.4}$$

2.2.5 Combined similarity

In some cases, it may require to combine two different similarities. For the query processing in the spatial keyword model, we need to retrieve the similar data on spatial similarity and keyword similarity. Many research [14, 90, 95] propose a weighted sum function $SimST(.)$ to combine these two different similarity with a smoothing parameter α:

$$SimST(p, q) = \alpha \cdot SimS(p, q) + (1 - \alpha) \cdot SimT(p, q) \tag{2.5}$$

$SimS(.)$ denotes the spatial similarity and $SimT(.)$ represents keyword similarity. Depends on the problem settings and applications, we use different similarities to combine.

2.3 Type of queries

We classify the query types as ranking query and reverse ranking query.

2.3.1 Ranking query

The type of ranking query is a kind of query processing that retrieves objects with respect to the similarity. Specifically, inputting a query point and getting a list of objects ranked by the similarities between query point and each object. For example, kNN query [19, 42, 76] is a ranking query which inputs a single query point and finds the k nearest objects ranking by distance. In the same way, the top-k queries with different similarity functions [9, 25, 43, 47, 82] also belong to ranking query. Some variants require to input multiple query points then find objects with an aggregate similarity, such as aggregate nearest neighbor query [71], group nearest neighbor query [70] and group nearest group query [27]. They are also ranking query since they also rank objects with a kind of similarity. To answer a ranking query, we only need to execute a one-time kNN like processing. Therefore, $o(N)$ is the worst computational complexity for solving a ranking query where N is the cardinality of objects.

2.3.2 Reverse ranking query

The type of reverse ranking query is a reverse version of the ranking query that retrieves objects with respect to their ranking query results. Specifically, inputting a query point and getting a list of objects, each of the object contains the inputted query point in their ranking query results. For example, reverse kNN query [11, 78, 80, 81, 99, 100] is a kind of reverse ranking query which retrieves the objects for a query point, and the objects treat this query point as their kNN result. Some reverse ranking queries focus on ranking value instead of similarity value, they find objects and rank them with respect to the ranking relationship between query point and objects. For example, the reverse k-rank query [106] returns a list of k objects, where each object in this list ranks the inputted query point higher than the objects which are not in this list, and the list is ranked by the ranking value between query point and objects. To answer a reverse ranking query, we need to execute multiple times of kNN like processing. Therefore, $o(N^2)$ is the worst computational complexity for solving a ranking query since we need to execute a ranking query for every object, and N is the cardinality of objects.

2.4 Data indexing

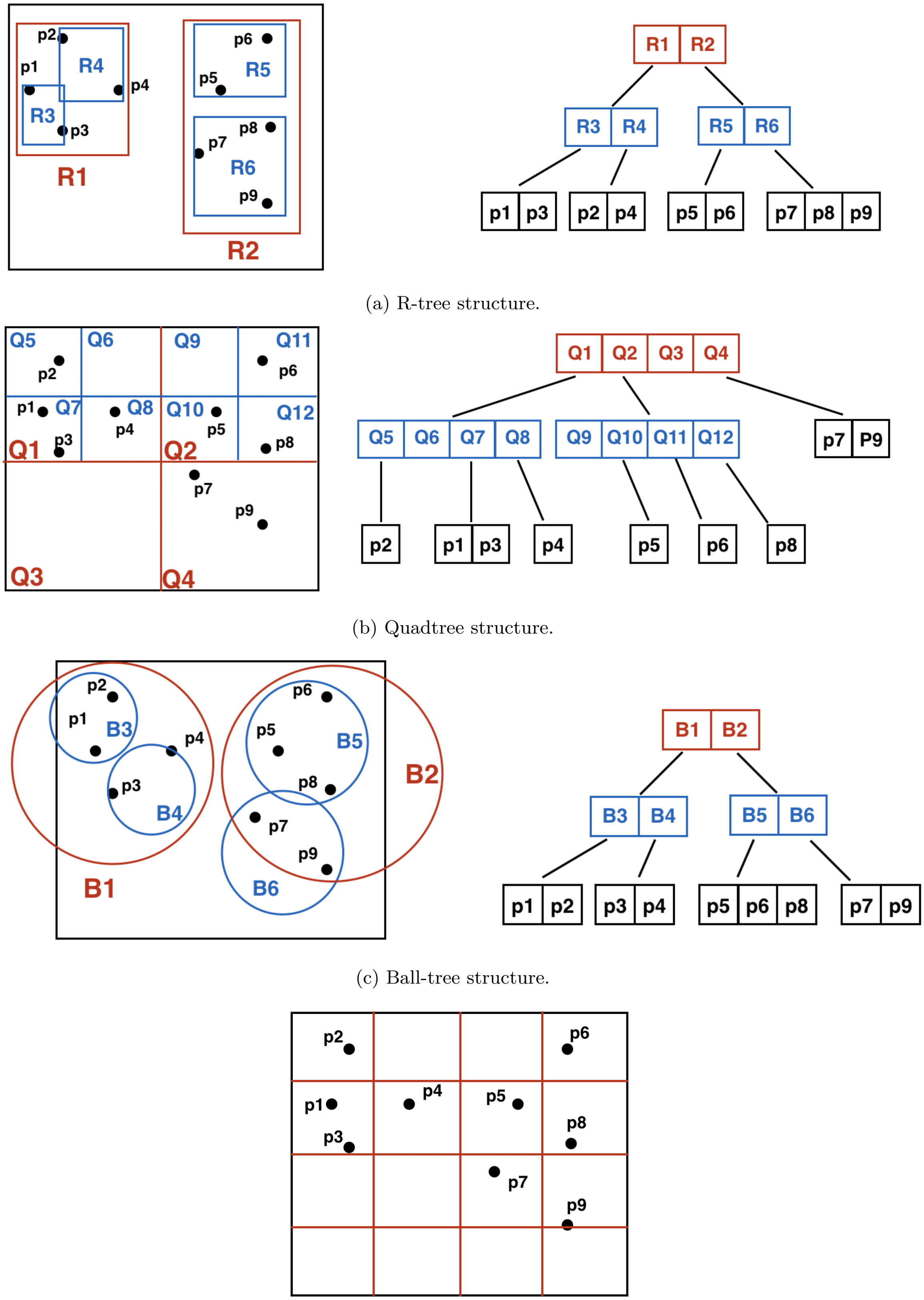
In this section, we introduce some index structures for multidimensional data. We first introduce several spatial indices which manage multidimensional data with their spatial attributes. We also introduce the textual indices and hybrid indices for the application of a spatial data binding with a non-spatial attribute (e.g. a set of keywords).

2.4.1 Spatial indices

To manage the multidimensional data, we always use the spatial indices which can group similar data with the feature of spatial distance. Here we introduce several spatial indices which are used in this dissertation or other related works.

R-tree. R-tree [41] was proposed by Antonin Guttman in 1984 and has found significant use in both theoretical and applied contexts. R-tree is a balanced search tree, it groups nearby objects and represents them with their minimum bounding rectangle (MBR). Specifically, each internal node in R-tree represents a group of objects with an MBR, and its children are smaller MBRs representing the subsets of these objects. The data of the objects are indexed in the leaf node on the lowest level of R-tree. Figure 2.3a shows an example of the R-tree structure in a 2d space. Besides the conventional R-tree, many variants were also proposed such as R*-tree [2] and R$^+$-tree [77] with different strategies on avoiding overlapping nodes.

Quad-tree. A Quad-tree [38] is a 2-dimensional space partition structure in which each internal node has exactly four children. Unlike the R-tree, Quad-tree is a non-balanced tree structure and data can be indexed at any level. Normally, the nodes of Quad-tree do not change when Inserting or deleting an object, unless deciding to delete a whole node or split it with four children nodes. Therefore, comparing to the data-partition indices (e.g. r-tree) which may incur an expensive structure updating when updating objects, Quad-tree is insensitive to the dynamic objects so it is a proper way to manage them. Figure 2.3b shows an example of a Quad-tree structure in a 2d space.



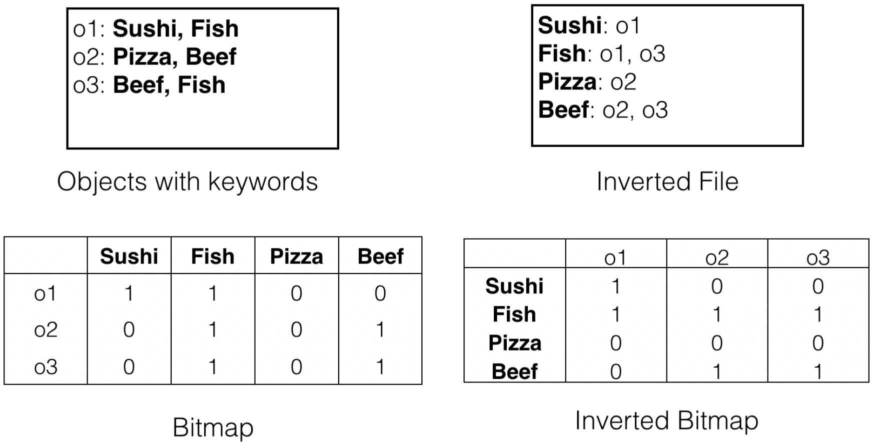
(a) R-tree structure.

(b) Quadtree structure.

(c) Ball-tree structure.

(d) Grid structure.

Figure 2.3: An example of spatial indices with points in 2-dimensional data space.



o1: **Sushi, Fish**
o2: **Pizza, Beef**
o3: **Beef, Fish**

Objects with keywords

Sushi: o1
Fish: o1, o3
Pizza: o2
Beef: o2, o3

Inverted File

	Sushi	**Fish**	**Pizza**	**Beef**
o1	1	1	0	0
o2	0	1	0	1
o3	0	1	0	1

Bitmap

	o1	o2	o3
Sushi	1	0	0
Fish	1	1	1
Pizza	0	0	0
Beef	0	1	1

Inverted Bitmap

Figure 2.4: An example of inverted file and bitmap.

Ball-tree. A Ball-tree [69] is a binary tree in which every node is a d-dimensional hypersphere (or ball) containing a subset of the objects from a higher level. When splitting a ball, each object is associated with a closer ball. Figure 2.3c shows an example of a ball-tree structure in a 2d space.

Grid. A Grid is a simple structure that divides a space into equal size cells. Objects are indexed into a covered cell. Because grid structure contains equal width cells with only one level, the complexity of insert and deletes data is $o(1)$. Moreover, a grid structure has a fixed size (i.e., the number of cells in a grid) after the construction, so there is no structure update. Therefore, grid structure is fit for the applications of indexing dynamic objects (moving objects). On the other hand, since we can not change the number of cells in a grid, we need to define the size carefully with a theoretical underpinning. Figure 2.3d shows an example of a grid structure in a 2d space.

Other indices. There are many other spatial indices. k-d tree [5] is a binary tree in which every leaf node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two part. X-tree [6] 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. M-tree [22] is similar a to the R-tree. It is constructed using a metric and relies on the triangle inequality for efficient range and rank-aware queries. Space-filling curves are studied to reduce the dimensionality of the space and manage data with a sequence, the most representative curves are Hilbert curve [7] and Z-order curve [64].

2.4.2 Textual indices

When indexing objects with respect to their non-spatial attributes (mainly keyword attributes), we usually take advantage of following structures to manage objects. Figure 2.4 shows the example of textual indices.

Inverted File. For a specific keyword, an inverted list is a sequence of objects which contain this keyword. As the example shown in Figure 2.4, the inverted list of the keyword “fish” contains o_1 and o_3. An inverted file is a set containing all inverted lists for a set of keywords. The purpose of an inverted file is to allow fast keyword matching search.

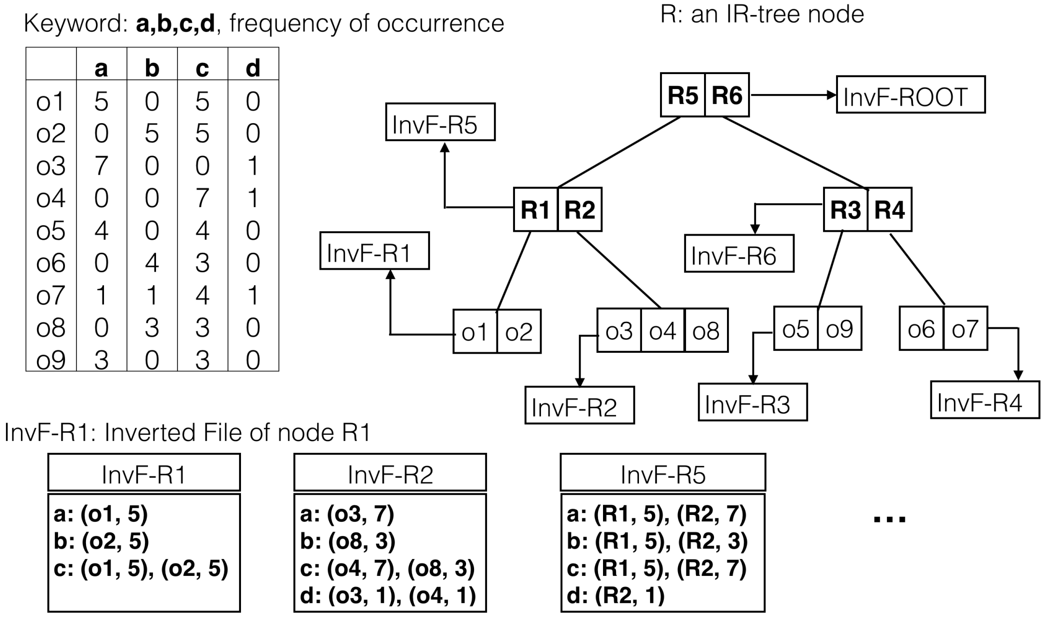
Bitmap. Like the example in Figure 2.4, a bitmap is a table that collects the presence and absence of each keyword for all objects. For some application, an inverted bitmap is also used.

2.4.3 Hybrid indices

For the spatial keyword model we mentioned before, it is inefficient if we search one attribute first then (search) the other. In other words, it is inappropriate to use two independent spatial index and textual index. Many research propose hybrid indices that combined a spatial index and a textual index for quick searching.

We introduce a representative IR-tree to give an image of hybrid spatial keyword index. The IR-tree [24,93] links each node of the R-tree with a pointer to an inverted file that describes keyword information of the objects’ current node. Figure 2.5 shows an example of IR-tree. Note that the inverted file also contains the frequency of the keywords, which is widely used in the frequency-aware applications. When traversing an IR-tree, we can check the spatial attribute and keyword attribute simultaneously for the objects indexed in a node.

Similar to the IR-tree structure, there are many others hybrid indices that tightly combined a spatial part and a keyword part. The IR^2-tree [37] integrates a bitmap into each R-tree node. [14,90] proposed structures that combine each Quad-tree node with an inverted file. [16] combines an inverted file with a Hilbert curve. [21] optimizes the structure of the Hilbert curve with an inverted file. The SFC-QUAD index [98] intergrades inverted file with Z-curve. [50] uses an inverted file to manage sequences of cells from a grid structure, it can handle spatial attribute and keyword attribute simultaneously.



Keyword: **a,b,c,d**, frequency of occurrence

	a	**b**	**c**	**d**
o1	5	0	5	0
o2	0	5	5	0
o3	7	0	0	1
o4	0	0	7	1
o5	4	0	4	0
o6	0	4	3	0
o7	1	1	4	1
o8	0	3	3	0
o9	3	0	3	0

R: an IR-tree node

InvF-R1: Inverted File of node R1

InvF-R1

a: (o1, 5)
b: (o2, 5)
c: (o1, 5), (o2, 5)

InvF-R2

a: (o3, 7)
b: (o8, 3)
c: (o4, 7), (o8, 3)
d: (o3, 1), (o4, 1)

InvF-R5

a: (R1, 5), (R2, 7)
b: (R1, 5), (R2, 3)
c: (R1, 5), (R2, 7)
d: (R2, 1)

Figure 2.5: IR-tree: a hybrid structure of R-tree and inverted file.

Chapter 3

Related Works

In this chapter, we survey the related works on rank-aware query processing on multidimensional data. We divide the related works with two groups: static data and dynamic data. In each group, we partition related works with ranking queries and reverse ranking queries. Table 3.1 is an overview that compares the works in this dissertation to the existing works with different taxonomies.

Type of Data	**Type of Queries**	
	Ranking Query	**Reverse Ranking Query**
Static Data	Preference top-k [9,25,43,47,82] kNN [19,42,76] Group kNN [27,70,71] Top-k spatial keyword [8,15,23,58]	Reverse kNN [51,56,99] Reverse top-k [85–87] Reverse k-rank [68,106] Aggregate reverse. Chpt.4 Weighted aggregate. Chpt.5
Dynamic Data	Continuous kNN [66,67,97,105] Moving Top-k spatial keyword [40,94,95] Publish/Subscribe [14,17,57,90] Dynamic spatial keyword. Chpt.6	Reverse NN for moving objects [4] Monitor reverse top-k [104] Moving reverse top-k [88]

Table 3.1: Related works and the position of the works in this dissertation.

3.1 Rank-aware query processing on static data

3.1.1 Ranking queries

kNN queries

Given a query point q and a point dataset P, kNN query returns k nearest points to q from the dataset P. kNN query is a fundamental problem in many fields such as database management system, data mining and information retrieval. Many research were studied to solve kNN query efficiently, one popular solution is the R-tree based processing [19, 42, 76]. R-tree [41] is a spatial index which groups close points in space with the minimum bounding rectangle (MBR), and manages MBRs with a tree structure. Nick et al. [76] first proposed an R-tree based method for processing the (k)NN query with geographic information systems (GISs). Hjaltason et al. [42] proposed a best-first method with R-tree to solve (k)NN query.

Group kNN queries

As real-life applications are not satisfied with only evaluating the relationship between single points like conventional kNN query. Various kinds of variants of group version NN query have been studied. The difference point is to input a group of query points (or multiple query points) instead of a single point in classic kNN query. Tao et al. developed aggregate nearest neighbor query (ANN) [70, 71] to retrieve points with the smallest aggregate distance to multiple query points in metric space. ANN inputs a group of query points Q and aims to find a nearest single point which has the smallest aggregate distance to Q. Yiu et al. [103] process ANN query in road networks with the graph structures. Deng et al. [27] proposed a *group nearest group query* (GNG), which is a spatial query returning a group of points. Choi et al. [20] indicated that the conventional kNN query may not give the best answer especially when the resulting set of k neighbors need to be clustered. They proposed a group version of kNN query named *Nearest Neighborhood Query* (NNH), it can return a nearest cluster for a given query point q.

Top-k preference queries

Top-k query (ranking query) has been studied extensively for decades and there are plenty of variants of top-k queries. Many works, such as [9, 25, 43, 47, 82], concern preference top-k queries with a linear combination similarity function. Many applications are designed for returning a limited set of ranked products on individual user preferences, the most basic of which is the top-k query. Here, we summarize some important work in ranking queries. Chang et al. proposed the Onion technique to pre-process and index data points in layers with convex hulls [9] for linear optimization queries; the onion-based index can help to filter data and compute efficiently. A tree-based index approach, which processes the top-k queries with a branch-and-bound methodology, has been studied in [79]. Fagin's algorithm [35] and the threshold algorithm (TA) [36] were proposed to compute top-k queries over multiple sources, where each source has only a subset of attributes. Other variants of the threshold-based algorithms for top-k queries were investigated in [1, 10, 61]. Ihab F. Ilyas et al. [47] gave an important study that describes and classifies top-k query processing techniques in relational databases. Mouratidis et al. [65] gave a summary of the geometric approaches for (reverse) top-k queries.

Top-k spatial keyword queries

Research on searching geo-textual objects with query locations and keywords are widely studied. Various works have retrieved spatial keyword objects through different types of queries such as boolean matching [24] or a combined score function evaluation [75]. The survey paper [15] and tutorials [23,58] give sufficient summaries of different problem settings and techniques in the spatial keyword search.

Other spatial keyword queries

There are also many other types of spatial keyword queries for different applications. Cao et al. [8] propose the spatial group keyword queries that find a group of objects cover all keywords and has a close aggregate distance. Lu et al. [56] propose the reverse spatial and textual k nearest neighbor query which retrieves the objects that have the query object as one of their k most similar objects with regards to both spatial and keyword similarities. Li et al. [52] propose the direction-aware spatial keyword query. Given a spatial point, a set of keywords and a direction vector. Direction-aware spatial keyword query finds k closed objects in the query direction and contains all keywords.

3.1.2 Reverse ranking queries

Reverse kNN query

Given a query point q and a point dataset P, reverse kNN query retrieves the points from P that treat q as the kNN. Korn et al. [51] proposed the reverse nearest neighbor (RNN) query. For reverse k-nearest neighbor (RKNN) queries, Yang et al. [99] carried out an in-depth investigation that analyzed and compared notable algorithms in [11,78,80,81,100]. Besides nearest neighbor, Yao et al. [101] proposed the reverse furthest neighbor (RFN) queries to find points in which the query point is deemed to be the furthest neighbor. Wang et al. [89] extended the RFN to RkFN queries for an arbitrary value of k and proposed an efficient filter in the search space. Reverse skyline query returns a user based on the dominance of competitors' products [26,54]. The preference of a user is described as an ideal product point in this query. However, preferences are represented as weighting vectors in reverse rank query.

Reverse top-k queries

In marketing analysis, such as identifying competing products or targeting potential customers, many variants of rank-aware queries have been widely researched [28–33,85–87,106]. The converse of rank queries, called reverse rank queries, have been studied extensively. Reverse rank queries evaluate the rank of a query product based on user preferences and retrieve the top-k users. One of these, reverse top-k query, has been proposed in order to find users who treat a query product as their top-k product [85,86]. For an efficient reverse top-k process, Vlachou et al. [87] proposed a branch-and-bound algorithm using a tree-based method with boundary-based registration. [18] proposed an efficient method to answer two dimensional reverse top-k queries. Gao et al. [39, 55] explore the why-not and why questions on reverse top-k queries, owing to its importance in multi-criteria decision making. Xiao et al. [96] models probabilistic reverse top-k queries over uncertain data in both monochromatic and bichromatic cases.

Reverse k-rank queries

Zhang et al. indicated that reverse top-k query in [85, 86] always returns an empty set for less-popular products. To ensure 100% coverage for any given query product, [106] proposed the reverse k-rank query, to find the top-k user preferences with the highest rank for a given object among all users. Mouratidis et al. [68] proposed maximum rank query, which can be seen as a monochromatic version of reverse k-rank query. Qian et al. [73] applied reverse k-rank query to large graphs.

3.2 Rank-aware query processing on dynamic data

3.2.1 Ranking queries

Continuous kNN queries

Our work is related to the problem of continuously searching for the spatial k nearest neighbor (kNN) queries over moving objects. This kind of research [66, 67, 97, 105] aims to keep the kNN moving objects to a fixed location (query point). Mouratidis et al. [66] proposed a grid index and the concept of the influence region which enlighten us to design a grid-based index for both dynamic spatial keyword objects and queries. However, the above solutions do not consider the keyword similarity so these techniques cannot be extended to our research.

Continuous top-k preference queries

Regard to the continuous top-k preference query on stream data, Das et al. [25] proposed a problem of supporting independent top-k queries over streams. Yu et al. [104] studied the problem of processing a large scale of continuous top-k queries, which maintains different ranking lists of multi-attribute objects for different preference queries. Hou U et al. [83] studied the problem of monitoring the document stream and continuously reports to each user the top-k documents that are most relevant to her keywords.

Moving top-k spatial keyword queries

An example of a “static objects, moving query” is keeping top-k gas-stations for a driving car. Literally, in this continuous query, the query is moving but the objects are unaltered. Wu et al.

first proposed a continuously moving top-k spatial keyword (MkSK) query [94, 95] using a filtering technique of safe-region on multiplicatively weighted Voronoi cells. Huang et al. [44] pointed out that the ranking function in [95] is ad-hoc. Hence, they studied MkSK queries with a general weighted sum ranking function and proposed a hyperbola-based safe-region to filter objects. In the above research, the spatial similarity is evaluated using the Euclidean distance. On the other hand, Zheng et al. [107] studied a continuous boolean top-k spatial keyword query over a road network with the techniques of the graph. Guo et al. [40] studied a continuous top-k spatial keyword query with a combined ranking function.

Top-k Spatial-keyword Publish/Subscribe

Another type of research of continuous search on spatial keyword objects is the Publish/Subscribe system. Users register their interests as continuous queries into the Publish/Subscribe system, then new streaming objects are delivered to relevant users. Similar to a snapshot spatial keyword search, the research topic is also separated by the matching function. Boolean matching is studied in [13,53,91], while the combined value between spatial similarity and keyword relevance are considered in [14, 90]. R. Mahmood et al. [59] proposed a frequency-aware structure to index spatial keyword objects. Besides processing continuous search spatial keyword objects in a single machine, many works focus on solving the problem in a parallel and distribute way. R. Mahmood et al. [57, 60] proposed a system named TORNADO based on the Apache Storm platform [1]. Chen et al. [17] proposed a distributed processing system which has a good load balancing among processing workers.

3.2.2 Reverse ranking queries

For reverse ranking queries on dynamic data, Benetis et al. [3, 4] frist proposed a reverse nearest neighbor queries for moving objects. Vlachou et al. [88] proposed efficient algorithms for processing distance-based reverse top-k queries over mobile devices. Yu. et al. [104] studied on the problem of processing a large number of continuous preference top-k queries. A dynamic index is proposed to support the reverse top-k query with a scalable solution for processing many continuous top-k queries.

[1] http://storm.apache.org/

Chapter 4

Aggregate Reverse Rank Queries

Finding top-rank products based on a given user's preference is a user-view rank model that helps users to find their desired products. Recently, another query processing problem named reverse rank query has attracted significant research interest. The reverse rank query is a manufacturer-view model and can find users based on a given product. It can help to target potential users or find the placement for a specific product in marketing analysis.

Unfortunately, previous reverse rank queries only consider one product, and they cannot identify the users for product bundling, which is known as a common sales strategy. To address the limitation, we propose a new query named *aggregate reverse rank query* to find matching users for a set of products. Three different aggregate rank functions (SUM, MIN, MAX) are proposed to evaluate a given product bundling in a variety of ways and target different users. To resolve these queries more efficiently, we propose a novel and sophisticated bound-and-filter framework. In the bound phase, two points are found to bound the query set for excluding candidates outside the bounds. In the filter phase, two tree-based methods are implemented with the bounds. For the situation of high-dimensional data, we propose a grid index method which uses pre-calculated score bounds to reduce multiplications in the simple scan. The theoretical analysis and experimental results demonstrate the efficacy of the proposed methods.

4.1 Introduction

Suppose that there are two types of datasets: user dataset and product dataset. The top-k query and reverse k-rank query are two different kinds of view-models. The top-k query is a user view-model that helps users by obtaining the best k products matching a user's preference. On the other hand, the reverse k-rank query [106] supports manufacturers by discovering potential users by retrieving the most appropriate user preferences. Therefore, it is a manufacturer view-model, and can be used as a tool for analysis and estimating product marketing.

Example 1. *Figure 4.1 shows an example of a reverse 1-rank query. Five different books (p_1–p_5) are scored on the attributes "price" and "rating". The preferences of three users (Tom, Jerry and Spike), consist of the weights for all attributes of the book. The score of a book w.r.t to a user is found by the inner product value of the book attribute and user preference vectors (Figure 4.1b). The results of the reverse 1-rank query are given in the last cells of Figure 4.1b* [1]*. For example, Jerry believes that p_2 is the best book, while Tom and Spike think that it is the second-best. Jerry is more likely to buy p_2 than Tom and Spike are, based on this ranking; hence, the reverse 1-rank query returns Jerry as a result.*

4.1.1 Motivation

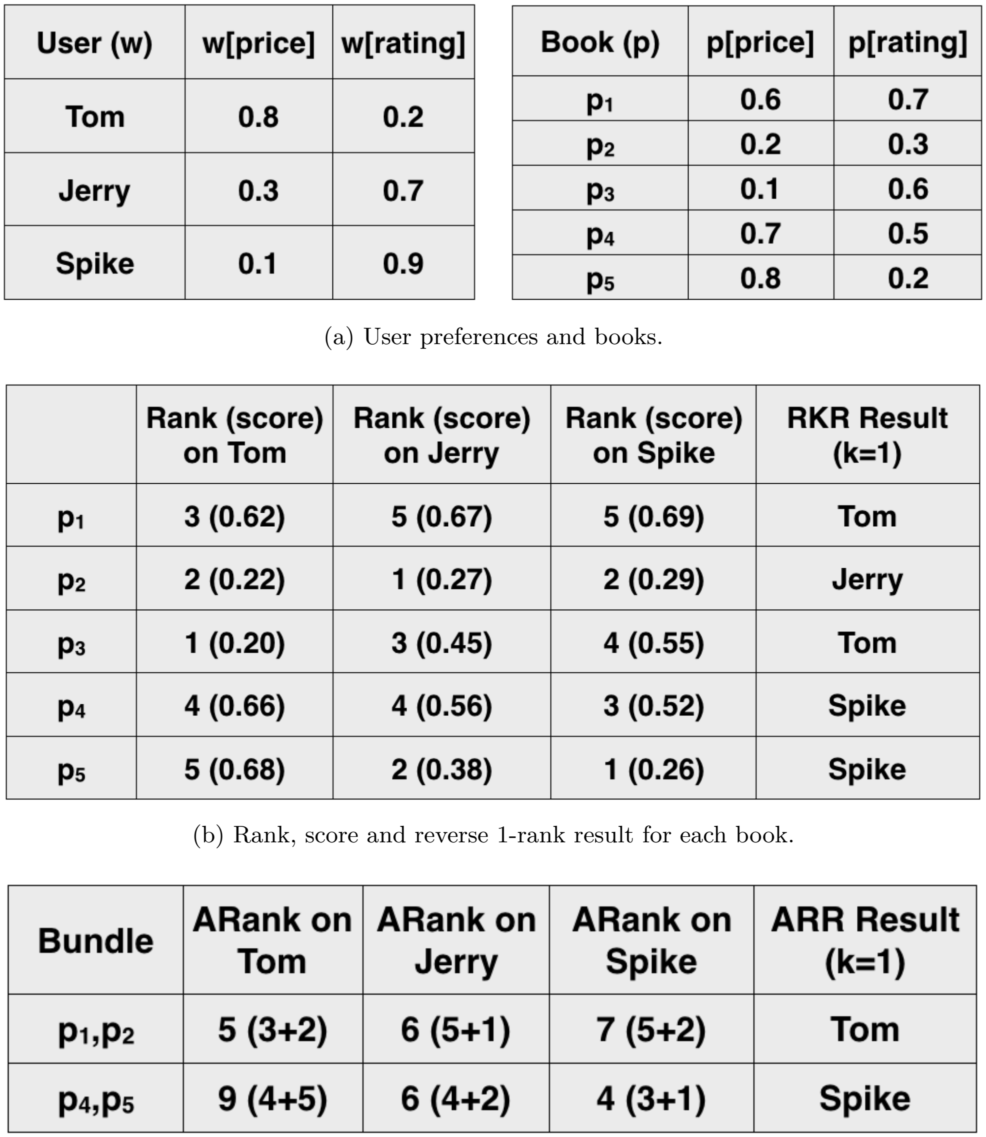
Besides the case of the single-product selling in Figure 4.1, manufacturers also use "product bundling" [2] for many marketing purposes. Product bundling offers several products for sale as one combined product. It is a common feature in many imperfectly competitive product markets. For example, Microsoft Co., Ltd. includes a word processor, spreadsheet, presentation program, and other useful software into a single Office Suite. The cable television industry often bundles various channels into a single tier to expand the channel market. Manufacturers of video games are also willing to group a popular game with other games of the same theme in the hope of obtaining more benefits by selling them together.

Because product bundling is a common business approach, helping manufacturers target users for their bundled products becomes an important issue. Unfortunately, the previous work on reverse k-rank query and other kinds of reverse rank queries [85, 86] were all designed for just one product. To address this limitation, we propose a new query definition named *aggregate reverse rank query* (*ARR query*) that finds k users with the smallest aggregate rank values.

Example 2. *Figure 4.1c shows an example of an ARR query with the* SUM *function. In this case, two books (p_1 and p_2) are bundled as a set for sale. ARR query evaluates aggregate rank (*ARank*) with the sum of each book, so the bundle's rank is $3 + 2 = 5$ based on Tom's preference. This ARR*

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

[2] https://en.wikipedia.org/wiki/Product_bundling



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

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

(a) User preferences and books.

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

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

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

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

Figure 4.1: Example of reverse rank query and aggregate reverse rank query.

query returns Tom as its result $(k = 1)$ *because Tom would rank a bundle of* p_1 *and* p_2 *higher than others would.*

Contribution. This chapter makes the following contributions:

- To the best of our knowledge, we are the first to address the "one product" limitation of the reverse rank query. We propose a new *ARR* query that returns k user preferences that best match a set of products.
- We propose a bound-and-filter framework for low-dimensional data. In the bounding phase, we preprocess preferences to determine possible upper and lower bounds. We also reduce query points with MAX/MIN aggregate function. In the filtering phase, we prune data with R-tree and Cone$^+$ tree.
- We propose a grid-index method for high-dimensional data. Grid-index method uses pre-calculated score bounds to reduce multiplications in the simple scan, it outperforms tree-based algorithms with the high-dimensional data. We also carry out a theoretical analysis to figure out an appropriate value of the grid size.
- Along with the theoretical analysis, we also perform experiments on both real and synthetic data. The experimental results validate the efficiency of the proposed methods.

4.1.2 Definitions

Preliminary definitions

The assumption of the product data, preference data, and the score function are the same as in the related research [85,87,106]. Let there be a set of products P and a set of preferences W. P and W are in a d-dimensional Euclidean space. Each product in the product dataset $p \in P$ is a d-dimensional vector that contains d nonnegative values. p is represented as a point $p = (p[1],\ p[2],\ \dots,\ p[d])$ where $p[i]$ is the attribute value of p in the ith dimension. The preference $w \in W$ is also a d-dimensional weighting vector, and $w[i]$ is a nonnegative weight that evaluates the ith attribute of products, where $\sum_{i=1}^{d} w[i] = 1$. The score of a product p based on a preference w is defined as the inner product of p and w expressed by $f(w,p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. Given q as the query product, which is in the same space of P, but not necessarily an element of P, the reverse k-rank query [106] is defined as follows:

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

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

Symbols	Description
P	set of products.
W	set of preferences.
Q	Query products.
d	Data dimensionality.
$f(w,p)$	The score of p based on w with inner product.
$p[i]$	Value of a product p in the ith dimension.
MBR	Minimum bounding rectangle.
$MBR(A).up$ $(MBR(A).low)$	left low point (right up point) of the MBR for a point set A.
e_p (e_w)	An MBR in Rtree of data set P (W).
$Q.up$, $Q.low$	Bounding points of Q.
$Grid$	Grid-index
$p^{(a)}$	Approximate index vector of a point p
$P^{(A)}$	Approximate index vectors set $\forall p \in P$
n	Number of partitions in grid-index
$L[f(w,p)], U[f(w.p)]$	Lower (Upper) bound of score of p on w

Table 4.1: Symbols and Notation

Aggregate Reverse Rank Query

As the above statement indicates, it is desirable for sellers to find potential users of their product bundles by using the reverse rank technique. Such queries can be dealt by extending the reverse rank query for more than one query point. We propose the *aggregate reverse rank query*, which is formally defined as follows.

Definition 3. (*aggregate reverse rank query, ARR*). *Given a point set P, weighting vector set W, positive integer k, and a set of query points Q, the ARR query returns the set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $ARank(w_i, Q) \leq ARank(w_j, Q)$ holds. If multiple w's have an equal ARank(.) value around boundary (k-th rank) of S, S contains a part of them randomly for the result.*

Aggregate rank function, denoted as ARank(w, Q), is the function used to evaluate the ranking of the query product set Q, which is the bundled product for which we want to find the target users.

$$\begin{aligned} &\bullet\ \textbf{SUM} : ARank(w,Q) = \sum_{q \in Q} rank(w,q) \\ &\bullet\ \textbf{MAX} : ARank_M(w,Q) = \operatorname*{Max}_{q \in Q}(rank(w,q)) \\ &\bullet\ \textbf{MIN} : ARank_m(w,Q) = \operatorname*{Min}_{q \in Q}(rank(w,q)) \end{aligned} \tag{4.1}$$

Then, the above three evaluating functions correspond to the following requests:

- **SUM**: Find users who more strongly believe than other users that this product set is better.

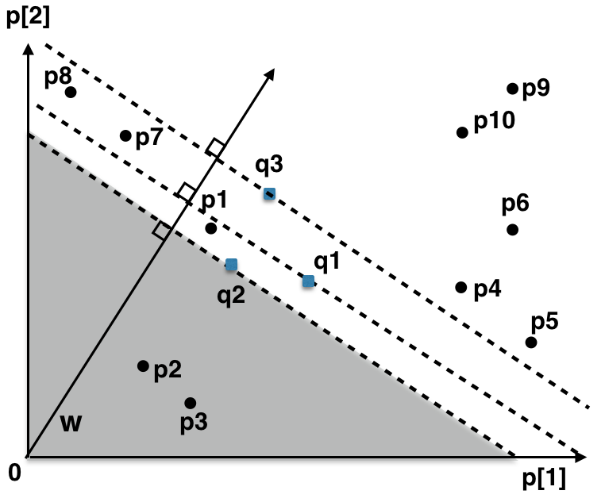


Figure 4.2: Geometric view of rank in 2-dimensional data, $ARank(w, Q) = 3 + 2 + 5 = 10$, $ARank_M(w, Q) = 5$, $ARank_m(w, Q) = 2$.

- **MAX/MIN**: Find users who more strongly believe than other users that the worst/best product in this set is better.

Example 3. *Figure 4.2 shows the geometric image of rank in a 2-dimensional data space of P. One product data $p \in P$ is represented as a point and a user preference w is represented as a vector. The score of inner product $f(w, p)$ is equal to the distance from o to the projection of p on w. The line that crosses the point p and is perpendicular to w is a borderline of the score $f(w, p)$. Obviously, the rank of q on w is the number of points under this borderline. For example, p_2 and p_3 are under the perpendicular line passing through q_2; hence, $f(w, p_2) < f(w, q_2)$ and $f(w, p_3) < f(w, q_2)$. By Definition 2, $rank(w, q_2) = 2$. For the aggregate rank of $Q = \{q_1, q_2, q_3\}$: $ARank(w, Q) = rank(w, q_1) + rank(w, q_2) + rank(w, q_3) = 3 + 2 + 5 = 10$; $ARank_M(w, Q) = rank(w, q_3) = 5$; $ARank_m(w, Q) = rank(w, q_2) = 2$.*

4.2 Solution for low-dimensional data: Bound-and-filter framework

4.2.1 Bound phase: Bound queries

The naive solution to the *ARR* query is to sum up the ranks for $q \in Q$ one by one against each $w \in W$ and $p \in P$. This is inefficient, especially when Q is large. In this section, we introduce the bounding phase of our bound-and-filter framework, in which a sophisticated method determines two

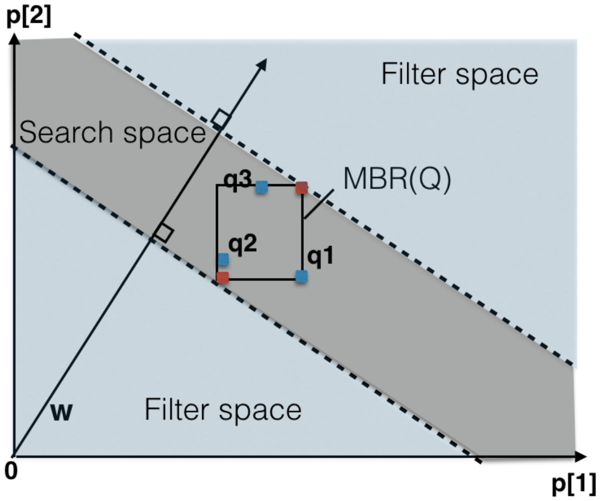


Figure 4.3: A 2-dimensional example of search space (gray) and filtering space (blue) with basic MBR(Q) bounding.

points $Q.up$ and $Q.low$ that bound Q. Our proposal is to bound the query point set Q with respect to W to avoid checking each $q \in Q$.

An intuitive method for bounding Q is to bound with the left-low corner and right-up corner points of Q's minimum bounding rectangle (MBR), denoted as $MBR(Q).low$ and $MBR(Q).up$. In the general case, $MBR(Q).low$ is dominated by any $q \in Q$ in all d dimensions, because the attribute values of $MBR(Q).low$ is always smaller than or equal to that of q. Moreover, all values are nonnegative so that the score function $f(w, q)$ is monotonically increasing; thus, it is obvious that for an arbitrary w, the score of $MBR(Q).low$ is smaller than or equal to that of $q \in Q$:

$$f(w, MBR(Q).low) \leq f(w, q),\ where\ q \in Q, w \in W. \tag{4.2}$$

On the other hand, $MBR(Q).up$ is the upper bound of Q in a similar way.

Example 4. *Figure 4.3 shows the search space and filter space of data P with $MBR(Q)$. For computing the $ARank(Q, u)$, the "search space" is the space in which we need to compute the scores of the inside data. "Filter space" means that we do not need to compute the data inside and just need to filter them since they have a clear relationship with Q. The search space is the middle part between the two perpendicular lines w.r.t $MBR(Q).low$ and $MBR(Q).up$. Apparently, a tighter bound (higher $MBR(Q).low$ and/or lower $MBR(Q).up$) can make this middle space smaller and filter more data in processing.*

Tighter bounding strategy

Motivated by the above observation, we propose a tighter bounding strategy. To bound Q for an arbitrary $w \in W$, we first find out the top-weighting vector in each dimension, denoted as $w_t^{(i)}, i = 1, 2, 3, .., d$. $w_t^{(i)}$ is the closest vector (the smallest angle) to the orthonormal basis vector of the ith dimension, as defined in the following.

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

A subset of W, denoted by $W_t = \{w_t^{(i)}\}_1^d$, is the set of *top-weighting vectors* for all dimensions. Because W_t contains the border of the weighting vector in all dimensions, we can use it to find the upper bound and lower bound points set of Q.

Definition 5. (*upper and lower bound query sets Q_u and Q_l*). *Let Q be a set of d-dimensional queries.*

$Q_u = \{q_i | q_i \in Q \wedge \forall q_j \in Q, \exists w_t^{(i)} \in W_t, f(w_t^{(i)}, q_i) \geq f(w_t^{(i)}, q_j)\}$ *and*
$Q_l = \{q_i | q_i \in Q \wedge \forall q_j \in Q, \exists w_t^{(i)} \in W_t, f(w_t^{(i)}, q_i) \leq f(w_t^{(i)}, q_j)\}$.

By definition, for each $w_t^{(i)}$, we can find a $q_i \in Q_u$ (Q_l) such that q_i's score with respect to $w_t^{(i)}$ is the largest (smallest) among Q. Different $w_t^{(i)}$ may apply to the same q_i. Generally, it is easy to find the MBR of a point set Q_u, and its upper-right and lower-left corners are the two bounding points required.

$$Q.up = MBR(Q_u).up \tag{4.3}$$

$$Q.low = MBR(Q_l).low \tag{4.4}$$

Example 5. *Figure 4.4 shows the example of Q.low and Q.up where $Q = \{q_1, q_2, q_3\}$. $w_t^{(1)} = w_5$ and $w_t^{(2)} = w_1$ are the top-weighting vectors in dimensions 1 and 2, respectively. Each $w_t^{(i)}$ is also a normal vector of the hyperplanes $H(w_t^{(i)})$. For Q.up, in 2-dimensional space, the hyperplanes $H(w_t^{(1)})$ are the dashed lines l_1, which are perpendicular to $w_t^{(1)}$. By sweeping l_1 parallelly from far infinity toward the original point $(0, 0)$, q_1 is the first point that is touched. Hence, q_1's score with respect to w is equal to $max_{q \in Q} f(w_t^{(1)}, q)$, and q_1 is included in Q_u. In the same manner, l_2 touches q_3 first; hence, $q_3 \in Q_u$. $Q.up = MBR(Q_u).up$ upper-bounds the scores for Q_u. Similarly, sweeping the perpendicular dashed lines l_3 and l_4 from $(0, 0)$ toward infinity, both touch q_2; hence $Q_l = \{q_2\}$ and $Q.low = q_2$. We show here that Q.up and Q.low bound the query set Q for ARR query.*

Theorem 1. *(Correctness of Q.up and Q.low) Given a set of d-dimensional query points Q, a set*

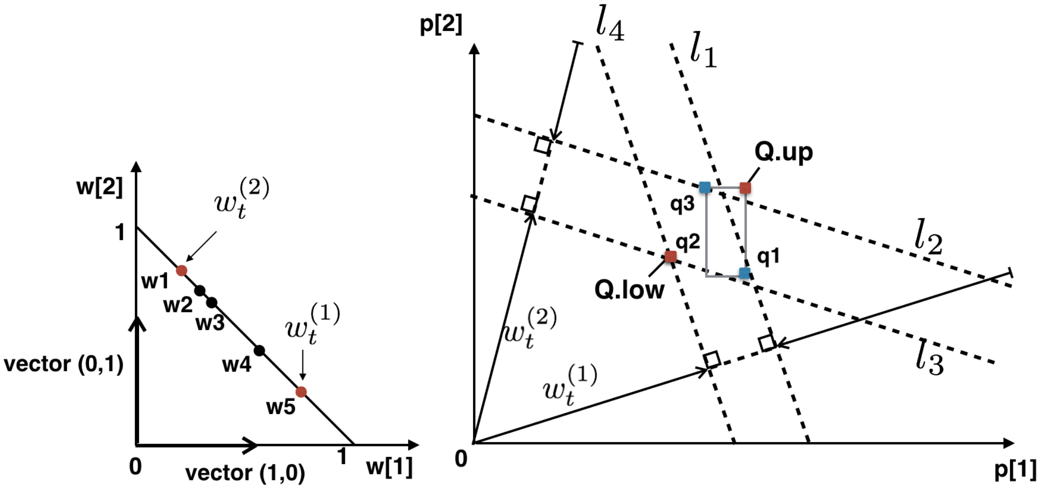


Figure 4.4: A 2-dimensional example. $w_t^{(1)} = w_5$, $w_t^{(2)} = w_1$, and $Q_u = \{q_1, q_3\}$, $Q_l = \{q_2\}$, $Q.low = MBR(Q_l).low = q_2$, $Q.up = MBR(Q_u)$.up

of weighting vectors W, and the bounds of Q: $Q.up$ and $Q.low$. For each $w \in W$ and each $q \in Q$, $f(w, Q.low) \leq f(w, q) \leq f(w, Q.up)$ always holds.

Proof. By contradiction. For $Q.up$, assume that $\exists q \in Q, q \notin Q_u$ holds so that $f(w, q) \geq f(w, Q.up)$. Therefore, $\exists q[i] > Q.up[i], i \in [1, d]$; therefore, there must exist a $w_t^{(j)} \in W_t, j \in [1, d]$, where W_t is a set of top-weighting vectors that makes $f(w_t^{(j)}, q)$ the maximum value, and q should be in Q_u. This leads to the contradiction (The geometric view is that there exists a hyperplane $H(w_t^{(j)})$ that first touches q rather than others.). A similar contradiction occurs with $Q.low$. □

We can use the rank of $Q.low$ to infer the bounds of the aggregate rank of Q for the three aggregate rank functions SUM, MIN, and MAX in Equation (4.1).

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

Proof. $\forall q_i \in Q, \forall w \in W$, it holds that $f(w, q_i) \geq f(w, Q.low)$; hence, $rank(w,\ q_i) \geq rank(w,\ Q.low)$. By definition, $ARank(w,\ Q) = \sum rank(w,\ q_i) \geq |Q| \times rank(w,\ Q.low)$; hence, $|Q| \times rank(w,\ Q.low)$ is the lower bound of $ARank(w,\ Q)$. Similarly, $|Q| \times rank(w,\ Q.up)$ is the upper bound. □

Corollary 2. (*Aggregate rank bounds of Q for w, MAX/MIN*): *Given a set of query points Q and a weighting vector w, the lower bound of $ARank_{M(m)}(w, Q)$ is $rank(w, Q.low)$, and the upper bound of $ARank_{M(m)}(w, Q)$ is $rank(w, Q.up)$.*

We only prove the MIN function case since the MAX function case is similar.

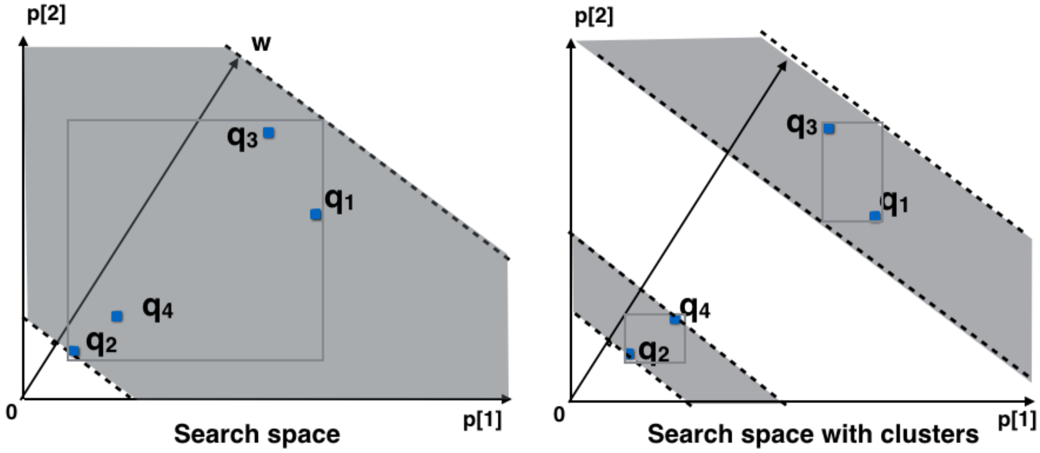


Figure 4.5: Search space of P.

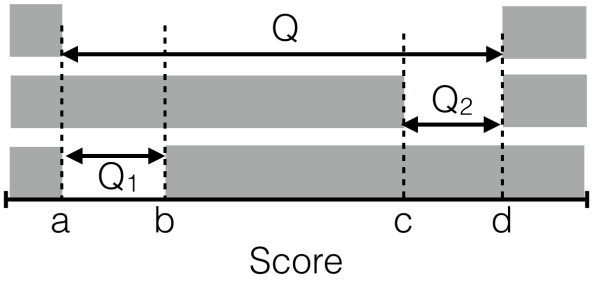


Figure 4.6: The score ranges against the whole Q, Q_1 and Q_2

Proof. $\forall q_i \in Q$, $\forall w \in W$, it holds that $Min(f(w, q_i)) \geq f(w, Q.low)$; hence, $Min(rank(w, q_i)) \geq rank(w, Q.low)$. By Equation 4.1, $ARank_m(w, Q) = Min(rank(w, q_i)) \geq rank(w, Q.low)$. Similarly, $rank(w, Q.up)$ is the upper bound of $ARank_m(w, Q)$. □

Bound queries with clusters

Recall that 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 search space is shown in Figure 4.5, where the search area is the area sandwiched between the two dashed lines. In the worst case, when Q is distributed as wide as the whole space, then the efficiency will degrade to a brute-force search.

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 4.6 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. Applying this result to the clusters, the lower bound (LB) becomes:

$$LB = \tilde{a} \cdot |Q_1| + \tilde{c} \cdot |Q_2| = \tilde{a} \cdot |Q| + (\tilde{c} - \tilde{a}) \cdot |Q_2| > \tilde{a} \cdot |Q| \tag{4.5}$$

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

Regarding to the clustering method, we simply utilize x-means [72], 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)[3] 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).

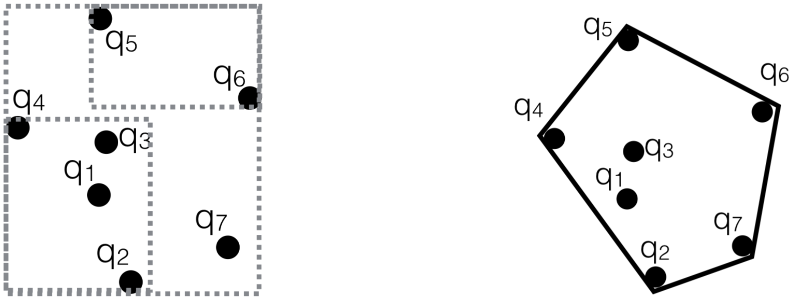
Reducing queries for MAX/MIN function

The MAX and MIN functions in ARR query enable to analyze the potential users who are interested only in the worst or best product in a product bundle. Different from the SUM function that requires summing up the ranks for all $q \in Q$, it is sufficient to only check the necessary q's instead of the whole Q. This means that we can make the proposed bound-and-filter framework more efficient. In this section, we introduce a method of reducing Q for the MAX and MIN functions.

It is natural to believe that such necessary query points are those located in the up-right and low-left parts of Q for MAX and MIN functions, respectively. Defined in the following, the *left corner MBR* and the *right corner MBR* try to remove unnecessary query points of Q.

Definition 6. (*Left-corners and right-corners, CM*). *Let Q be a point set. The left-corners of Q is a subset of Q and is denoted as $CM(Q).l$. $CM(Q).l$ contains the points in the MBR formed by $\{h^{(i)}\}_{i=1}^{d}$, where $h^{(i)} \in Q$ satisfies: (1) $h^{(i)}[i] = MBR(Q).low[i]$, and (2) $h^{(i)}$ is the nearest point to $MBR(Q).low$ among all points p satisfying $h^{(i)}[i] = MBR(Q).low[i]$. The right-corners of Q, denoted as $CM(Q).r$ is defined in a similar manner.*

[3] https://en.wikipedia.org/wiki/Bayesian__information__criterion



(a) Corner MBRs.

(b) Convex Hull.

Figure 4.7: Two ways to reduce Q.

Example 6. *Figure 4.7a gives an example of* left-corners *and* right-corners *in 2-dimensional data. There are q_1 to q_7 in a query points set Q. q_4 is the nearest point to $MBR(Q).low$ and is on the left vertical edge of $MBR(Q)$. On the down horizontal edge, q_2 is the nearest point to $MBR(Q).low$. Therefore, $CM(Q).l = \{q_1, q_2, q_3, q_4\}$ is formed by $\{q_2, q_4\}$. In the same way, $CM(Q).r = \{q_5, q_6\}$ is formed by $\{q_5, q_6\}$.*

Obviously, the points in $CM(Q).l$ have smaller values in all dimensions than the other points of Q. Therefore, given an arbitrary w:

$$\underset{q \in Q}{\text{Min}} \quad rank(w, q) = \underset{q \in CM(Q).l}{\text{Min}} rank(w, q). \tag{4.6}$$

In other words, the $q \in Q$ that minimizes $rank(w, q)$ is always found from $CM(Q).l$. Similarly, the $q \in Q$ that maximizes $rank(w, q)$ is always found from $CM(Q).r$.

Lemma 3. (*Reduce Q to $CM(Q)$*)*: Given a set of query points Q, $ARank_m(w, Q) = ARank_m(w, CM(Q).l)$ and $ARank_M(w, Q) = ARank_M(w, CM(Q).l)$.*

Another way to reduce Q for MAX and MIN functions is to build the convex hull of Q. The convex hull is the smallest convex set that contains all $q \in Q$, and we denote the vertices set of the convex hull of Q by $CH(Q)$.

Example 7. *Figure 4.7b shows the convex hull of the given Q in the same example, where $CH(Q) = \{q_2, q_4, q_5, q_6, q_7\}$. Viewing a point as a vector, when $q \in Q$ are projected to an arbitrary vector w, both the shortest and the longest length of projection are from $CH(Q)$, because the vertices of the convex hull are the boundary points. Recall that the inner product $f(w, q)$ is equal to the length of q's projection on w, and $CH(Q)$ contains such q's that minimize and maximize $rank(w, q)$.*

Lemma 4. (*Reduce Q to $CH(Q)$*)*: Given a set of query points Q, $ARank_m(w, Q) = ARank_m(w, CH(Q))$ and $ARank_M(w, Q) = ARank_M(w, CH(Q))$.*

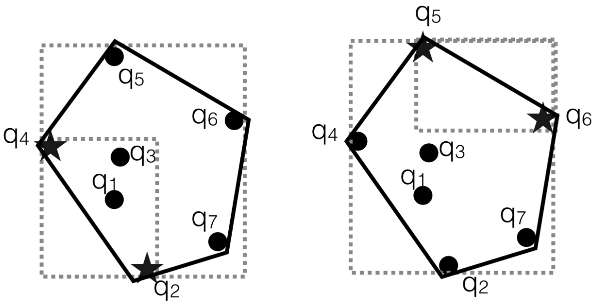


Figure 4.8: Query points in convex hull vertices and corner MBRs. Necessary queries for MAX function: $\{q_2, q_4\}$, for MIN function: $\{q_5, q_6\}$.

Taking the advantages of both $CM(Q)$ and $CH(Q)$, we can only check the query points at their intersection. Lemma 3 and Lemma 4 help to conclude the following Theorem.

Theorem 2. (*Correctness of query reducing*) *Given a set of query points* Q*, let* $Q_m = CM(Q).l \cap CH(Q)$ *and* $Q_M = CM(Q).r \cap CH(Q)$*. Then* $ARank_m(w, Q) = ARank_m(w, Q_m)$ *and* $ARank_M(w, Q) = ARank_M(w, Q_M)$*.*

Algorithm 1 Reduce Q

```
Input: $Q$
Output: reduced set $Q_R$
$CH(Q) \Leftarrow ConvexHull(Q).getVertex()$
if MIN function then
    $CM(Q) \Leftarrow get\ CM(Q).l$
if MAX function then
    $CM(Q) \Leftarrow get\ CM(Q).r$
$Q_R \Leftarrow CM \cap CH(Q)$
return $Q_R$
```

The above theorem guarantees that for MAX/MIN *ARR* queries, we only need to process the reduced Q_M/Q_m instead of the original Q. Figure 4.8 shows an example of reducing Q. $CH(Q) = \{q_2, q_4, q_5, q_6, q_7\}$, $CM(Q).l = \{q_1, q_2, q_3, q_4\}$ and $CM(Q).r = \{q_5, q_6\}$. By Theorem 2, the reduced query set for the MAX function is $Q_M = CH(Q) \cap CM.l = \{q_2, q_4\}$, and the reduced query set for the MIN function is $Q_m = CH(Q) \cap CM.r = \{q_5, q_6\}$.

4.2.2 Filter phase: Prune P data

Instead of comparing the product data $p \in P$ one by one with the query bounds, we use the index to compare similar data simultaneously, thus making the process efficient. The idea is to index the dataset P in an R-tree to group similar points, and compares the bounds of MBRs (the R-tree entries, also denoted by e) with $Q.up$ and $Q.low$ to reduce computing costs.

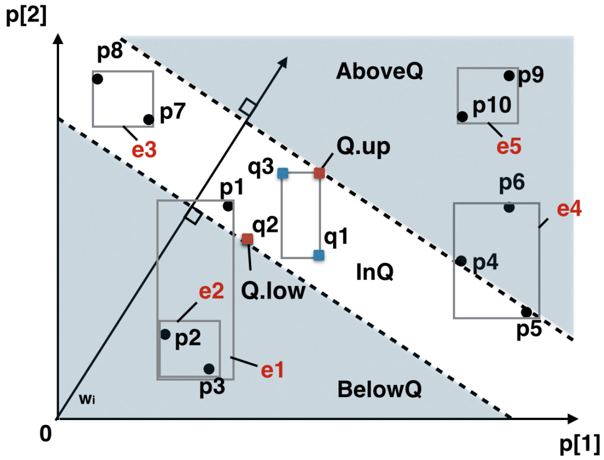


Figure 4.9: The sub-spaces of BelowQ, InQ, and AboveQ based on $Q.low$ and $Q.up$ with a single w_i in the 2-dimensional space of dataset P.

Example 8. *First, we introduce how to filters P with $Q.low$ and $Q.up$. Figure 4.9 shows the geometric view for an example of 2-dimensional data. The two dashed lines across the bounds $Q.low$ and $Q.up$ respectively, and are perpendicular to the weighting vector w_i, and form the boundary values of the score. The space is partitioned into three parts, which are marked as BelowQ, InQ, and AboveQ in Figure 4.9. For example, e_2 is in BelowQ and e_5 is in AboveQ. MBRs in BelowQ or AboveQ can be filtered by checking the upper and lower boundaries; otherwise, it needs further refinement.*

Formally, the pruning rules are as follows. Notice that the filtering methodology of partitioned spaces can also apply to the multiple dimensional spaces.

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

ARank-P algorithm

Given P, w, Q, $Q.up$, $Q.low$, and a positive integer *minRank*, the ARank-P algorithm checks whether the aggregate rank of Q is smaller than the given *minRank*. It also returns the value of

Algorithm 2 ARank-P

```
Input: $P, w, Q, minRank, Q.up, Q.low$
Output: return $rnk$ when $w$ should be included
        return $-1$ when $w$ should be discarded
 1: $rnk \Leftarrow 0$, $Cand \Leftarrow \emptyset$
 2: Initialize $heapP$ as an empty queue structure.
 3: $heapP.enqueue(RtreeP.Root())$
 4: while $heapP.isNotEmpty()$ do
 5:   $e_p \Leftarrow heapP.dequeue()$
 6:   for each child $e_i \in e_p$ do
 7:     if $f(w, e_i.low) < f(w, Q.up)$ then
 8:       if $e_i$ in $BelowQ$ then
 9:         $rnk \Leftarrow rnk + Counting(e_i, Q)$        //Rule 1
10:         if $rnk \geq minRank$ then
11:           return -1
12:       else if $e_i$ in $InQ$ then
13:         $Cand \Leftarrow Cand \cup e_i$              //Rule 3
14:       else
15:         if $e_i$ is a data point then
16:           $Cand \Leftarrow Cand \cup e_i$
17:         else
18:           $heapP.enqueue(e_i)$
19: Refine $Cand$ by processing the MBRs and points in $Cand$ with each $q \in Q$.
20: if $rnk \leq minRank$ then
21:   return rnk
22: else
23:   return -1
```

the aggregate rank when $ARank(w, Q) < minRank$. Algorithm 2 shows that the ARank-P function uses the R-tree to prune similar points in a node of the R-tree. In this algorithm, the counter rnk is used to count the aggregate rank of Q (Line 1). Then, the algorithm recursively checks the MBRs in the R-tree of P from the root (Line 2). If e_i is contained in $BelowQ$,

the counter rnk is increased by the return value from the counting function in Equation 4.7, which is based on Corollary 1 and Corollary 2.

$$Counting(e, Q) = \begin{cases} e.size \times |Q|, & \textbf{SUM} \\ e.size, & \textbf{MAX} \text{ or } \textbf{MIN} \end{cases} \tag{4.7}$$

When rnk becomes larger than $minRank$, the algorithm returns -1 to terminate (Lines 9–10). If e_i overlaps the space of InQ, then it is added into the candidate set $Cand$ for refinement, and either it is an internal node (Lines 11–12) or data point (Lines 14–15). Otherwise, e_i is added to the queue (Line 17). After the traversal of RtreeP, refinement is performed where the $Cand$ set is compared with each $q \in Q$ and rnk is updated (Line 18). Note that $Cand$ contains both the MBR

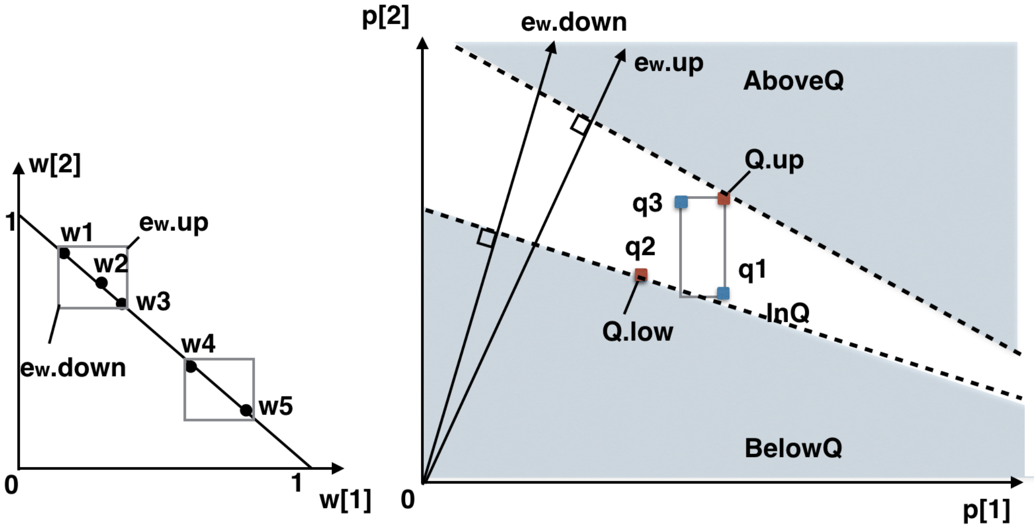


Figure 4.10: The sub-spaces of BelowQ, InQ, and AboveQ based on $Q.low$ and $Q.up$ with an MBR e_w in the 2-dimensional space of dataset P.

and data point p in InQ. The refinement also considers the upper and lower bounds of the MBR to filter each q. As the results, rnk is returned as the aggregate rank if $rnk < minRank$, or -1 is returned that indicates that the current w is not a result.

4.2.3 Filter phase: Prune W data

Prune W data with the R-tree

Similar to pruning P data with the R-tree, We also index the W set in an R-tree so that removing redundant computing by grouping similar w. The R-trees for P and W are denoted as $RtreeP$ and $RtreeW$, respectively.

Example 9. *Figure 4.10 shows the three parts of BelowQ, InQ, and AboveQ, which are separated by the bounds of the MBR e_w in $RtreeW$ and Q.up (Q.low). Based on the MBR features in $RtreeP$ and $RtreeW$, we can obtain the score bounds of a single data point on the MBR e_w of $RtreeW$.*

Lemma 5. (*Score bound of* p)*: Given an MBR with the weighting vector e_w in $RtreeW$ and $p \in P$, the score $f(w,p)$ is lower-bounded by $f(e_w.low,p)$ and upper-bounded by $f(e_w.up,p)$.*

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

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

Algorithm 3 ARank-WP

```
Input: $P, e_w, Q, minRank, Q.up, Q.low$
Output: return $rnk$ when all $w \in e_w$ should be included
        return $-1$ when all $w \in e_w$ should be discarded
        return $-1$ when it is uncertain
$rnk \Leftarrow 0$, $Cand \Leftarrow \emptyset$
Initialize $heapP$ as an empty queue structure.
$heapP.enqueue(RtreeP.root())$
while $heapP.isNotEmpty()$ do
  $e_p \Leftarrow heapP.dequeue()$
  for each child $e_i \in e_p$ do
    if $f(e_w.low, e_i.low) < f(e_w.up, Q.up)$ then
      if $e_i$ in $BelowQ$ then
        $rnk \Leftarrow rnk + Counting(e_i, Q)$
        if $rnk \geq minRank$ then
          return -1
      else if $e_i$ in $InQ$ then
        $Cand \Leftarrow Cand \cup e_i$
      else
        if $e_i$ is a data point then
          $Cand \Leftarrow Cand \cup e_i$
        else
          $heapP.enqueue(e_i)$
Refine $Cand$ and process the MBRs and points in $Cand$ with each $q$.
if $rnk \leq minRank$ then
  return 1
else
  return 0
```

Lemma 6. (*Score bound of MBR*): *Given the MBR e_w of $RtreeW$ and the MBR e_p of $RtreeP$, the score of every $p \in e_p$ is lower-bounded by $f(e_w.low, e_p.low)$ and upper-bounded by $f(e_w.up, e_p.up)$.*

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

By the above Lemmas, we can construct the bounds of the aggregate rank for Q on MBR e_w. Corollary 1 and Corollary 2 lead to the following conclusion almost straightly.

Theorem 3. (*Aggregate rank bounds of Q for e_w*): *Given the set of query points Q and the MBR of the weighting vector e_w. For the SUM function, the lower bound of rank for every $w \in e_w$ is $|Q| \times rank(e_w.low, Q.low)$, and the upper bound of $ARank(w,Q)$ is $|Q| \times rank(e_w.up, Q.up)$. For the MAX/MIN function, the lower bound of rank for every $w \in e_w$ is $rank(e_w.low, Q.low)$, and the upper bound is $rank(e_w.up, Q.up)$.*

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

```
Input: $P, W, Q, Q.up, Q.low$
Output: result set $heap$
 1: initialize $heap$ with the first $k$ weighting vectors and the aggregate ranks of $Q$
 2: $minRank \Leftarrow heap$'s last rank.
 3: $heapW.enqueue(RtreeW.root())$
 4: while $heapW.isNotEmpty()$ do
 5:   $e_w \Leftarrow heapW.dequeue()$
 6:   if $e_w$ is a single weighting vector then
 7:     call the function ARank-P and update $minRank$.
 8:   else
 9:     $flag \Leftarrow$ ARank-WP$(P, e_w, Q, minRank, Q.up, Q.low)$
10:     if $flag = 0$ then
11:       $heapW.enqueue$(all children $\in e_w$)
12:     else
13:       if $flag = 1$ then
14:         for each $w \in e_w$ do
15:           call the function ARank-P and update $minRank$.
16: return $heap$
```

The ARank-P algorithm computes the rank of Q with respect to a single w. Instead, ARank-WP in the algorithm checks a node of the R-tree that contains multiple similar w's. Algorithm 3 helps check these $w \in e_w$ with Q and $minRank$. The algorithm returns 1 if all $w \in e_w$ make the Q rank in $minRank$ and returns -1 if none of $w \in e_w$ makes the Q rank better than $minRank$. Otherwise, the algorithm returns 0, indicating that e_w cannot be filtered and its children entries need to be checked.

DTM indexes both P and W in two R-trees. Hence, it enables the pruning of both the weighting vectors and points. Algorithm 4 shows the detail of DTM. DTM checks the nodes in $RtreeW$, and calls Algorithm 3 to check the aggregate rank of Q on a node e_w (Line 9). If $flag$ (the returned value from ARank-WP) is 0, all children MBRs are added to *heapW* for further check (Lines 10–11). If $flag$ is 1, this means that every w in e_w makes Q rank better than $minRank$. Thus, Algorithm 2 computes the rank of each w in e_w and *heap*, which keeps the best k answers so far, and *minRank* are updated (Lines 14–15). When the leaf node of a single w is being checked, Algorithm 2 is called (Lines 6–7). When the algorithm terminates, *heap* is returned as the result of the aggregate reverse rank query.

Prune W data with the $Cone^+$ tree

Figure 4.11 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 $\sum_{i=1}^{d} w[i] = 1$. 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*

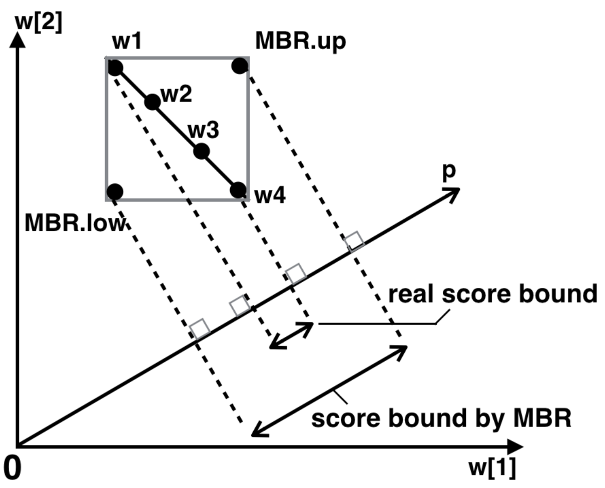


Figure 4.11: The difference of score bounds created by MBR and real score bounds.

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. In conclusion, it is not an appropriate way since the bounding points in spatial MBR will over-enlarge the bound from the actual inner product value.

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 [74] 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.

Cone tree [74] 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 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 [74], 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 [74] does not hold in ARR queries.

From Figure 4.11, 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 7.

Lemma 7. *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*

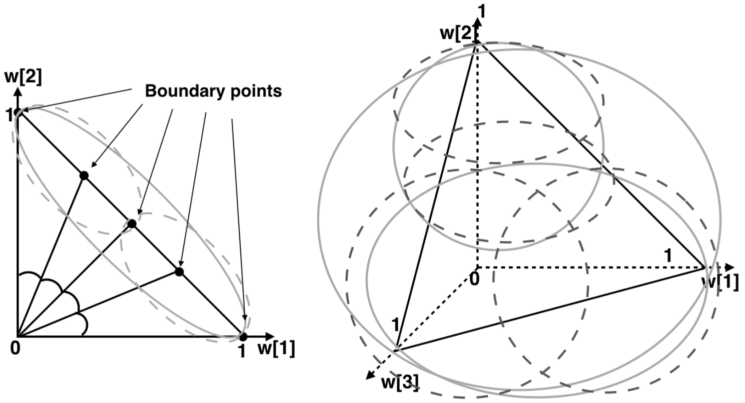


Figure 4.12: Images of 2-dimensional (left) and 3-dimensional (rigth) $cone^{+}$ tree, respectively.

$B.boundar$.

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 7 and proposed a $cone^{+}$ tree which keeps the boundary points for each node. Therefore the precise score bounds can be computed directly. Figure 4.12 shows the example of $cone^{+}$ tree. Algorithm 5 and 6 show the construction of $cone^{+}$ tree. The indexed boundary points are the points containing the maximum value on a single dimension (Algorithm 6, Line 2).

Algorithm 5 $Cone^{+}$TreeSplit($Data$)

Input: points set, $Data$

Output: two centering points of children, a, b ;

```
Select a random point $x \in Data$.
$a \Leftarrow \arg\max_{x' \in Data} cosineSim(x, x')$
$b \Leftarrow \arg\max_{x' \in Data} cosineSim(a, x')$
return {a,b}
```

Algorithm 6 Build$Cone^{+}$Tree($Data$)

```
Input: points set, $Data$
Output: $cone^{+}$ tree, $tree$ ;
$tree.data \Leftarrow Data$
$tree.boundary \Leftarrow \{w \in Data : \underset{i\in[0,d)}{\arg\max}\, w[i]\}$
if $|Data| \leq M_n$ then
    return $tree$
else
    $\{a, b\} \Leftarrow Cone^{+}TreeSplit(Data)$
    $left \Leftarrow \{p \in Data : cosineSim(p, a) > cosineSim(p, b)\}$
    $tree.leftChild \Leftarrow BuildCone^{+}Tree(left)$
    $tree.rightChild \Leftarrow BuildCone^{+}Tree(Data - left)$
    return $tree$
```

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

$$\begin{cases} Below\ Q : \\ \qquad Max_{w_b \in c_w.boundar}\{f(w_b, e_p.up)\} < \\ \qquad Min_{w_b \in c_w.boundar}\{f(w_b, Q.low)\} \\ Above\ Q : \\ \qquad Min_{w_b \in c_w.boundar}\{f(w_b, e_p.low)\} > \\ \qquad Max_{w_b \in c_w.boundar}\{f(w_b, Q.up)\} \\ \mathrm{U}nknow : otherwise \end{cases} \tag{4.8}$$

We can apply the above *cone*$^{+}$ tree bounds easily to Algorithm 3 by using above Equation (4.8) to define the filter space in Lines 7 and 12.

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

Table 4.2: Time cost for reading data and processing reverse rank queries with 6-dimensional data.

4.3 Solution for high-dimensional data: Grid-index Method

4.3.1 Curse of the dimensionality

By now, we proposed tree-based methods for *ARR* query. However, as pointed out by [6, 12, 92], these tree-based methods suffer from similar problems: When processing high-dimensional data sets, the performance declines to even worse than that of linear scan.

For real-world applications, there also has a requirement to process *ARR* for high dimensional data. 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. Therefore, processing *ARR* 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 tree structure to solve it.

Despite its performances advantages on high-dimensional queries, there are challenges in processing *ARR* with the simple scan. *ARR* 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 $O(|P| \times |W|)$. *ARR* 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 *ARR* queries and for the pairwise computations in the inner product. Table 4.2 shows that the time taken to read different sizes of data file is almost negligible in the *ARR* processing. Rather, the major cost of processing *ARR* is the pairwise computations. We also found that the proportion of pairwise computations in processing *ARR* 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 *ARR* efficiently.

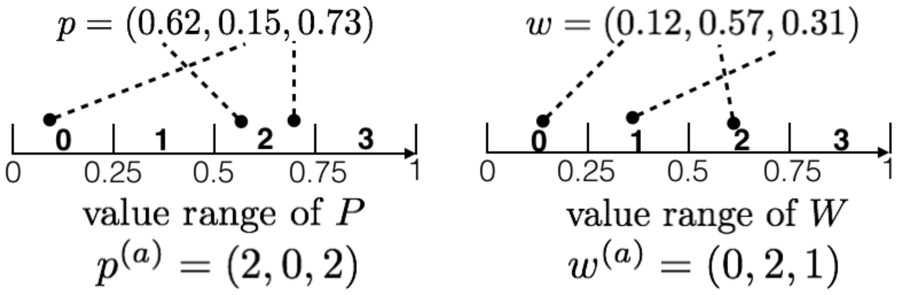


Figure 4.13: Equally dividing value range into 4 partitions, allocating real values into approximate intervals and getting the approximate vector $p^{(a)}$ and $w^{(a)}$.

4.3.2 Grid-index Method

For the above reasons, we develop an optimized version of the simple scan, called the Grid-index method (GIM) 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 GIM 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.

According the above statement, 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.

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 4.13 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 $p^{(a)} = (2, 0, 2)$

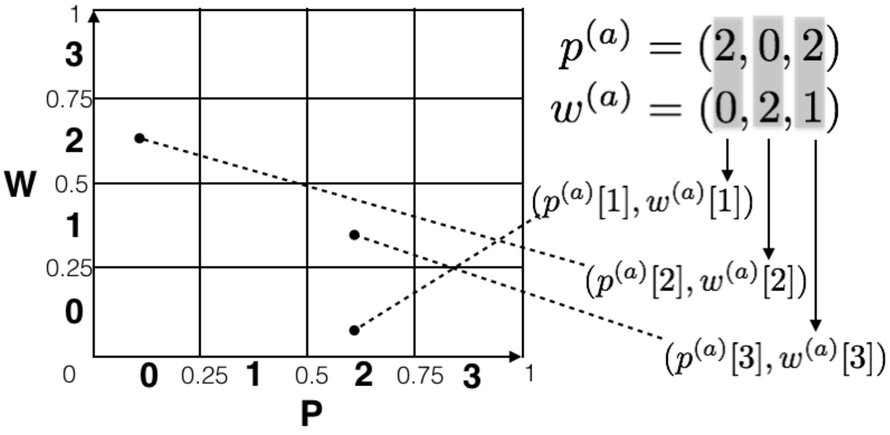

$p^{(a)} = (2, 0, 2)$
$w^{(a)} = (0, 2, 1)$
$(p^{(a)}[1], w^{(a)}[1])$
$(p^{(a)}[2], w^{(a)}[2])$
$(p^{(a)}[3], w^{(a)}[3])$
W
P
1
3
0.75
2
0.5
1
0.25
0
0
0
0.25
1
0.5
2
0.75
3
1

Figure 4.14: 4 × 4 Grids for points and weighting vectors, mapping $p^{(a)}$ and $w^{(a)}$ onto Grids.

and $w^{(a)} = (0, 2, 1)$.

Since the inner product is the sum of pairwise multiplications of $p[i]$ and $w[i]$, we combine the ranges of p and w to form the grids. Figure 4.14 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 $(p[i], w[i]),\ i = \{1, 2, 3, ..., d\},\ p \in P$ 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 4.13, $\alpha_p = \alpha_w = (0, 0.25, 0.5, 0.75, 1)$. 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:

$$Grid[i][j] = \alpha_p[i] \cdot \alpha_w[j],\ i, j \in [0, n] \tag{4.9}$$

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 (4.9), $Grid[2][0] = 0.5 \times 0$, $Grid[2 + 1][0 + 1] = 0.75 \times 0.25$, meaning $0.5 \times 0 \leq p[1] \cdot w[1] \leq 0.75 \times 0.25$.

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:

$$L[f(w,p)] \leq f(w,p) \leq U[f(w,p)] \tag{4.10}$$

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

$$L[f(w,p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i]][w^{(a)}[i]] \tag{4.11}$$

$$U[f(w,p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i]+1][w^{(a)}[i]+1] \tag{4.12}$$

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

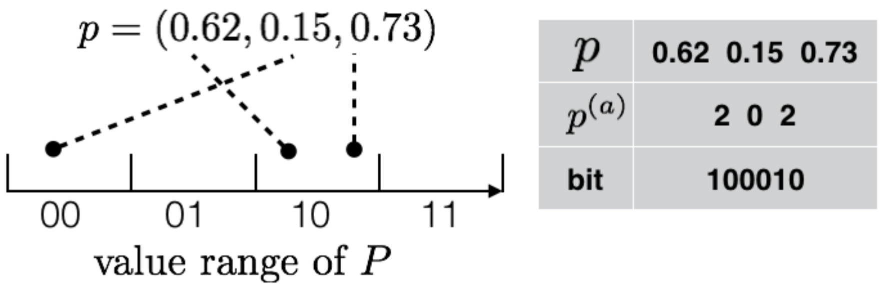
- Case 1 ($p \prec_w q$): If $U[f(w,p)] < f(w,q)$, p precedes q, p has a higher rank than q with w.
- Case 2 ($q \prec_w p$): If $L[f(w,p)] > f(w,q)$, q precedes p, p does not affect the rank of q with w.
- Case 3 ($p \asymp q$): Otherwise, p and q are incomparable, i.e., $L[f(w,p)] \leq f(w,q) \leq U[f(w,p)]$. 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 (4.11) and (4.12). 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.

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 4.15 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 4.3.3, $b = 6$ 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 [4]. This kind of bit-string compressing technique is also used in [92].

[4] 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.



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

Figure 4.15: 6-bit string for compressing the p to $p^{(a)}$.

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.

Algorithm 7 GIM

```
Input: $P_A, W_A, P, W, Q, k$
Output:
 1: $buffer \leftarrow \emptyset$
 2: $kthRank \leftarrow \infty$
 3: for each $w_a \in W_A$ do
 4:     $Candidate \leftarrow \emptyset$
 5:     $Counter \leftarrow Domin.size$
 6:     Compute each $f(w, q)$ of $q \in Q$ and store in $Q_{scores}$
 7:     Sort $Q_{scores}$ in descending
 8:     for each $p_a \in (P_A\ /\ Domin\ )$ do
 9:         for $i$ to $|Q|$ do
10:             if $U[f(w, p)] \leq Q_{scores}[i]$ then
11:                 $Counter \leftarrow Counter + |Q|$ - $i$
12:                 if $counter == kthRank$ then
13:                     break to check next $w_a$
14:             if $L[f(w, p)] \leq Q_{scores}[i] \leq U[f(w, p)]$ then
15:                 $Candidate \leftarrow Candidate \cup \{p\}$
16:         if $p$ dominates $Q$ then
17:             $Domin \leftarrow Domin \cup \{p\}$
18:     Refine $Candidate$ and update $Counter$.
19: if $Counter \geq kthRank$ then
20:     continue to next $w_a$
21: else
22:     $buffer.insert(w, Counter)$
23:     $kthRank \leftarrow$ the kth rank in $buffer$.
24: return buffer
```

Grid-index Aggregate Reverse Rank Algorithm (GIM)

Algorithm 7 describes the proposed method GIM. It is a double looping framework that scans each $p_a \in P_A$ for each $w_a \in W_A$, and look up the Grid-index to avoiding computing. For each w_a, we use a *Counter* to record the aggregate rank $ARank(w, Q)$ (Line 4). In the inner loop(Line 8-17), the *Counter* will be update when the $f(w, q)$ is greater than a current point p. We carry out an optimization that calculates and sorts each q's score in advance (Line 6-7). We use Grid-index to obtain the $U[f(w, p)]$, if $U[f(w, p)]$ is smaller than a q's score, the *Counter* will increase by the remained number of q and we don't need to compare more (Line 10-11). If the rank relationship can not decide by the upper and lower from Grid-index, we add these kind of p into *Candidate* (Line

14-15), and we will refine the *Candidate* after scanning all p_a if it is necessary (Line 18). Another optimization is that "global dominating point", a p is a global dominating point if all $p[i] \leq q[i]$ where $i = 1, 2, ..., d$, and we keep them into a set *Domin* (Line 16-17). We will skip checking the global dominating points (Line 8) instead of initializing *Counter* by the size of *Domin* (Line 5). A k-element *buffer* is used to keep the top-k w's and their aggregate ranks for Q as the result of *ARR* query (Line 1, 22). The algorithm will break and start to check the next w_a when the *Counter* reaches *kthRank*, which is initialize as ∞ (Line 2) and update by the rank value of the last element in *buffer* (Line 23).

4.3.3 Theoretical analysis

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

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 [106], MPA 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 [106] 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.

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 4.3 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. It is a shortcoming of tree-based algorithms that the MBRs will always overlap with each other when the data is high-dimensional.

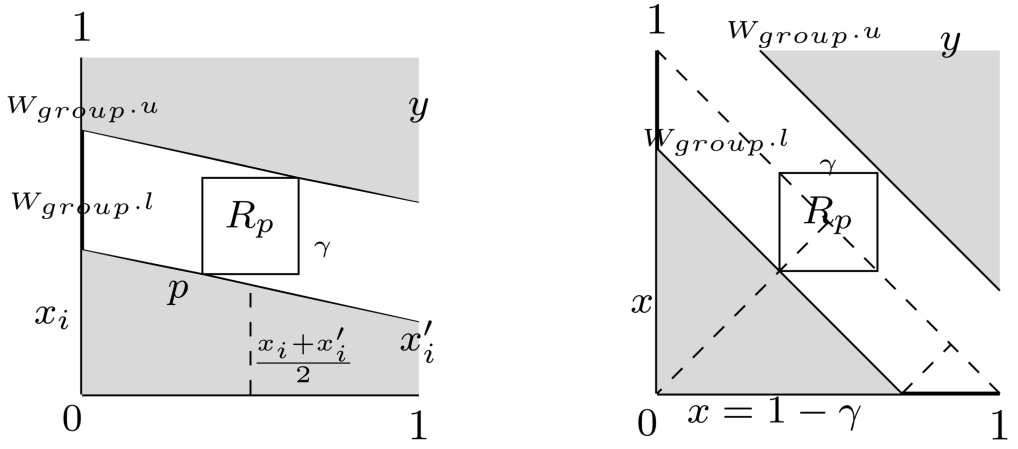
Besides the shortcoming from the tree-based index itself, we also found that the filterable space with tree-based methodology reduces as the dimensionality increases. This conclusion is supported by the following estimation.

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. The gray area is the safely filtered space. The shape of the gray area can be a

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



(a) Trapezoidal prism

(b) Tetrahedron

Figure 4.16: Two kinds of Filtering areas (gray) of R-tree.

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:

$$Vol = Vol_{TetraX} \cdot Vol_{PrismX} + Vol_{TetraY} \cdot Vol_{PrismY} \tag{4.13}$$

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

$$Vol = 2 \cdot Vol_{Tetra} \cdot Vol_{Prism} \tag{4.14}$$

Firstly, the volume of hyper-tetra is: [5]

$$Vol_{Tetra} = \frac{1}{g!}(\prod_{i=1}^{g} x_i) = \frac{1}{g!}(1-\gamma)^g \tag{4.15}$$

then, the volume of the hyper-prism (the area in Figure 4.16 (a)) is:

$$S_i = \frac{1}{2}(x_i + x_i') \cdot H \leq (\frac{1-\gamma}{2}) \leq \frac{1}{2} \tag{4.16}$$

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

$$Vol_{Prism3d} = \frac{1}{3}(S_1 + S_2 + \sqrt{S_1 S_2}) \cdot H \leq \frac{1}{2} \tag{4.17}$$

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

$$Vol_{max} = 2 \cdot \frac{1}{g!}(1-\gamma)^g \cdot \frac{1}{2} = \frac{1}{g!}(1-\gamma)^g \tag{4.18}$$

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 (4.18), 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 tree-structures 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] 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 $V_d = \int V_{d-1} dx \sim \frac{cx^d}{d}$.

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:

$$F(a,\ b) = 1 - Prob(a < S < b). \tag{4.19}$$

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:

$$Prob(\frac{k}{n^2} < w \cdot p < \frac{k+1}{n^2}) = \frac{1}{n^2},\ k = 1, 2, ..., n^2. \tag{4.20}$$

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:

$$Prob(\sum_{i=1}^{d}(w[i] \cdot p[i]) = s) \tag{4.21}$$

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:

$$t(x) = (x^1 + x^2 + ... + x^{n^2})^d \tag{4.22}$$

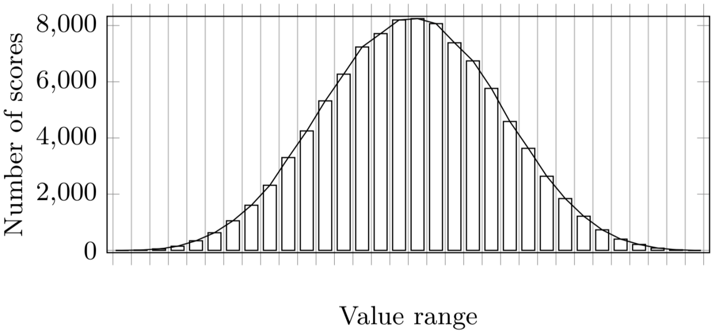


Figure 4.17: Grid-index scores distribution in dimension d = 4, partitions n = 4, $|P|$ = 100K, $|W|$ = 100K.

By [84], the probability of obtaining s score on d n-sided dice is

$$Prob(s,d,n) = \frac{1}{n^{2d}} \sum_{k=0}^{\lfloor (s-d)/n^2 \rfloor} (-1)^k \binom{d}{k} \binom{s-n^2k-1}{d-1} \tag{4.23}$$

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 (4.23). On the other hand, we found that the distribution of scores approaches a normal distribution, even in low dimensional cases, such as 4. Figure 4.17 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

$$\mu = \frac{1}{2}r \qquad \sigma = \frac{1}{2\sqrt{3}}r \tag{4.24}$$

The average score value of a point p is

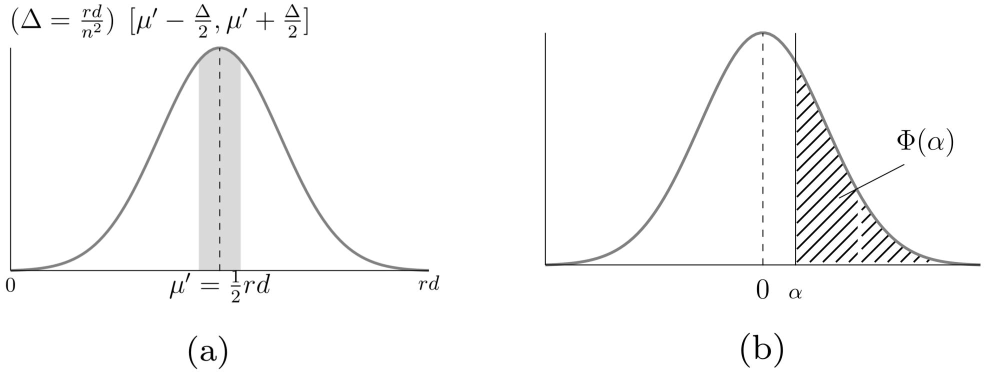
$$\overline{p \cdot w} = \frac{1}{d}\sum_{i=1}^{d}(p[i] \cdot w[i]) \tag{4.25}$$

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

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

$$Z = \frac{\sqrt{d}}{\sigma}(\overline{p \cdot w} - \mu) \tag{4.26}$$

follows the standard normal distribution (SND). In other words, $Z \sim N(0,1)$, where μ and σ are as in Equation (4.24).



(a) (b)

Figure 4.18: (a): The normal distribution of point scores $N(\mu', \sigma')$ and the largest probability interval (gray). (b): $\Phi(\cdot)$ of the SND showing $1 - \int_{-\alpha}^{\alpha} \cdot \ = 2\Phi(\alpha)$.

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 (4.24),

$$\mu' = \frac{1}{2}rd \qquad \sigma' = \frac{\sqrt{d}}{2\sqrt{3}}r \tag{4.27}$$

From Lemma 1 and (4.19), we may now estimate the filtering performance.

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

$$\begin{aligned} F(x, x+\Delta) &= 1 - Prob(x < S < x + \Delta) \\ &= 1 - \int_{x}^{x+\Delta} f(x)dx \end{aligned} \tag{4.28}$$

where

$$f(x) = \frac{1}{\sigma'\sqrt{2\pi}} exp(-\frac{(x-\mu')^2}{2\sigma'^2}) \tag{4.29}$$

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:

$$Z = \frac{d \cdot \overline{p \cdot w} - \mu d}{\sigma\sqrt{d}} = \frac{S - \mu'}{\sigma'} \tag{4.30}$$

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:

$$\Delta = \frac{r}{n^2}d \tag{4.31}$$

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 4.18(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 (4.30) and Equation (4.27), we find that $S_\Delta = \mu' \pm \frac{\Delta}{2}$ corresponds to

$$Z_\Delta = \frac{S_\Delta - \mu'}{\sigma'} = \frac{\mu' \pm \frac{\Delta}{2} - \mu'}{\sigma'} = \pm\frac{\sqrt{3d}}{n^2} \tag{4.32}$$

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

$$F(x, x+\Delta) > F_{worst}(x, x+\Delta) = 1 - \int_{\mu'-\frac{\Delta}{2}}^{\mu'+\frac{\Delta}{2}} f(x)dx = 2\Phi(\frac{\sqrt{3d}}{n^2}) \tag{4.33}$$

where $\Phi(\cdot)$ is the area.

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*

$$n > \sqrt{\frac{2\sqrt{3d}}{\delta}} \tag{4.34}$$

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

$$\Phi(\frac{\delta}{2}) = \frac{1-\epsilon}{2} \tag{4.35}$$

Proof. By Equation (4.34), $\frac{\delta}{2} > \frac{\sqrt{3d}}{n^2}$. Since Φ is a monotonically decreasing function (Figure 4.18), $\Phi(\frac{\sqrt{3d}}{n^2}) > \Phi(\frac{\delta}{2})$. Combining Equation (4.33) 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

$$\frac{\sqrt{3d}}{n^2} < \delta = 0.0125 \quad \longrightarrow n > \sqrt{\frac{2\sqrt{3d}}{\delta}} = \sqrt{80\sqrt{3d}} \tag{4.36}$$

If $d = 20$ then $n = 32$ satisfies Equation (4.36) hence a 32 × 32 Grid-index is enough for filtering

P W	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 4.4: Filtering performance of Grid-index with different distributions. $|P| = 100K$, $|W| = 100K$, $d = 6$, $n = 32$

.

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 4.4. Different σ between these distributions lead to slight differences in filtering power. But the filtering power is always efficient.

4.4 Experiments

In this section, we report the experimental results of aggregate reverse rank queries. We present the experimental evaluation of the proposed algorithms which were implemented in C++, and the experiments were run on a Mac with 2.6 GHz Intel Core i7 and 16 GB RAM. We report the above measurements with the average values over 100 times.

4.4.1 Data, algorithms and setting

Both synthetic and real data were employed for the dataset P and W.

Synthetic data. The synthetic datasets were uniform (UN), clustered (CL), and anti-correlated (AC) with an attribute value range of $[0, 1)$. We used the same method as in related work [85, 87, 106] to generate synthetic datasets: UN: All attribute values are generated independently and following a uniform distribution; CL: The cluster centroids are selected randomly and follow a uniform distribution. Then, each attribute is generated with the normal distribution; AC: Select a plane perpendicular to the diagonal of the data space. Then each attribute is generated in this plane and follows a uniform distribution.

Real data. We also have two real data sets: NBA[6] and AMAZON [7]. 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

[6] NBA: http://www.databasebasketball.com.

[7] AMAZON: http://jmcauley.ucsd.edu/data/amazon/.

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. We generated Q by using clustered data

Algorithms. We present the experimental evaluation of following methods:

- **NAIVE.** The brute force method.
- **DTM.** The basic method of bound-and-filter framework. Both P and W are indexed in R-tree.
- **CHDTM.** DTM method with reducing query strategy for MAX and MIN functions.
- **CPM.** Both P and W are indexed in R-tree. Q is divided into clusters.
- **C$^+$TM.** P is indexed in R-tree and W is indexed in Cone$^+$ tree.
- **CC$^+$M.** P is indexed in R-tree and W is indexed in Cone$^+$ tree. Q is divided into clusters
- **GIM.** Grid-index method. Using approximate values in grid-index to skip computations while linear scanning P and W.

4.4.2 Experimental results

UN data on vary dimensionality d. Figure 4.19 shows the experimental results for the synthetic datasets UN with varying dimensions d (2-5), where the *ARank* function is SUM. Both datasets P and W contained 100K tuples. Q had five query points, and the target is to find the five best preferences ($k = 5$) for this Q. According to the CPU cost comparison results shown in Figure 4.19a, DTM is at least ten times faster than the NAIVE method since it avoids checking each p and w, and the performance is also stable for various dimensional cases. Comparing to the DTM which is the basic bound-and-filter method, CPM and C$^+$TM boost the performance at least 1.2 times, and the combined method CC$^+$M takes advantages of them and be the best. Figure 4.19b and Figure 4.19c show that CC$^+$M had less I/O cost and less pairwise computations than other bound-and-filter methods.

Reducing queries on MAX and MIN aggregate functions. The comparison results with regard to the MAX and MIN functions are shown in Figure 4.20 and Figure 4.21. CHDTM optimizes DTM by reducing unnecessary $q \in Q$ on the MAX/MIN function. The experimental results also confirm that CHDTM was better than DTM in terms of CPU cost, I/O cost, and pairwise computations on UN data. Of course, the reducing strategy is a pre-processing technique, so it can also applied in other bound-and-filter methods.

CL and AC on vary dimensionality d. We also test other synthetic data CL and AC and the comparison results of *ARR* query with the SUM function are shown in Figure 4.22 and Figure 4.23. Regarding to the low dimensional data, CC$^+$M has better performance than other algorithms on both CL data and AC data for low dimensional data. Tree-based methods (DTM, CPM, C$^+$TM, CC$^+$M) require less querying time for CL data than other data distributions, the reason is that it is easier to index clustered data with the R-tree than other distributions. This also makes the combined CC$^+$M have less I/O cost and pairwise computations.

UN data on varying $|Q|$. For the varying $|Q|$ in Figure 4.24, because the number of products in a product bundle is not large, we test the Q from 5 to 25. The CPU time of branch-and-bound methods (DTM, CPM, C$^+$TM, CC$^+$M) has only a slight increase because they bounded Q in advance. However, the efficiency of the Naive and GIM decreased with increasing $|Q|$ since it had to calculate every $q \in Q$ for assembling the ARank(.) value.

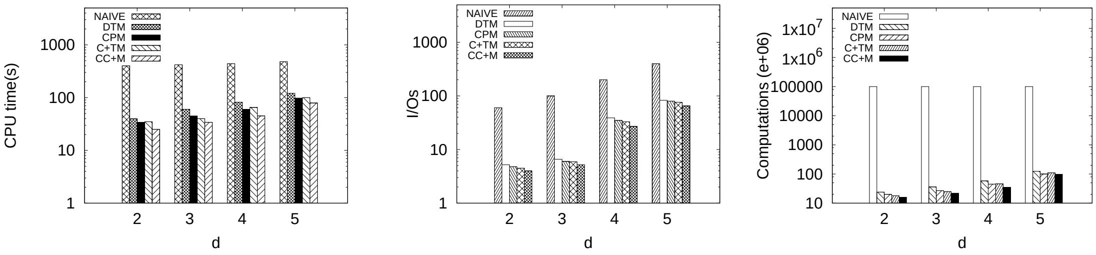
CL data on varying k. From the results provided in Figure 4.25, we can see that all algorithms are insensitive to k. This is because of the following two reasons: (a) k is far smaller than the cardinality of W and P. (b) In our proposed bound-and-filter framework, a k-element buffer in ascending order is kept to store the top-k $w's$ and their ranking while processing, and the comparing only happens with the last element (*minRank*) rather than all k candidates in the buffer.

Scalability Figure 4.26 shows the scalable property for varying $|P|$ and $|W|$. We show the results of $|P| = |W| =$ 100K, 500K, 1M. The CPU cost of CC$^+$M increases slightly with increasing $|P|$ and $|W|$ because the majority of pairwise computations were filtered by the strategy of the bound-and-filter.

Amazon data on vary k. Figure 4.27 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. As we expected, CC$^+$M is the fastest method since AMAZON is a low dimensional data set.

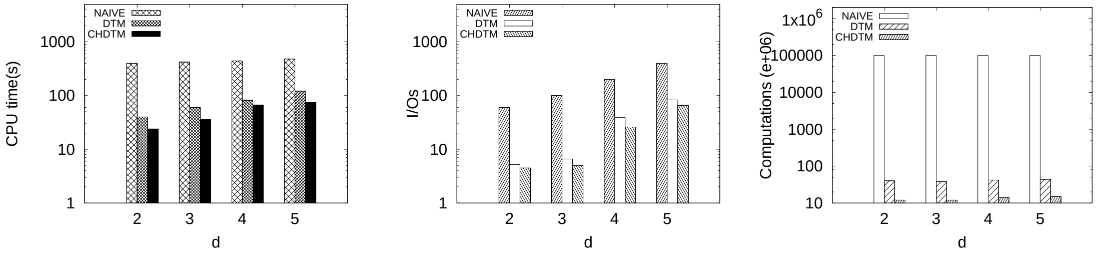
NBA data on vary q. Using the NBA dataset, the *ARR* query can answer such practical questions as ”who likes a team more than others?” We selected five, ten, and fifteen players from the same team as Q and then generated the dataset W as various user preferences. The three *ARank* functions represent different ways of thinking: To find the people who care about all players of an NBA team (SUM) or are just concerned about the most favorite/unlike player in a team (MAX/MIN). Figure 4.28 shows the results of NBA data. As expected, CC$^+$M found the answer the fastest since NBA is a low dimensional data set. The reducing query strategy also works with CHCC$^+$M method on NBA data.

High dimensional data. Figure 4.29 shows the comparison results on the high-dimensional data (6 - 20). Although the CC$^+$M is still the best one of the bound-and-filter methods, its performance decreases with the increasing dimensionality. As we said before, this results in a limitation of the proposed method using an R-tree (or any other spatial indexes) suffers from a problem named “Curse of Dimensionality”, and subsequently leads to low performance when processing high-dimensional data sets. According to the result, we can see that GIM method becomes faster than CC$^+$M method (and other bound-and-filter methods) after eight six dimensions.



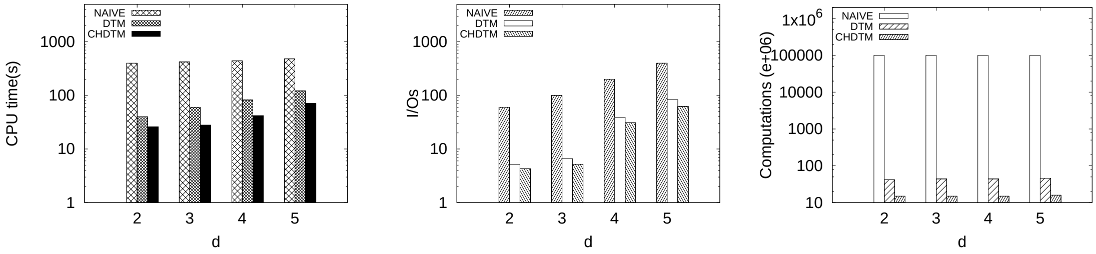
(a) CPU cost. (b) I/O cost. (c) Pairwise computations.

Figure 4.19: Comparison results of varying d on UN data with ARR query (SUM function), $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.



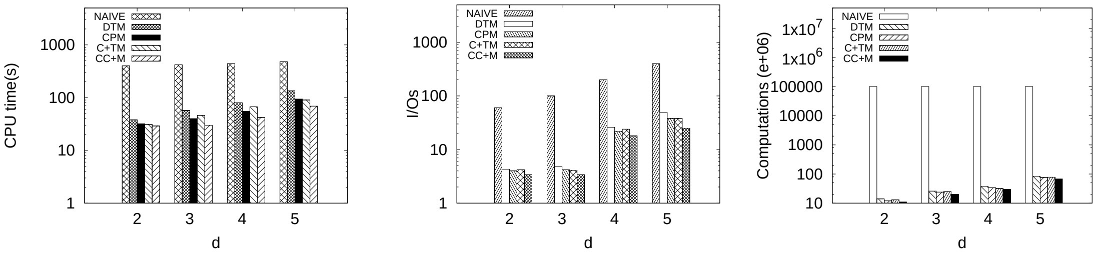
(a) CPU cost. (b) I/O cost. (c) Pairwise computations.

Figure 4.20: Comparison results of varying d on UN data with ARR query (MAX function), $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.



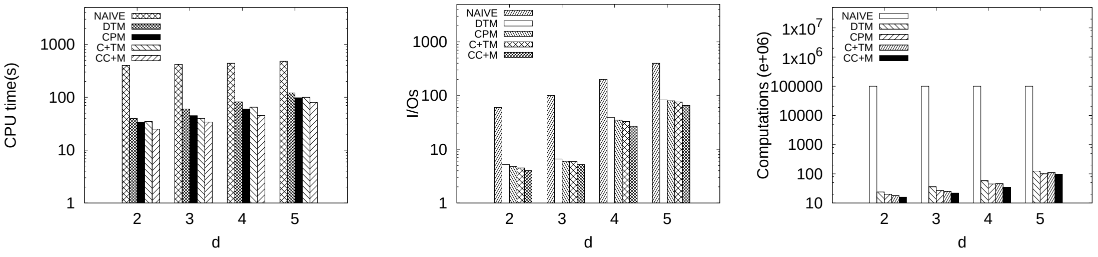
(a) CPU cost. (b) I/O cost. (c) Pairwise computations.

Figure 4.21: Comparison results of varying d on UN data with ARR query (MIN function), $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.



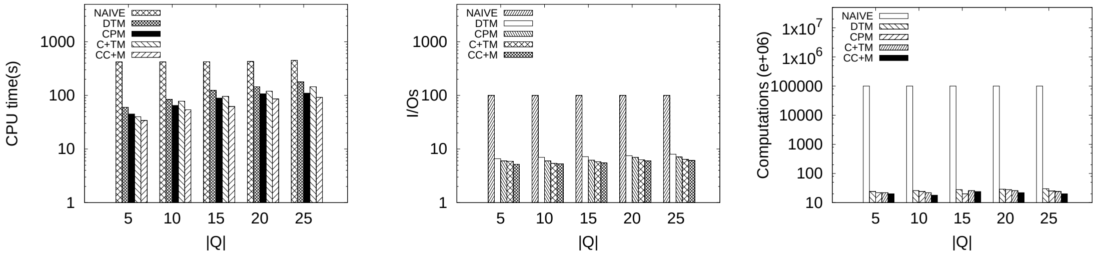
(a) CPU cost. (b) I/O cost. (c) Pairwise computations.

Figure 4.22: Comparison results of varying d on CL data with *ARR* query (SUM function), $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.



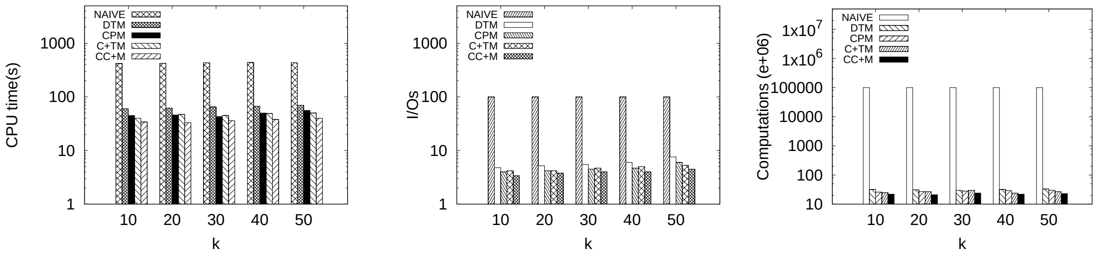
(a) CPU cost. (b) I/O cost. (c) Pairwise computations.

Figure 4.23: Comparison results of varying d on AC data with *ARR* query (SUM function), $|P| = |W| = 100$K, all with $|Q| = 5$, $k = 10$.



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

Figure 4.24: Comparison results of varying $|Q|$ on UN data with *ARR* query (SUM function), $|P| = |W| = 100$K, all with $d = 3$, $k = 10$.

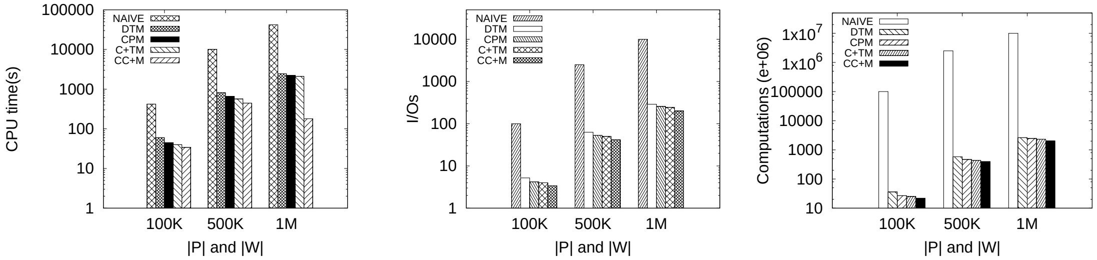


(a) CPU cost.

(b) I/O cost.

(c) Pairwise computations.

Figure 4.25: Comparison results of varying k on CL data with *ARR* query (SUM function), $|P| = |W| = 100\text{K}$, all with $|Q| = 5$, $d = 3$.

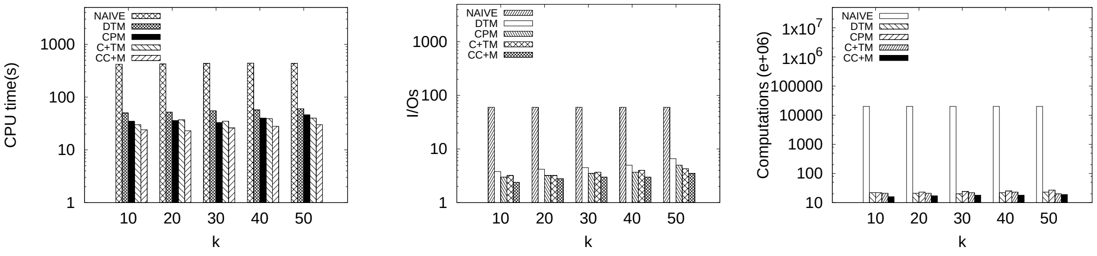


(a) CPU cost.

(b) I/O cost.

(c) Pairwise computations.

Figure 4.26: Scalability on varying $|P|$ and $|W|$, P: UN data, W: UN data, all with $k = 10$, $|Q| = 5$, $d = 3$.

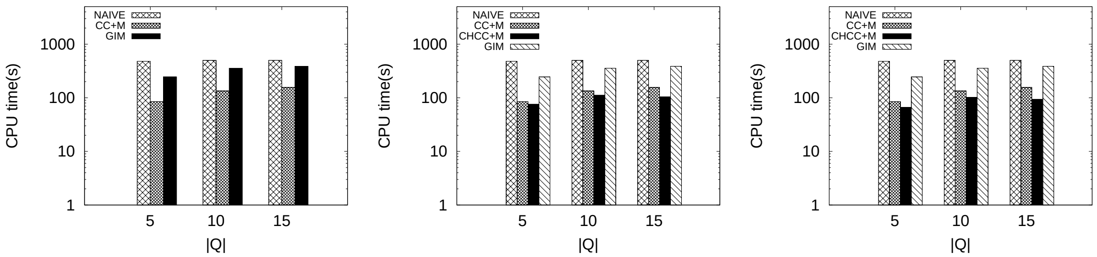


(a) CPU cost.

(b) I/O cost.

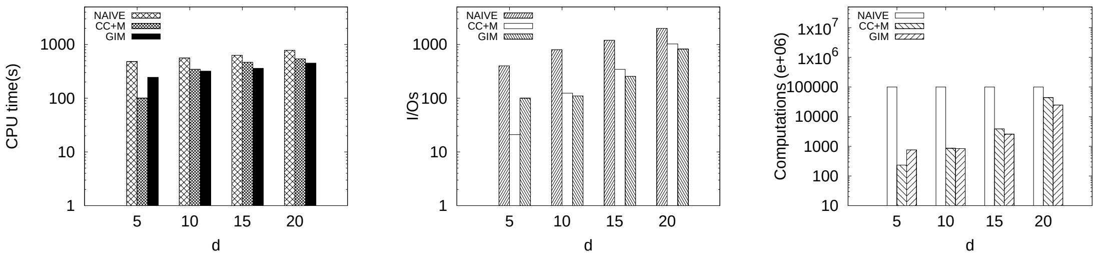
(c) Pairwise computations.

Figure 4.27: Comparison results of varying k on AMAZON data with *ARR* query (SUM function), W: UN data, $|W| = 100\text{K}$, all with $|Q| = 5$.



(a) CPU cost (SUM). (b) CPU cost (MAX). (c) CPU cost (MIN)

Figure 4.28: Comparison results of varying $|Q|$ on NBA data with *ARR* query (SUM, MAX, MIN functions), W: UN data, $|W| = 100$K, all with $k = 10$.



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

Figure 4.29: Comparison results of high dimensional UN data with *ARR* query (SUM function), $|P|$ = $|W| = 100$K, all with $|Q| = 5$, $k = 10$.

4.5 Summary

Reverse rank queries have become important tools in marketing analysis. However, related research on reverse rank queries has focused on only single product, which cannot deal with the common sale strategy, product bundling. We proposed the aggregate reverse rank query (*ARR*) to address the situation of product bundling where multiple query products exist. Three different aggregate rank functions (SUM, MIN, MAX) were defined to target potential users in three normal views. To solve *ARR* efficiently, we devise a novel bound-and-filter framework to with low-dimensional data. In bound-and-filter framework, queries are bounded to calculate an approximate aggregate rank value efficiently, then tree-based structures are used to filter data in processing. For the situation of high-dimensional data, we proposed a grid index method which uses pre-calculated score bounds to reduce multiplications in the simple scan. We compared the methods through experiments on both synthetic data and real data.

Chapter 5

Weighted Aggregate Reverse Rank Queries

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

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

5.1 Introduction

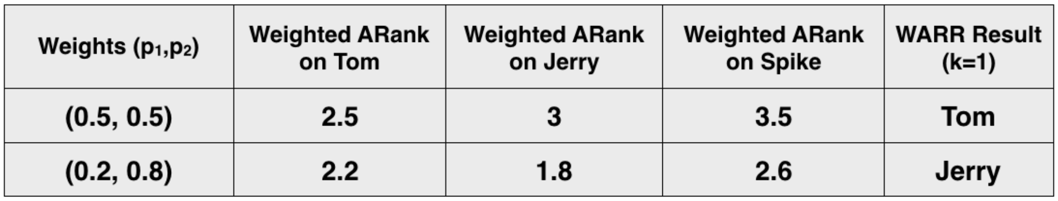
Top-k query is a user-view model that can help customers find their favorite products and is widely used in many practical applications [9, 43, 47]. For two different datasets, user preferences and products, this model retrieves the top-k products matching a given user preference. Conversely, manufacturers must also target potential customers for their products. Reverse k-rank query (*RkR*) [106], a manufacturer-view query model, solves this problem by returning the top-k user preferences for a given product. Another manufacturer-view query in Chapter 4, aggregate reverse rank query (*ARR*) addressed a limitation of *RkR* query, allowing it to retrieve user preferences for a set of products and help manufacturers with product bundling.

5.1.1 Motivation

ARR query is an essential tool for finding potential customers for a given product bundle. However, the aggregate rank function (*ARank*), which be used to evaluate the bundled products, is a simple sum operation ($ARank(w, Q) = \sum rank(w, q), q \in Q$). Although there also has MAX and MIN function that evaluating the maximum (minimum) ranking product in a product bundle, it is not enough to cover all situations of real-life. Therefore, it is a limitation of *ARR* query because these functions are only a part of scenario in which ranks are evaluated in real-world applications; neither customers nor manufacturers consider every product in the bundle to be equal in most situations. Many customers buy a product bundle because they want some subset of the products, and they may not care much about the others. Sellers also make use of this point, bundling unpopular products with attractive ones to maximize profits or liquidate inventories. For example, Xiaomi [1], a famous cell phone company in China, always bundles a newly released cell phone with some accessories (mobile battery, cell phones case, earphone, etc.); customers have to accept this product bundle if they want to obtain the new phone upon its release. Therefore, it is more accurate to add weights for different products in the bundle in the market analysis of interested consumers. In addition to e-commerce, user-product-based *ARR* queries can also be applied to other fields. *ARR* can used on NBA player statistics to analyze preferences for a given NBA team (of several players). In business investment, startup teams (of several members) would like to know which angel investor is most willing to invest in them. In these cases, adding weights to different roles on an NBA or startup team is also reasonable.

The model is closer to reality if weights are assigned for the *ARR* query. Inspired by this, we generalize the *ARR* query and then propose a new query problem called weighted aggregate reverse rank query (*WARR*). *WARR* query uses weights to handle the rank value of each product so that it can find customers using their different perspectives on products in the bundle. In this extension, the summing *ARank* function in the *ARR* query becomes a specific situation in which the weights

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



Weights (p₁,p₂)	Weighted ARank on Tom	Weighted ARank on Jerry	Weighted ARank on Spike	WARR Result (k=1)
(0.5, 0.5)	2.5	3	3.5	Tom
(0.2, 0.8)	2.2	1.8	2.6	Jerry

Figure 5.1: *WARR* query results for a bundle of p_1 and p_2 with different weights.

are equal to each other ($\alpha_i = \frac{1}{|\alpha|}, i = 1, 2, ..., |\alpha|$). More specifically, the weights $\alpha_i \in \alpha$ where $\sum_{i=1}^{|\alpha|} \alpha_i = 1$ correspond to the query products, note that this weighted preference is different from the user preference w on attributes. e.g.: α_1 is the weighted value for q_1 in Q. These weights can adjust the rank of query products with a weighted *ARank* function (*WARank*): $WARank(w, Q, \alpha) = \sum \alpha_i \times rank(w, q_i), q_i \in Q, \alpha_i \in \alpha$. Sellers and data analysts from the manufacturer can use *WARR* query to analyze various marketing positions by testing product bundles with different weights.

Based on the example of *ARR* query in Figure 4.1, Figure 5.1 shows a *WARR* query example for $\{p_1,\ p_2\}$ with two different weights: $\{0.5,\ 0.5\}$ and $\{0.2,\ 0.8\}$. As Figure 5.1(d) shows, Tom's weighted rank of $\{p_1,\ p_2\}$ is $3 \times 0.5 + 2 \times 0.5 = 2.5$ when $\alpha = \{0.5,\ 0.5\}$ and $3 \times 0.2 + 2 \times 0.8 = 2.2$ when $\alpha = \{0.2,\ 0.8\}$. As with the *ARR* query, Tom is also the result when the weights are equal. However, for weights $\{0.2,\ 0.8\}$, we want to find customers who consider p_2 to be the main reason to buy this bundle. In this case, Jerry's weighted aggregate rank value is $5 \times 0.2 + 1 \times 0.8 = 1.8$, which is the best rank among the three people; hence, *WARR* query returns Jerry as its result. In this query process, *WARR* helps manufacturers target Jerry, who treats p_2 as a main priority.

As a generalized version, *WARR* follows the previous work of *ARR*, which was a manufacturer-view query processing. Therefore, the weights for the query products in *WARR* query is assigned by the manufacturers. If we allow customers to set weights on the products, for each customer, we need to know her preferences for such a huge number of products and maintain them. Moreover, when a manufacturer releases a new product, it means a new vector be added to P in our model. However, it also means to add $|W|$ vectors to update the preference to all customers for allowing them to assign weights to products. In our model, it makes sense that let manufacturers assign weights to their products then issue a *WARR* query on these products. Specifically, to analysis the target customers with the variety considering on bundled products, a manufacturer can adjust different weights and issue different *WARR* queries, then *WARR* retrieves the target users under these weighted preferences.

Let us continue to use the example of Figure 4.1 in Chapter 4. As shown in Figure 5.1, assuming that the seller of the bookshop wants to use the book p_2 as the main product for sale, so she sets the weight as (0.2, 0.8) corresponding to $\{p_1,\ p_2\}$. Then *WARR* can help to return Jerry as the best target since his weighted aggregate rank is the best.

5.1.2 Challenges and contributions

Challenges. There are two challenges in efficient processing a *WARR* query.

The first challenge is to solve *WARR* queries with the *bound-and-filter* framework. *WARR* query is a complicate processing, it requires to evaluate the rank of each query products with respect to all users and returns the top-k users. The basic Naive algorithm for *WARR* leads a huge computation with a complexity of $O(|P| \cdot |W| \cdot |Q|)$. The key point is to reduce the pairwise computation between P and W. The most relevance work is the *ARR* query, which offers the solutions that bound query products Q with two dummy points and utilizes the spatial index R-tree to filter data with the divide and conquer methodology. We formalize the techniques as the *"bound-and-filter"* framework, it is an efficient strategy that reduces the computational complexity to $O(log|P| \cdot log|W|)$. *WARR* is a generation of the previous work *ARR* and has much more complicated processing. As the example in Figure 5.1 demonstrates, the value of the ranking changes with differing weights. Therefore, the previous techniques which was designed only for the products that are equal, cannot handle the weighted ranking relationship hence cannot be extended to solve *WARR* query effectively.

To this, we design sophisticate bounding methods and filtering strategies for the weighted ranking, and propose a solution EFM that based on the bound-and-filter framework (Section 5.2.2). Specifically, for the bounding phase, we propose weighted aggregate rank bounds to bound Q safely. We propose a novel early stopping strategy which takes into consideration the weights, and helps more efficient filtering.

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

To this, we proposed an optimal bounding method OBM (Section 5.2.3) which finds the optimal bounds for an arbitrary Q then filter more data than EFM. We proved the optimal bounding with the theory of linear programming. It is important to note that the proposed optimal bounds are effective also in previous *ARR* query but it helps to enhance the performance remarkable in *WARR*.

Contributions. This chapter makes the following contributions:

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

- We develop three solutions to process *WARR* queries, as existing approaches cannot be directly applied, called SFM, EFM and OBM. SFM is a straightforward method that uses a spatial R-tree. EFM adapts the bound-and-filter framework to the additional weights in the *WARR* query. We study this filtering space in the bound-and-filter framework and propose an optimal

Symbols	Description
P	set of products.
W	set of preferences.
Q	Query products.
α	Weights for Q.
d	Data dimensionality.
$f(w,p)$	The score of p based on w with inner product.
$p[i]$	Value of a product p in the ith dimension.
$H(w,q)$	The (d-1) dimensional hyper-plane perpendicular to w and cross q.
MBR	Minimum bounding rectangle.
e_p (e_w)	An MBR in Rtree of data set P (W).
$L[MBR]$, $U[MBR]$	The lower-left and upper-right corners in an MBR.
Q_l (Q_u)	The set of $q_l^{(i)}$ $(q_u^{(i)})$ in all d dimensions.
$Q.low$, $Q.up$	Bounding of Q in Chapter 4.
Q_{opt}^{low}, Q_{opt}^{up}	The Optimal bounding of Q.

Table 5.1: Symbols and Notation

bounding approach that is proven to find the tightest bound of Q. The OBM is based on this optimal bound in bound-and-filter framework. This optimal bounding strategy can also adjust into the previous *ARR* query.

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

5.1.3 Definitions

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

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

Definition 2. (*weighted aggregate reverse rank query, WARR*). *Given datasets P and W, positive integer k, and query product set Q, the WARR query returns set S, $S \subseteq W$, $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$,* WARank$(w_i,Q,\alpha) \leq$ WARank(w_j,Q,α) *holds.*

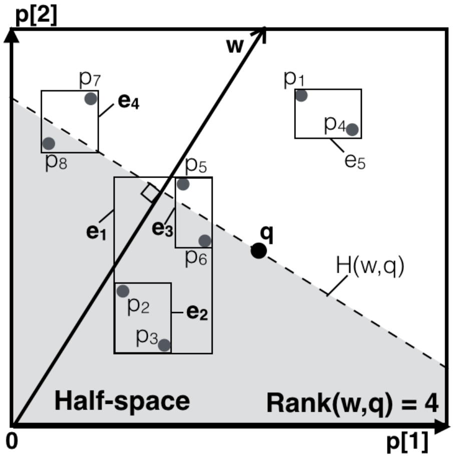


Figure 5.2: Geometric view of the rank of q and a tree-based methodology

5.2 Solutions

5.2.1 Straightforward Filtering Method (SFM)

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

The geometric view of a 2-dimensional example that ranks a single q based on preference vector w ($rank(w,q)$) is shown in Figure 5.2. There is a $(d-1)$ dimensional hyperplane denoted by $H(w,q)$, which is a line in the 2-dimensional example of Figure 5.2, that crosses q and perpendicular to w. The $rank(w,q)$ is equal to the number of p_i enclosed in the half-space (the gray area) defined by the hyperplane $H(w,q)$. Many previous works [85,86,106] have used a tree-base methodology to find the number of points in half-space efficiently; in particular, R-trees are used to index and filter dataset P. An R-tree can group nearby points with a minimum bounding rectangle (MBR) to filter data at the lower-left and upper-right borders. For example, the upper-right border of MBR e_2 is in the half-space. Therefore, all points contained by e_2 should be counted for the rank of q. On the contrary, p_1 and p_4 should be discarded, since the lower-left border of MBR e_5 is not in the half-space. The MBRs are checked recursively while traversing the R-tree. We propose a straightforward method, called SFM, based on this technique. As Algorithm 8 shows, SFM uses the tree-base method on P to process *WARR* straightforwardly.

Algorithm 8 Straightforward filtering method (SFM)

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

5.2.2 Extended Filtering Method (EFM)

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

Re-building the Bound-and-filter Framework in *WARR* query.

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

$$rank(L[e_w], Q.low) \cdot |Q| \leq ARank(w, Q) \leq rank(U[ew], Q.up) \cdot |Q|. \tag{5.1}$$

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

Obviously, this does not work for *WARR* rank query, which has a series of weights for Q. In order to bound the $WARank(w, Q, \alpha)$ for an arbitrary α, we multiply the left side of inequality (5.1) by the minimum value in α, denoted as α_{min}, and multiply the right side by the maximum value, α_{max}. This adaptation bounds the weighted aggregate rank as inequality (5.2) based on the following lemma:

$$\alpha_{min} \cdot rank(L[e_w], Q.low) \cdot |Q| \leq WARank(w, Q) \leq \alpha_{max} \cdot rank(U[e_w], Q.up) \cdot |Q|. \tag{5.2}$$

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

Proof. $\forall q_i \in Q, \forall w_i \in e_w$, it holds that $f(w_i, q_i) \geq f(L[e_w], Q.low)$; hence, $rank(w, q_i) \geq rank(L[e_w], Q.low)$.

Algorithm 9 Extended Filtering Method (EFM)

```
Initialize $buffer$ to store the first $k$ w's and the $WARank(w, Q, \alpha)$.
$Lrank \Leftarrow 0$
$minRank \Leftarrow$ the last rank in $buffer$.
Bounding phase: get $Q.low$ and $Q.up$
$heap_w.enqueue(R\text{-}tree_w.root)$
while $heap_w$ is not empty do
  $ew \Leftarrow heap_w.dequeue$
  $Cand \Leftarrow \emptyset$
  $heap_p.enqueue(R\text{-}tree_p.root)$
  while $heap_p$ is not empty do
    $e_p \Leftarrow heap_p.dequeue$
    if $e_p$ is located below the lower hyperplane then
      // Lemma 1.
      $Lrank \Leftarrow Lrank + e_p.size \cdot |Q| \cdot \alpha_{min}$
      if $Lrank \geq minRank$ then
        Continue
    if $e_p$ in sandwiched space then
      $Cand \Leftarrow Cand \cup e_p$
    if $e_p$ covers the upper or lower hyperplane then
      $heap_p.enqueue(e_p.children)$
  if $Lrank \leq minRank$ then
    // Corollary 3.
    Compute further bounds and update $Lrank$ and $Cand$.
    if $Lrank \leq minRank$ then
      if $ew$ is a single $w$ then
        // Lemma 2.
        Compute exact weighted rank $WARank$.
        if $WARank \leq minRank$ then
          Update $buffer$ and $minRank$.
      else
        $heap_w.enqueue(ew.children)$
return $buffer$
```

By definition, because α_{min} is the minimum value in α, $WARank(w, Q, \alpha) = \sum rank(w, q_i) \cdot \alpha_i \geq |Q| \cdot rank(L[e_w], Q.low) \cdot \alpha_{min}$. Similarly, $|Q| \cdot rank(U[e_w], Q.up) \cdot \alpha_{max}$ is the upper bound of $WARank(w, Q, \alpha)$. □

Early Stopping Strategy

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

If the lower bound of rank given by (5.2) cannot filter the e_w directly, it is necessary to check the weighted rank for all $w \in e_w$ and $q \in Q$. To reduce computations, we can reuse the value of

$rank(w, Q.low)$, which has been calculated in Lemma 1, to figure out the exact value of weighted rank. The correctness of the re-using is introduced and proved in the following lemma:

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

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

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

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

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

EFM Algorithm

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

5.2.3 Optimal Bounding Method (OBM)

The key point of efficiency in the bound-and-filter framework is the bounding phase, because the tightness of the bounds of Q determines the effectiveness of filtering both P and W. The score of the bounds $f(ew, Q.up)$ and $f(ew, Q.low)$ determine the amount of data in P that can be filtered. Moreover, as mentioned in Section 5.2.2, the higher the lower hyperplane $H(ew, Q.low)$ is located, the higher the value of the lower rank bound will be, and the more data from W will be filtered. If Q could be bounded more tightly, ew_2 might be filtered directly, without further computation. In conclusion, tightening the bounds of Q is significant to the performance.

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

The Optimal Bounds for Q.

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

$$w_a = \sum_{i=1}^{d} \gamma_i w_t^{(i)},\ where\ \gamma_i \geq 0\ and\ \sum_{i=1}^{d} \gamma_i = 1 \tag{5.3}$$

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

$$\begin{aligned} 1 = \sum_{j=1}^{d} w_a[j] &= \sum_{j=1}^{d} \sum_{i=1}^{d} \gamma_i w_t^{(i)}[j] \\ &= \sum_{i=1}^{d} \gamma_i \sum_{j=1}^{d} w_t^{(i)}[j] \\ &= \sum_{i=1}^{d} \gamma_i \times 1 \end{aligned} \tag{5.4}$$

Before formally stating and proving the d-dimensional case, we explain the lower bound optimum with 2-dimensional data. (The process for the upper bound can be illustrated in exactly the same way.) In Figure 5.3, there are top preferences $W_t = \{w_t^{(1)}, w_t^{(2)}\}$ of all dimensions, and two corresponding perpendicular lines (hyperplanes) of minimum values for $f(w_t^{(i)}, q_l^{(i)}), i = 1, 2, ..., d$ are determined.

The construction of the hyperplanes indicates that p has a lower score than $q_l^{(i)}$ on $w_t^{(i)}$ when p is located in the half-space below $H(w_t^{(i)}, q_l^{(i)})$. In Figure 5.3 where p is located in the overlap area (dark gray) of the two half-spaces, $p \cdot w_t^i \leq q_l^{(i)} \cdot w_t^{(i)}$ for $i = 1, 2, ..., d$. According to the Equation

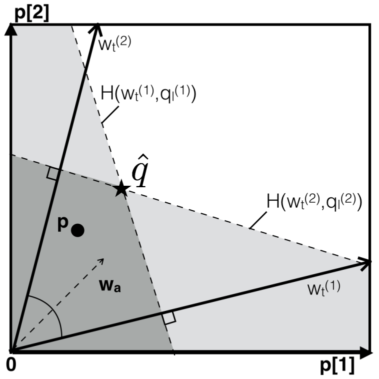


Figure 5.3: The half-spaces of $H(w_t^{(i)}, q_l^{(i)})$, $i = 1, 2, ..., d$. The intersection point is the optimal lower bound of Q for an arbitrary $w \in W$.

(5.3), the score of p based on an arbitrary w_a, $f(w_a, p) = p \cdot w_a$, can be presented as follows:

$$p \cdot w_a = p \cdot (\gamma_1 w_t^{(1)} + (1-\gamma_1) w_t^{(2)}) \leq \gamma_1 q_l^{(1)} \cdot w_t^{(1)} + (1-\gamma_1) q_l^{(2)} \cdot w_t^{(2)} \tag{5.5}$$

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

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

Replacing the *r.h.s* of Equation (5.5) with the *l.h.s* of $\gamma_1 \times (a) + (1-\gamma_1) \times (b)$ gives

$$p \cdot w_a \leq \gamma_1 \hat{q} \cdot w_t^{(1)} + (1-\gamma_1) \hat{q} \cdot w_t^{(2)} = \hat{q} \cdot w_a \tag{5.6}$$

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

Based on this discussion, we can lay out the formal conclusion that shows that the optimal bounds of Q are the intersection points. [2]

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

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

Proof of the Optimal Bounds (Theorem 1).

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

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

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

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

By the theory of linear programming, the optimal lower bound is the intersection point(s) of all hyperplanes $\{H(w_t^{(i)}, q_l^{(i)}) | i = 1, 2, ..., \mathrm{d}\}$. Generalizing the Equations (a) and (b) to d-dimensional case, the intersection point(s) $\hat{q}$ found by solving the following simultaneous equations (5.7), is the optimum solution of the above problem and hence the lower bound. In other words, like Equation (5.6), the score of a p under all the hyperplanes based on an arbitrary w_a satisfies $p \cdot w_a \leq \hat{q} \cdot w_a$.

$$A \cdot \hat{q} = c \tag{5.7}$$

where

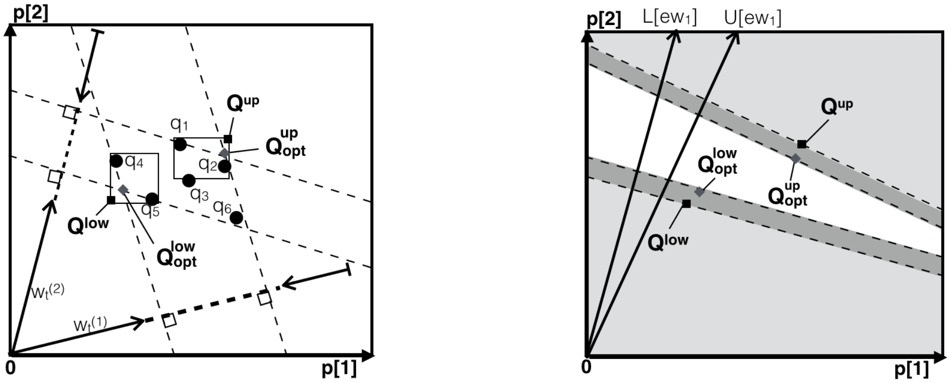
$$A = \begin{bmatrix} w_t^{(1)}[1] & \cdots & w_t^{(1)}[d] \\ \vdots & \ddots & \vdots \\ w_t^{(d)}[1] & \cdots & w_t^{(d)}[d] \end{bmatrix} \; and \; c = \begin{bmatrix} f(w_t^{(1)}, q_l^1) \\ \vdots \\ f(w_t^{(d)}, q_l^d) \end{bmatrix} \tag{5.8}$$

□

OBM Algorithm

Since computing Q_{opt}^{low} and Q_{opt}^{up} is equivalent to solving the linear equations of Equation (5.8) and finding $\hat{q}$, Gaussian elimination [3] is an easy and low-cost method for doing so. The total complexity of Gaussian elimination is approximate to $O(d^3)$, where d is the dimensionality of the data and indicates the number of linear equations. Therefore, the complexity of the bounding phase in OBM is $O(d \cdot |Q| + d^3)$. Notice that the complexity of finding $Q.up$ and $Q.low$ is $O(d \cdot |Q|)$ and the cube of dimensionality is still a very small value. The cost of bounding Q is negligible to the whole bound-and-filter algorithm, since both $|Q|$ and d are far smaller than the cardinality of W and P.

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



(a) The bounding phase.

(b) The filtering phase (gray area).

Figure 5.4: Bound-and-filter in OBM. (a) Finding the optimal bounds Q_{opt}^{low} and Q_{opt}^{up}. (b) Comparing the filtering space of the previous $Q.up(Q.low)$ and optimal bounds.

We take advantage of this optimal bound and propose the OBM method (Optimal Bound Method). The bounding phase of OBM is described in Algorithm 10. The filtering phase of OBM is to replace the $Q.low$ and $Q.up$ in EFM with the best bounds Q_{opt}^{low} and Q_{opt}^{up}. Furthermore, Lemmas 1 and 2 and Corolllary 3 also hold to the optimal bounds. Figure 5.4 shows the optimal bounds and filtering space of OBM. In Figure 5.4a, it is obvious that Q_{opt}^{low} and Q_{opt}^{up} bound Q more tightly than the previous $Q.low$ and $Q.up$. Figure 5.4b also shows that more data from P can be filtered than in EFM.

Algorithm 10 Optimal Bounding

```
$W_t$ has been found offline
$A_l$ and $A_u$ are matrixes for storing hyperplane equations.
for each $w_t^{(i)} \in W_t$ do
    $q_l^i \leftarrow argmax(f(w_t^{(i)}, q)),\ q \in Q$
    $Q_l \leftarrow Q_l \cup \{q_l^i\}$
    $A_l \leftarrow A_l \cup \{w_t^{(i)} \cup \{f(w_t^{(i)}, q_l^i)\}\}$
    $q_u^i \leftarrow argmin(f(w_t^{(i)}, q)),\ q \in Q$
    $Q_u \leftarrow Q_u \cup \{q_u^i\}$
    $A_u \leftarrow A_u \cup \{w_t^{(i)} \cup \{f(w_t^{(i)}, q_u^i)\}\}$
$Q_{opt}^{low} \leftarrow GuassianElimination(A_l)$
$Q_{opt}^{up} \leftarrow GuassianElimination(A_u)$
return $\{\ Q_{opt}^{low},\ Q_{opt}^{up}\}$
```

Table 5.2 summarizes the space and time complexities for NAIVE and the proposed SFM, EFM and OBM. NAIVE has the highest time complexity $O(|P| \cdot |W|)$ but no needs extra index storage. SFM uses R-tree to index P so it costs $O(|W| \cdot \log |P|)$. EFM and OBM are based on a bound-and-filter framework with two R-trees and have the complexities of $O(\log |W| \cdot \log |P|)$. The difference

Table 5.2: The complexities of the methods.

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

is that the optimal bounding strategy makes OBM better than EFM.

5.3 Experiments

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

5.3.1 Data sets and Metrics

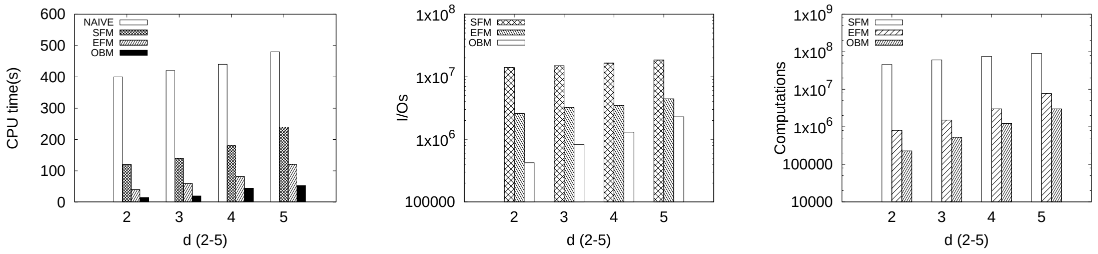
Product dataset P: We used both synthetic and real-world data for P:

- *Synthetic datasets*: The synthetic datasets are uniform (UN), clustered (CL), and anti-correlated (AC). The attribute value range of each dimension is [0,1). For the UN dataset, all attribute values are generated independently and following a uniform distribution. The AC dataset is generated by selecting a plane perpendicular to the diagonal of the data space using a normal distribution; we generate attributes value in this plane and follow a uniform distribution. For the CL dataset, first, the cluster centroids are selected randomly and follow a uniform distribution. Then, each attribute is generated with the normal distribution. We use the centroid values as the mean and 0.1 as variance. All of the above distributions were used in related work of other reverse rank queries [85, 86, 106].

- *Real-world datasets*: We also use two real data sets, NBA[4] and Amazon.[5] The NBA dataset contains 20,960 tuples of box scores of basketball players in NBA seasons from 1949 to 2009. We extracted 5-tuples to evaluate a player using points, rebounds, assists, blocks and steals statistics. The NBA dataset was also used in *ARR* query. Another real-world dataset is the metadata of products from Amazon.com, a well-known online retailer. This metadata contains 1,689,188 user reviews on 208,321 tuples of products in the categories of Movies and TV, in which product bundling is common. Each user provides at least five reviews, and each product is reviewed by at least five users. All the values were normalized based on the definition. We

[4] NBA: http://www.databasebasketball.com.

[5] Amazon: http://jmcauley.ucsd.edu/data/amazon/.



(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.5: Comparison results of varying d (2-5) on UN data P, W:UN, $|P|$ = 20K, $|W|$ = 200K, all with $|Q|$ = 5, k = 10.

extracted price and sales rank from the metadata as 2-dimensional vectors that represent a product. The Amazon data are also used in other research, such as [62, 63].

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

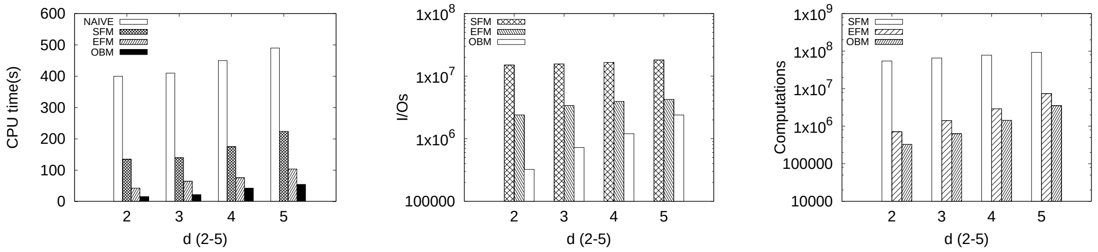
Query products Q: For the query products Q, we have two strategies to generate the queries. The first is to select a clustered subset from the product data P. In particular, we select a product in dataset P randomly, then find its m nearest neighbor in P, where m is the pre-defined cardinality of Q.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

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

Efficiency metrics: We use three metrics to observe the efficiency of all algorithms. a) The query execution time (CPU time) required by each algorithm; b) the I/O cost. I/O is estimated by checking accessed nodes in $R\text{-}tree_p$ and $R\text{-}tree_w$. We also observe c) the number of pairwise computations between P and W, which is a statistic that clearly shows the performance of each algorithm. We present average values over 1000 queries in all cases.

5.3.2 Experimental Results

Synthetic data: Figure 5.5 presents the comparative performance of all algorithms on UN data of both P and W for varying dimensionality d. The cardinality of $|P|$=20K, $|W|$=200K, k=10, and $|Q| = 5$. According to the execution time results shown in Figure 5.5a, our three proposed methods

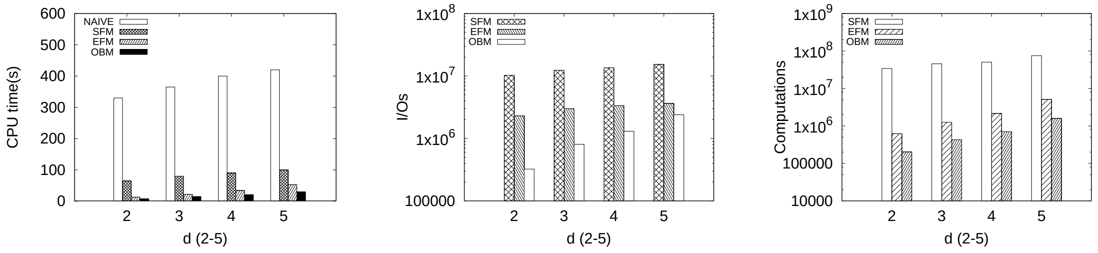


(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.6: Comparison results of varying d (2-5) on AC data P, W: UN, $|P|$ = 20K, $|W|$ = 200K, all with $|Q|$ = 5, k = 10.

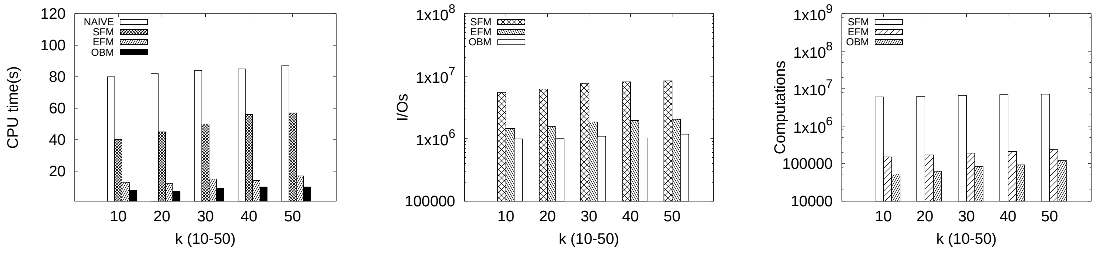


(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.7: Comparison results of varying d (2-5) on CL data P and W, $|P|$ = 20K, $|W|$ = 200K, all with $|Q|$ = 5, k = 10.

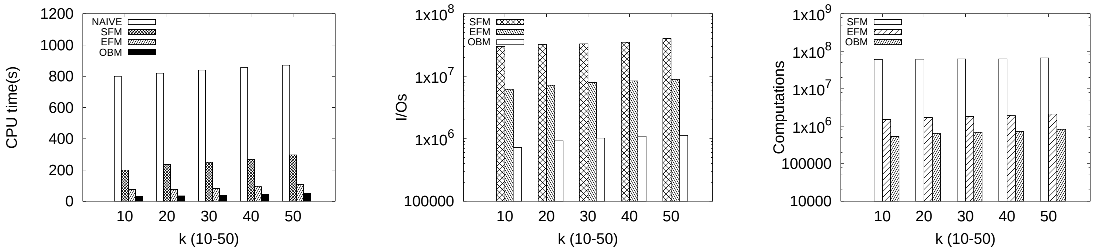


(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.8: Comparison results of varying k (10-50) on NBA data, $|P|$ = 20960, $|W|$: UN, $|W|$ = 100K, all with $|Q|$ = 5, d = 5.

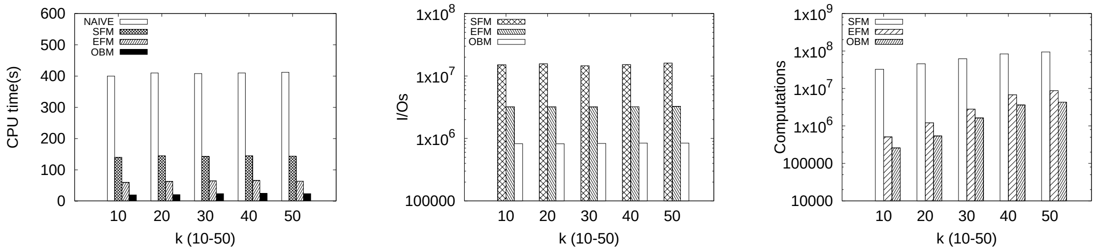


(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.9: Comparison results of varying k (10-50) on AMAZON data, $|P|$ = 208,321, $|W|$ = 1,689,188, all with $|Q| = 5$, $d = 2$.

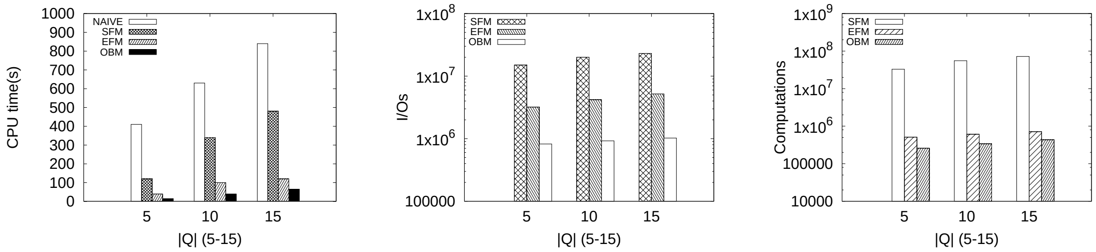


(a) CPU time.

(b) I/Os cost.

(c) Pairwise computations.

Figure 5.10: Comparison results of varying k (10-50) on UN data P and W, $|P| = 20$K, $|W| = 200$K, all with $|Q| = 5$, $d = 3$.

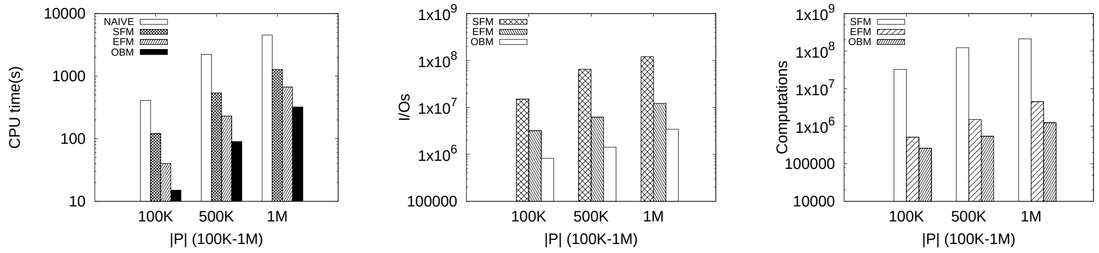


(a) CPU time.

(b) I/Os cost.

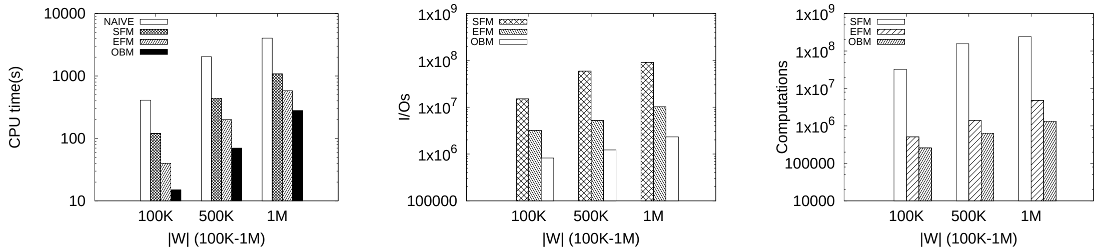
(c) Pairwise computations.

Figure 5.11: Comparison results of varying $|Q|$ (5-15) on UN data P and W, $|P| = 20$K, $|W| =$ 200K, all with $k = 10$, $d = 3$.



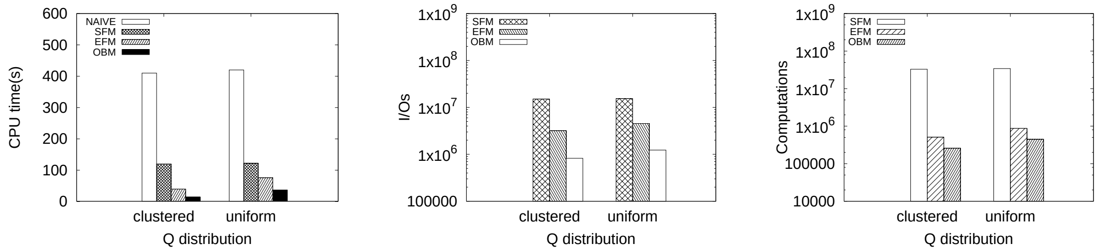
(a) CPU time.
(b) I/Os cost.
(c) Pairwise computations.

Figure 5.12: Scalability of varying P (100K-1M) on UN data P and W, $|P|$ =100K, all with $k = 10$, $d = 3$, $|Q| = 5$.



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

Figure 5.13: Scalability of varying W (100K-1M) on UN data P and W, $|W|$ = 100K, all with k = 10, $d = 3$, $|Q| = 5$.



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

Figure 5.14: Comparison results of different distribution on Q, $|P|$ = 20K, $|W|$ = 200K, all with k = 10, d = 3. $|Q|$ = 5

are significantly faster than the NAIVE algorithm. The EFM and OBM methods, which use the bound-and-filter framework with two R-tree, are superior to SFM because they avoid checking each $w \in W$. OBM is the most efficient, 2–3 times faster than EFM with the help of its optimal bounding strategy. We also found that the performance of SFM, EFM and OBM decrease as dimensionality increases; in higher dimensional space, query Q intersects more MBRs of the R-tree and the tree-based algorithms traverse deeper layers of the tree-structure for filtering data. Figure 5.5b shows the I/O cost of the proposed algorithms. EFM and OBM are better than SFM, since they only access a part of W with the $R\text{-}tree_w$ while SFM needs to check every w. OBM has a lower I/O cost than EFM due to the optimal bounds on Q in OBM, which allows it to filter more data than EFM does, as was proved in Theorem 1. The observation of pairwise computations is shown in Figure 5.5c, which is an insight view of all algorithms. OBM makes the fewest pairwise computations because it can filter the most data among all the algorithms; this also proves that OBM requires the least computation time when processing queries.

The comparison results of AC and CL data in the same setting as the UN experiment are shown in Figures 5.6 and 5.7, respectively. Similarly to the results of the UN data, OBM is the most efficient method; not only is it the fastest algorithm, but it also has the lowest I/O cost and number of pairwise computations. We found that the performance of EFM and OBM on CL data are better than on UN data, since the bound-and-filter framework can filter more MBRs in $R\text{-}tree_p$ and $R\text{-}tree_w$ when P and W are clustered.

Real-world NBA data . Figure 5.8 shows the CPU time and I/O cost for all algorithms on NBA data, with varying k. Clearly, OBM is more efficient than others and has lower I/O cost and fewer pairwise computations. The NBA data were also used in *ARR* query to answer the question: "Who loves a given basketball team more than other people do?". Every player on a basketball team has his responsibility; e.g., the Center and Power Forward defend and take rebounds, while the Point Guard and Score Guard need to pass and score. In addition to verifying *WARR* query's efficiency, we also tested its practical applicability. We set a query team that was good at defense, with (a) equal weights (*ARR* query) and (b) large weights on the Center and Power Forward (*WARR* query). The results from the *WARR* query all have greater preferences for the rebound and block attributes; this means that *WARR* query returns a correct set of people, those who prefer defensive teams. For the above observation, we conclude that *WARR* is more reasonable than the *ARR* query.

Real-world Amazon data. The Amazon dataset contains e-commerce data. Comparison results of CPU time, I/O cost and pairwise computing times are shown in Figure 5.9 with varying k on UN data from W. We randomly selected five movies or TV programs from the Amazon data as a product bundle query set. OBM maintains its efficiency in execution time and I/O cost, as can be seen in Figures 5.9a and 5.9b. This is a very strong result that demonstrates the efficiency of OBM in practical marketing applications.

Effect of varying k. Performance results when varying k on 3-dimensional UN data with

$|Q| = 5$, $|P|$=20K, $|W|$=200K, are shown in Figure 6.12. All algorithms are insensitive to k. First, k is far smaller than $|W|$ and $|P|$. Second, k is the number of results of w for *WARR* query, so the value of k does not affect performance very much for any algorithms, even though the NAIVE and SFM algorithms check all $w \in W$. EFM and OBM keep a k-element ascending *buffer* while processing, so they are only concerned with the last element with *minRank* rather than all k candidates in the *buffer*.

Effect on varying $|Q|$. For the varying $|Q|$ in Figure 5.11, because the number of products in a product bundle is not generally large, we test $|Q|$ from 5 to 15. EFM and OBM are insensitive to $|Q|$ based on these results because they bound Q in advance. However, the efficiency of the NAIVE and SFM algorithms decrease as $|Q|$ increases, as they check every q.

Scalability with varying $|P|$. Figure 5.12 shows the performance of all algorithms when increasing the cardinality of dataset P. We show the results of $|P| = 100K, 500K, 1M$, with $|W| = 100K$. The scalability of all algorithms are with respect to $|P|$, and OBM maintains the advantage over other algorithms. Because SFM, EFM, and OBM all use R-trees to index P, the execution time and data accessed grow faster than linearly with P. This is clearly shown in Figures 5.12a, 5.12b and 5.12c.

Scalability with varying $|W|$. We also test the scalability of all algorithms for varying $|W|$. We show the results of CPU time and I/O cost with the setting of $|W| = 100K, 500K, 1M$, with $|P| = 100K$, $d = 3$, $|Q| = 5$, and k = 5. These results differ with those of varying $|P|$ in Figure 5.13; in this case only EFM and OBM maintain their growth with increasing $|W|$. OBM is still the most efficient algorithm. Figure 5.13c gives the insight view of processing with the number of pairwise computations.

Effect on the distribution of Q. Figure 5.14 shows the comparison results on clustered Q and uniform Q. We can see that the performances in NAIVE and SFM are not changed with the clustered Q since they process queries in Q independently. On the other hand, the clustered Q has a better performance in the bound-and-filter based methods EFM and OBM. This is because the uniform Q may select a large distribution of products, then loose the bound of Q and compute more data than a tighter bound in a cluster.

5.3.3 Effectiveness

We test the effectiveness of *WARR* with AMAZON metadata and reviews data in Section 5.3.1, in comparison with previous *ARR*. The details are as follows:

- Product (P): Price, sales rank and rating are three attributes of a product. Price and sales rank are from the metadata which is also used in our experiments of performance comparison. The rating of a specific product is computed as the average value of the reviews on this product.

- User preference (W): The user preference is also a three-dimensional vector which has values

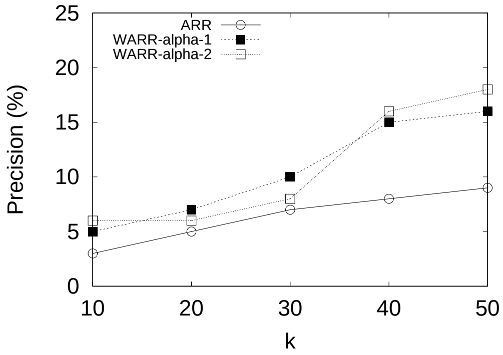


Figure 5.15: Precision of *ARR* and *WARR* on AMAZON data.

on (price, sales rank, rating), corresponding to the attributes of a product. For a user, the value of price and sales rank are computed as the average value of the products she has bought, and the value of the rating is the average value of her reviews.

- Query bundled products (Q): We first select a product p from P randomly, then find the "bought together" product of the selected p, and use these two products as a product bundle.

We issued two types of queries of *WARR* and *ARR* with 100 randomly selected Q, and recorded 50 results for each Q (i.e., $k = 10 \sim 50$). The precision is defined as the proportion of the users who have bought all products of Q. We set $\alpha_0 = (0.5, 0.5)$ as the *ARR* query which treats everything equally. On the other hand, we set $\alpha_1 = (0.75, 0.25)$ and $\alpha_2 = (0.25, 0.75)$ which means that either would be the appreciative one. Figure 5.15 reports the precision of *ARR* and *WARR* with α_1 and α_2. According to these results, *WARR*'s precision is better than *ARR* in all situations. Of course, the precision depends on α. Nevertheless, people always evaluate products unequally when they consider buying a bundled products, and *WARR* enables it to adjust the weights reflexing such request.

5.4 Summary

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

On the other hand, *WARR* is a general version of *ARR* query. In this Chapter, we adjust the bound-and-filter framework to solve *WARR* efficiently. As we mentioned in Section 5.1.2, the proposed methods of *ARR* query cannot be used for *WARR* directly since the effect of weights. In the future, we plan to study on adjusting the solutions with Grid-index and Cone$^+$ tree structures in Chapter 4 to *WARR* query.

Chapter 6

Continuous Spatial Keyword Search

As the popularity of SNS and GPS-equipped mobile devices increases, a large number of web users frequently change their location (spatial attribute) and interested keywords (keyword attribute) in real-time. An example of such would be when a user watches the news, videos and blogs while moving. Many location-based web applications can benefit from continuously searching for these dynamic *spatial keyword objects*. For example, a real-time coupon delivery system can search for potential customers by matching their locations and interested keywords. Then it sends coupons to attract these potential customers.

In this chapter, we define a novel query problem to continuously search for dynamic spatial keyword objects. To the best of our knowledge, this is the first work to consider dynamic spatial keyword objects. We employ a novel grid-based index to manage both queries and dynamic spatial keyword objects. With the proposed index, we devise a group-pruning technique to efficiently find the queries affected by the dynamic objects. Moreover, to quickly update the searching results for affected queries, we develop a buffer named *partial cell list* to reduce the computation cost in the top-k reevaluation. Theoretical analysis and experiments confirm the superiorities of our proposed techniques.

6.1 Introduction

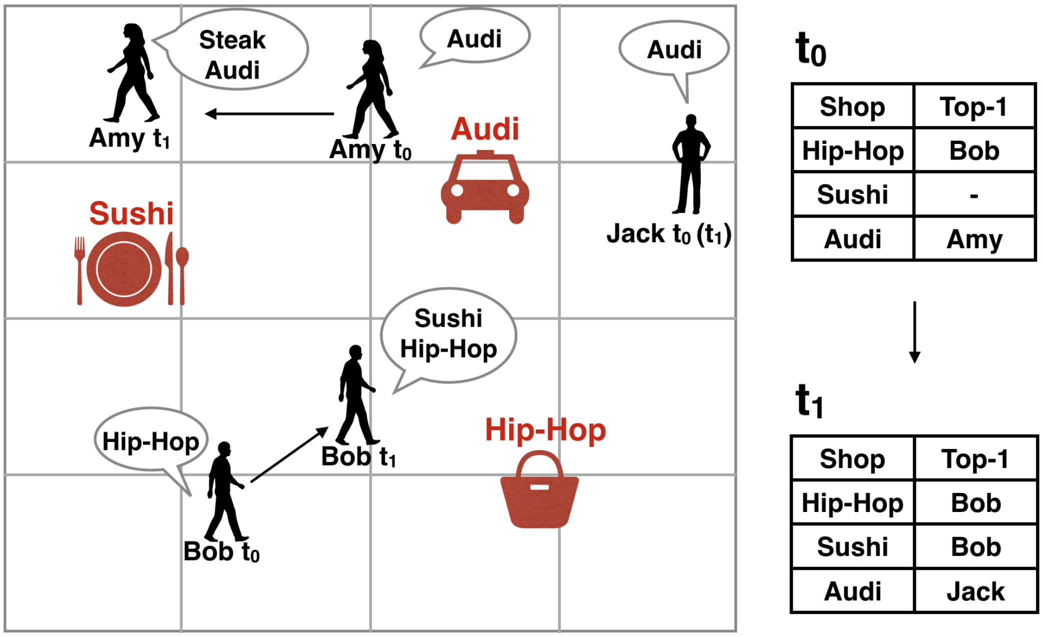
Nowadays, people are more likely to access the web from mobile devices such as a smartphone or a tablet than a desktop or a laptop computer. Smartphones and other mobile devices enable to receive information anywhere and enrich people's lives. People are used to watching the news, short video clips (e.g. Youtube, TikTok), posting on SNS (e.g. Twitter, Weibo) with smartphones while moving outside. The prevalence of GPS-equipped mobile devices generates massive volumes of *dynamic spatial keyword data* which is constantly updated on the web. In the real world, a spatial keyword object contains dynamic data of spatial and keyword attributes. For example, a person changes the content on his/her cellphone (keyword attribute) while moving around (spatial attribute). Many applications can benefit from the continuous searching of dynamic spatial keyword data. For instance, with the location-awareness recommendation system, a sports apparel shop can discover potential customers by continuously searching for nearby people who are watching sports related news and videos, or posting sports related topics on SNS. This research aims to define a novel searching problem that continuously searches for top-k dynamic spatial keyword objects.

6.1.1 Motivation

Although different types of continuous spatial keyword queries have been studied, the existing research considers only static objects. In other words, there is no research on dynamic spatial keyword problem, which is more realistic in the real world applications. Since an update of a dynamic object can be considered as a deletion of the previous status and an insertion of a new status, the key point is to consider the effect of deleting the previous status. After deleting a top-k member, we would refill the top-k list. Researchers have investigated streaming Publish/Subscribe systems, such as CIQ [14] and SKYPE [90]. It is noteworthy to mention that these studies differ from the problem in our research because they only consider the incremental static objects (i.e., the appended objects will not change their attributes.). For example, when searching for tweets, both CIQ and SKYPE search the top-k tweets dynamically in the sense that newly posted tweets are considered, however the tweet itself does not change. Unlike them, we search top-k users while the users are updated with their recent tweets, as well as their locations. Although an update operation can be implemented as a combination of a deletion operation and an insertion operation, existing research cannot process an unpredictable update of an object, which is a random insertion/deletion pair. Neither CIQ nor SKYPE support the above update operation because CIQ is an append-only system without deletions, and as a sliding window system, in SKYPE, insertions and deletions must be orderly w.r.t. the sequence of streaming data.

This gap in the existing research motivated us to study the realistic problem of continuous search on dynamic spatial keyword objects.

Figure 6.1 shows a scenario that explains how our research works with the application of the



t_0

Shop	Top-1
Hip-Hop	Bob
Sushi	-
Audi	Amy

t_1

Shop	Top-1
Hip-Hop	Bob
Sushi	Bob
Audi	Jack

Figure 6.1: E-coupon recommendation system.

E-coupon recommendation system. Assume that a Hip-Hop cloth store, a Sushi restaurant and an Audi car dealer are registered on our system. Our system continuously searches the top-1 people for these three shops and sends out coupons. At a certain time t_0, Bob is watching a hip-hop music video on his cellphone and becomes the top-1 result of the Hip-hop store. Amy searches “Audi car” with her phone, and Jack is watching a news about “Audi car”. Our system adds Amy to the top-1 for the Audi dealer since she is closer than Jack. Nobody is watching content about “Sushi”, so the top-1 for the Sushi restaurant is empty. At t_1, Bob has changed his location and has started to watch an eating show about Sushi. Then the keyword attribute of Bob changes to “Sushi, Hip-Hop” [1]. Our system still keeps Bob as the top-1 of the Hip-Hop shop since there are no other better options, and adds Bob to the top-1 list of the Sushi restaurant. Amy has left away from the Audi dealer and has searched “Steak near me” on her cellphone. Our system recognizes Amy’s change and has reevaluated the top-1 for Audi dealer, finding that Jack who has stayed and still watching the news about the ”Audi car” becomes the top-1.

6.1.2 Challenges and contributions

Challenges. There are two challenges in our research.

1. Indexing objects and queries. The first challenge is to design efficient indexing structures

[1] We suppose that the keyword attribute remains some keywords of the previous status.

to manage the big data of dynamic objects and queries. The most relevant works of index streaming spatial keyword objects are the Publish/Subscribe systems CIQ [14] and SKYPE [90] we mentioned before. First, we discuss the object index. CIQ and SKYPE used an IR-tree structure [24] to index the objects since they do not update the objects in the index. The IR-tree (Inverted File R-tree) has a similar structure to the R-tree, and each node links an inverted file to index the keyword information. Therefore, IR-tree suffers from the well-known limitation [49] of the R-tree where structure updates have an expensive cost. We cannot use the existing spatial keyword indexing technique for dynamic objects. Secondly, when a new status of a dynamic object is received, we need to find the affected queries promptly. Both CIQ and SKYPE use a quad-tree structure to index queries. CIQ sets a decay function according to the similarity score, so all queries must be indexed into every leaf-node. This feature loosens the spatial pruning and leads to an extremely high memory cost. SKYPE sets a non-decaying similarity function so that a single query is indexed into one leaf-node. However, it still requires overhead computing to target affected queries by traversing the tree. Consequently, it is a challenge to index queries and to retrieve them efficiently.

2. Top-k reevaluation. Another critical challenge is the top-k reevaluation, which occurs after a dynamic object is removed from the top-k list. It is time-consuming if the top-k is naively reevaluated from scratch. It is also infeasible to buffer all objects with their scores for each individual query to avoid the top-k reevaluation. The maintenance cost is extremely high for the dynamic objects, and the tremendous cardinality of the queries incurs expensive memory consumption. Therefore, related techniques have been proposed to balance the trade-off between the number of reevaluations and the buffer size. Yi et al. [102] introduced a $kmax$ buffer, which maintains top-k' results, and k' is a value between k and $kmax$. However, the $kmax$ solution just transforms the top-k maintenance to top-$kmax$ maintenance. The problem on how to reduce the number of evaluations remains unsolved. Later, Wang et al. [90] studied the top-k maintenance with spatial keyword objects and developed a cost-based k-skyband buffer with a proper threshold θ via theoretical analysis of the cost-model. Unfortunately, the above k-skyband framework is proposed for the streaming process with a sliding window. Specifically, the k-skyband buffer is constructed and maintained with the order of the incoming objects sequence, which is inapplicable to a scenario that searches for dynamic objects because the dynamic objects always change randomly and unpredictably. Hence, the design of an appropriate buffer with theoretical support to efficiently process the top-k reevaluation remains a serious challenge.

Many research on moving objects [66, 67, 97, 105] utilize a grid index to manage moving objects. Because they only consider the dynamics of spatial attribute (i.e., moving), they can take advantage of the priority features among cells and search the candidate objects efficiently. In Figure 6.2, to find the nearest object ($(k+1)$-th object) beyond the top-k, we can find the red cells first. Then if there exist at least one object that is not in the top-k, we just only check the blue cells and the green cells can be safely skipped. However, these priority constraints cannot work with the spatial

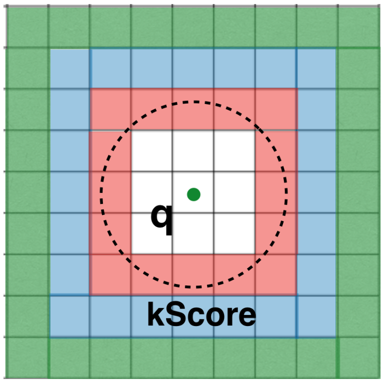


Figure 6.2: Priorities of cells with spatial-only similarity.

keyword similarity of our research. For example, there may exist an object in the green cells but it may have a higher relevance of keyword similarity, which makes it better than any objects in the red and blue cells. In conclusion, the technique of moving object cannot be used with the dynamic spatial keyword object.

Contributions. We define a new query process, which continuously searches the top-k dynamic spatial keyword objects. To overcome the first challenge of indexing, we design a novel grid-based index to manage both dynamic objects and static queries. The grid-based index can support rapid and economical updates of dynamic objects. In addition, queries are indexed with a sophisticated strategy of influential circles. Queries affected by a dynamic object can be quickly identified. For the second challenge of the top-k reevaluation, we propose a novel strategy that refills one candidate object rather than reevaluating the entire top-k list. To take advantage of the cells in the grid-based index, we design a buffer named PCL (partial cell list). PCL balances the trade-off between the search process and buffer maintenance to optimize efficiency. Our principal contributions:

- We formalize the continuous search problem on dynamic spatial keyword objects.
- We design a grid-based index to manage both objects and queries. We propose sophisticated strategies on affected queries finding and top-k refilling. We propose a buffer named PCL to refill top-k. PCL has a theoretical basis to maximize the efficiency. We also extend proposed solution to a batch process.

6.1.3 Definitions

In this section, we formally define the dynamic spatial keyword object, the spatial keyword query, the score function between them, and the problem of a continuous search on them. Table 6.1 summarizes the notations frequently used in this chapter.

Definition 1. *(**Dynamic Spatial Keyword Object,** o). A dynamic spatial keyword object o*

Symbols	Description
o, O	object, objects set
q, Q	query, query set
top-$k(q)$	k objects with the largest scores with q
$kScore(q)$	smallest score in the *top*-$k(q)$
o, o'	previous status and current status of a dynamic object
$grid$	grid-based index
c	a cell in the grid-based index.
n, n^2	partition number in a grid, cells number in a grid
$o.cell$	cell in a grid where object o is located
OutQ	q's which contain o in their top-k
InQ	q's satisfying $SimST(o', q) > kScore(q)$
$maxscore(c, q)$	maximum score of cell c and q
$minscore(c, q)$	minimum score of cell c and q
CL	sorted cell list
PCL	partial sorted cell list

Table 6.1: Symbols and Natation

is defined as $o = (o.\rho, o.\psi, t)$, where $o.\rho$ is the location attribute with coordinates, $o.\psi$ is a set of keywords, and t is the timestamp. Both $o.\rho$ and $o.\psi$ change over time. $o.\rho$ is updated with the up-to-date location. $o.\psi$ keeps the keywords of the up-to-date m status of object o, where m is a user-determined window size.

Since $o.\psi$ represents the interested keywords of object o, it is accord with the real application where we set $o.\psi$ as a window which keeps some previous keywords.

Definition 2. *(**Spatial Keyword Query,** q). Spatial keyword query q also has a location attribute and a keywords set of $q.\rho$ and $q.\psi$, In addition, $q.k$ is the number of results (the k in top-k). $q.\alpha$ is an user-defined smoothing parameter for spatial keyword similarities. The attributes of a query are static.*

We abbreviate dynamic spatial keyword object as *object* and spatial keyword query as *query.* Note that both location attribute and keyword attributes of an object are changing over time. Updating the conditions of a query is equivalent to deleting a previous query and adding a new one, so the queries are defined as static but we support incremental (decremental) queries in our problem. To evaluate the relevance of an object o to a query q, we define a score function as follows:

Definition 3. *(**Spatial Keyword Similarity,** $SimST$). Given object o and query q, the spatial* *keyword similarity between them is defined as:*

$$SimST(o,q) = q.\alpha \cdot SimS(o.\rho, q.\rho) + (1 - q.\alpha) \cdot SimT(o.\psi, q.\psi) \tag{6.1}$$

$SimST$ is a combined value of spatial similarity $SimS$ and keyword similarity $SimT$. Firstly, the $SimS$ is calculated by the normalized Euclidean similarity. Note that the $maxDist$ in the equation is the maximum distance in the data space.

$$SimS(o.\rho, q.\rho) = 1 - \frac{Euclidean(o.\rho, q.\rho)}{maxDist} \tag{6.2}$$

Secondly, $SimT$ is computed by the inner product between the tf-idf weights of $q.\psi$ and $o.\psi$.

$$SimT(o.\psi, q.\psi) = \sum_{w \in o.\psi \cap q.\psi} wt(o.w) \cdot wt(q.w) \tag{6.3}$$

where $wt(w)$ denotes the tf-idf weights vector of keyword w, and the weights of objects and queries are normalized to the unit length. The similarity functions are also used in the related work [90]. To ensure that both spatial and keywords are relevant, we require that every object in the top-k of a query must contain at least one common keyword with this query. Finally, we define the problem of the continuous search with above objects and queries.

Definition 4. *Given an object set O and a query set Q, for each query $q \in Q$ the continuous search is to keep the current top-k objects o's ($o's \in O$) ranked by the descending order of $SimST(o,q)$.* [2]

6.2 Proposed system

6.2.1 System overview

Figure 6.3 is a overview of how the proposed system solves the continuously search on dynamic objects. We assume that at the initial state, there already exists some objects and queries, as well as the top-k results. The new coming status of dynamic objects are handled sequentially while processing. When receiving a new status of an object, the grid-based index is updated first. Then the **affected queries finder** is executed to find queries affected by this object. Then, the top-k lists of these queries are updated. If a dynamic object causes the result to become short of k-elements, then the **top-k refiller** is triggered to refill the top-k list by checking it against a result buffer of candidates. Since data can be easily inserted and removed from the grid-based index, our system also supports incremental (decremental) objects and queries.

[2] When the similarities between two objects are equal, we assume that either of two objects can be correct for the result.

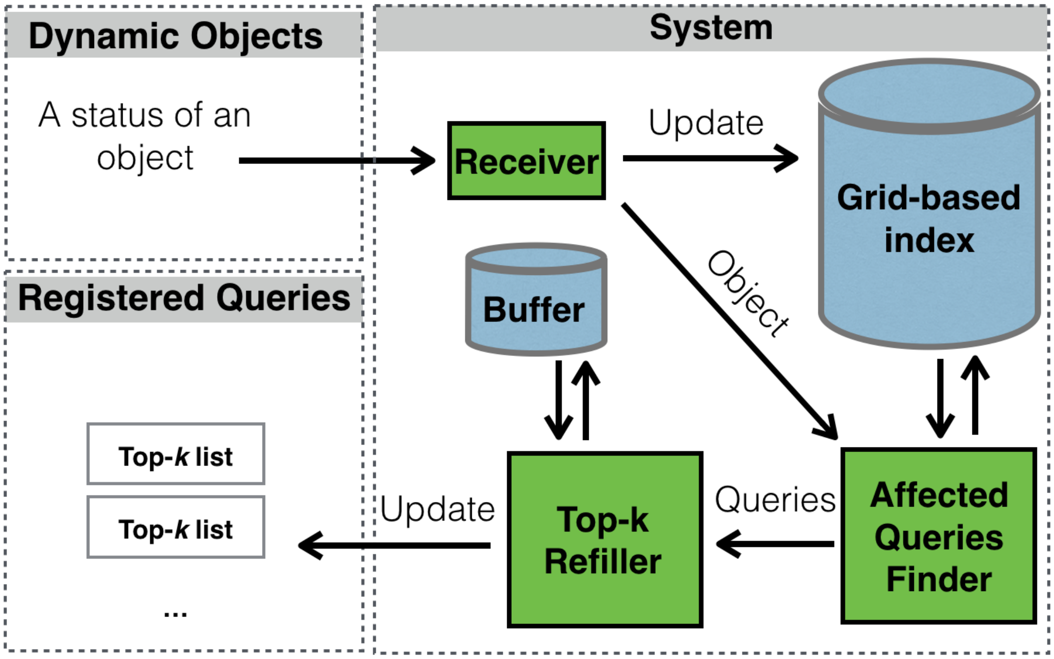


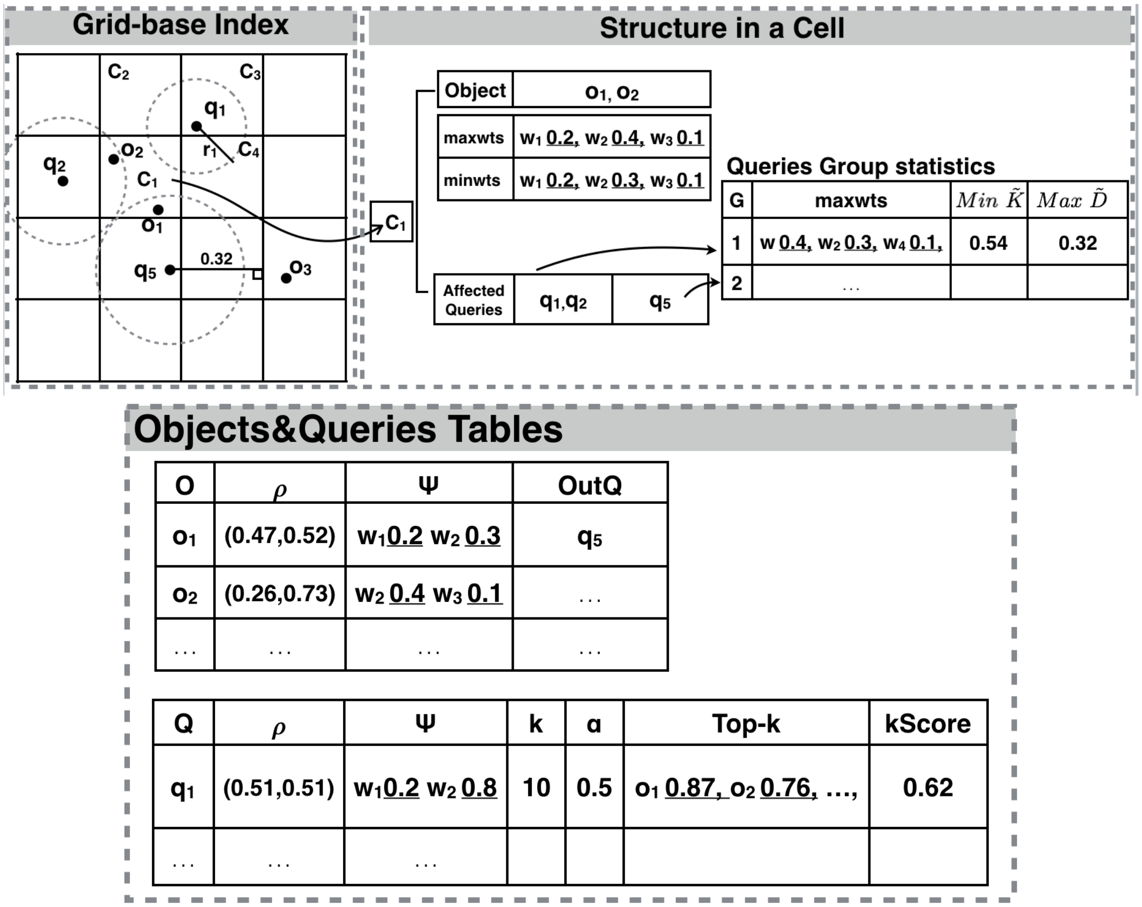
Figure 6.3: The system and flow of the process: *Grid-based index* (Section 6.2.2); *Affected queries finder* (Algorithm 11); *Top-k refiller* (Algorithms 12 and 15) and *PCL buffer* (Section 6.2.4).

6.2.2 Data structure: Grid-based index

To manage dynamic data, the index should respond quickly and have a low updating cost. Similar to the existing research [66, 67, 97, 105], we use a regular grid-based index to maintain both objects and queries (but they are not in a same indexing rule) because the data in the grid can be accessed and updated directly (o(1) complexity). Unlike some complicated indices (R-tree, IR-tree, etc.) that require extra and expensive costs to maintain the structure, the grid-based index is better suited for frequently updated applications.

Assume that there is a two-dimensional spatial space, the lengths of the x-axis (longitude) and y-axis (altitude) are both fixed and denoted by $maxDist$. The grid equally divides the x-axis and y-axis into n partitions and contains n^2 cells. We use $c_{i,j}$ to denote the cell in ith partition on the x-ais and the jth partition on the y-ais. Therefore, given an object o, if its spatial coordinate $o.\rho[0]$ is in the range $[i \cdot \frac{maxDist}{n},\ (i+1) \cdot \frac{maxDist}{n}]$ and $o.\rho[1]$ is in the range $[j \cdot \frac{maxDist}{n},\ (j+1) \cdot \frac{maxDist}{n}]$, object o is located in cell $c_{i,j}$.

Objects are indexed in the grid w.r.t their covering cells. On the other hand, to find the affected queries efficiently for the new status of an object, the queries are indexed into the grid according to their “influential circles”. We use $q.\rho$ (the location attribute) as the center point and create an



Object	o1, o2
maxwts	w1 0.2, w2 0.4, w3 0.1
minwts	w1 0.2, w2 0.3, w3 0.1

Affected Queries	q1,q2	q5

G	maxwts	$Min\ \tilde{K}$	$Max\ \tilde{D}$
1	w 0.4, w2 0.3, w4 0.1,	0.54	0.32
2	…		

O	ρ	Ψ	OutQ
o1	(0.47,0.52)	w1 0.2 w2 0.3	q5
o2	(0.26,0.73)	w2 0.4 w3 0.1	…
…	…	…	…

Q	ρ	Ψ	k	α	Top-k	kScore
q1	(0.51,0.51)	w1 0.2 w2 0.8	10	0.5	o1 0.87, o2 0.76, …,	0.62
…	…	…				

Figure 6.4: Grid-based index, inner structure of a cell, and tables

influential circle with a radius r calculated by:

$$r = \frac{1 - kScore(q)}{q.\alpha} \cdot maxDist \tag{6.4}$$

where $maxDist$ is the maximum distance in the space and $kScore(q)$ is the smallest score in the top-k list of q. If an object is located outside of this influential circle, it will not be an element in the top-k of q. We indexed q into the cells that overlap q's influential circle. Figure 6.4 shows an example where the circle of q_1 overlaps with c_1, c_2, c_3, and c_4. Hence, q_1 is indexed into these four cells.

A cell can bound the range of the spatial attribute for the objects. For the keyword attribute, maximum weights (*maxwts*) and minimum weights (*minwts*) of the objects are also indexed. The *maxwts* (*minwts*) can bound keyword similarities between a query and a cell; the details are in Section 6.2.3 and Section 6.2.4.

Example 1. *In Figure 6.4, cell* $c1$ *contains objects* o_1 *and* o_2*. According to the information of* o_1 *and* o_2 *in the Object table,* $c1.maxwts = \{0.2, 0.4, 0.1\}$ *and* $c1.minwts = \{0.2, 0.3, 0.1\}$*, corresponding to the keywords* w_1, w_2 *and* w_3*.*

In summary, a grid cell contains four kinds of information: objects, *maxwts*, *minwts* and affected queries. Note that the objects are indexed into a sequential list, but the affected queries are indexed into a sophisticated group-based structure. This arrangement supports efficient group-pruning technique used in our *Affected Queries Finder* module. The information and useful statistics of objects and queries are stored in two different tables, we only index the object id and query id into the grid-based index. Since our system is an in-memory system, all other information can be retrieved from the tables via the random access.

6.2.3 Affected queries finder

In this section, we introduce the *Affected Queries Finder* (AQF) module. When the system receives a new status of a dynamic object, AQF helps to find the affected queries, i.e., the queries whose top-k needs to updated. We use o and o' to represent the previous status and the current status, respectively. Updating from o to o' affects two kinds of queries: OutQ and InQ. OutQ is a set of queries corresponding to the previous o, each query in OutQ contains o in its top-k. (i.e., the dynamic o' may reduce OutQ's top-k lists to less than k objects). On the other hand, when object o changes to o', InQ collects queries corresponding to the current o' and $SimST(o', q)$ becomes larger than their $kScore$. (i.e., the moving $o \rightarrow o'$ makes InQ's top-k lists updated with o').

$$InQ = \{q \mid q \in Q \wedge SimST(o', q) > kScore(q)\} \tag{6.5}$$

Usually, OutQ is initialized when the system starts, and is updated in the object table when a process for a dynamic object is completed. Therefore, we can retrieve OutQ easily by looking up the object table, and the problem of identifying affected queries becomes how to efficiently find InQ. In the rest of this section, we introduce the techniques of finding InQ.

Influential circle pruning. If dynamic objects are locate much too far from a query q, then q can be safely pruned. This is the reason that we index a query into the overlapping cells w.r.t its influential circle, and only the indexed queries in the cell $o'.cell$ need to be considered.

Group query pruning. To avoid comparing the indexed queries with o' one by one, for each cell, we divide the indexed queries into several groups. Then a threshold for each group is generated to prune multiple queries simultaneously.

Single threshold. Assuming that an object moves to a cell (denote as $o'.cell$), then for an arbitrary query q, the upper bound of the spatial similarity to q ($SimSUB(o'.cell, q)$) can be estimated based on Equation (6.2):

$$SimSUB(o'.cell, q) = 1 - \frac{minD(o'.cell, q.\rho)}{maxDist} \tag{6.6}$$

where $minD(o'.cell, q)$ is the minimum Euclidean distance between q and $o'.cell$ (e.g. in Figure 6.4, $minD(o_3.cell, q_5) = 0.32$. Note that if q is covered by $o'.cell$ then $minD(o'.cell, q) = 0$). We can infer a keyword similarity threshold $T_\theta(o'.cell, q)$ with $SimSUB(o'.cell, q)$ and Equation (6.1),

$$T_\theta(o'.cell, q) = \frac{kScore(q)}{1 - q.\alpha} - \frac{q.\alpha}{1 - q.\alpha} \cdot SimSUB(o'.cell, q) \tag{6.7}$$

Therefore, for the current status o', a query can be pruned by comparing the keyword similarity to the threshold instead of calculating the whole spatial keyword score.

Group threshold. According to Equation (6.7), the calculation of a threshold for a single query can be divided into two parts: $\frac{kScore(q)}{1-q.\alpha}$ and $\frac{q.\alpha}{1-q.\alpha} \cdot SimSUB(o'.cell, q)$. For simplicity, we denote $\frac{kScore(q)}{1-q.\alpha}$ as $\tilde{K}$ and $\frac{q.\alpha}{1-q.\alpha} \cdot SimSUB(o'.cell, q)$ as $\tilde{D}$ respectively. Then the lower bound is derived as the group threshold as:

$$T_\theta(o'.cell, g) = Min_{q \in g}\{\tilde{K}\} - Max_{q \in g}\{\tilde{D}\} \tag{6.8}$$

$\tilde{D}$-based partition. To generate a tighter group threshold, we should group queries with similar $T_\theta(o'.cell, q)$. The value of $\tilde{D}$ is irrelevant for the spatial attributes of objects. Moreover, $\tilde{D}$ is a static value because queries will not change their location and α, but $\tilde{K}$ will change as the top-k list is updated. By intuition, we group the indexed queries in a cell by their $\tilde{D}$ value, such that the queries inside a group have similar $\tilde{D}$'s. We employ a quantile-based method to partition the domain of $\tilde{D}$ to ensure the tight grouping. Figure 6.4 also gives the image of a query group in a cell.

Maximum keyword similarity. On the other hand, the maximum keyword similarity between o' and g, which is denoted as $maxT(g, o')$, can be estimated as:

$$maxT(g, o') = SimT(g.maxwts, o'.\psi) \tag{6.9}$$

In Equation (6.9) *g.maxwts* denotes the maximum weights of all keywords contained by the queries in g. (The similar mechanism exists for *maxwts* of all objects in a cell.) As shown in Figure 6.4, the information of *maxwts* is kept for each query group in our index.

Finally, we can use Theorem 1 to prune a group of queries safely. Algorithm 11 concludes our processing of AQF. In Algorithm 11, we first check the incoming o' to prune query groups simultaneously using Theorem 1 (Line 4). Then we check the queries one by one if they can not pass the group pruning (Lines 4-7). The complexity of condition checking is $O(1)$, since we can precompute the values of $\tilde{K}$ and $\tilde{D}$.

Algorithm 11 AQF (Affected Queries Finder)

```
Input: $o'$
Check object table with $o'$ and get OutQ.
InQ= $\emptyset$
for each group $g \in G$ of $o'.cell$ do
    if $maxT(g, o') \leq g.T_\theta$ then
        for each query $q \in g$ do
            if $SimST(o', q) > kScore(q)$ then
                InQ= InQ$\cup\{q\}$
return OutQ, InQ
```

Theorem 1. *For a group of queries g, and a dynamic object* o'*, all queries in g will not be present in InQ of object* o' *if* $maxT(g, o') < T_\theta(o'.cell, g)$*.*

6.2.4 Top-k refiller

When a dynamic object o changes to o', after obtaining the affected queries, i.e., OutQ and InQ, from the *Affected Queries Finder*, the following step is to update their top-k lists. Updating the top-k's for queries of InQ is a low-cost since only the top-k list is considered with the o'. In other words, we just need to insert o' into the previous top-k. However, for the queries of OutQ, o' may get out of the top-k list. In this case, we must reevaluate the top-k result from all objects. It is natural that the cardinality of the objects is always much larger than k. Consequently, compared to the process of InQ, the main task is to reevaluate the OutQ's top-k lists efficiently.

In this subsection, we first describe a straightforward solution (GCL) that maintains a sorted cell list (CL) to retrieve the latest top-k result efficiently. We also propose a novel solution (GPCL) with a partial cell list (PCL). Different from CL, GPCL can avoid the redundant maintenance and computations by maintaining only a few cells and refilling the candidate objects to the previous top-k list.

GCL method with the sorted cell list

The related research such as *kmax* [102] and SKYPE [90] focus on maintaining the candidate objects in a result buffer to support the top-k reevaluation. However, keeping objects in the buffer is inefficient in our problem because the objects are dynamic, that incurring frequent and expensive buffer maintenance. To address this limitation, we propose a sorted cell list (CL) buffer that maintains all cells of our grid-based index w.r.t a similarity priority. Comparing to maintain objects, to maintain cells have advantages that: (a) the number of cells is much smaller than the number of objects; (b) cells have fixed spatial similarity bounds and infrequent changing keyword similarity bounds (only changed when an object affects the *maxwts* or *minwts* of a specific cell).

Algorithm 12 GCL

```
Input: $o$, $o'$, $OutQ$, $InQ$
1: Update $q.CL$ w.r.t $o.cell$, $o'.cell$
2: for each $q \in \{InQ\}$ do
3:     $top\text{-}k(q)$.insert($o'$)
4: for each $q \in \{OutQ - InQ\}$ do
5:     $top\text{-}k(q)$ = Branch-and-bound searching with $q.CL$
```

We use $q.CL$ to denote the sorted cell list w.r.t the query q. Figure 6.5 gives an example of CL. All cells in the grid-based index are sorted by their *maxscore* and stored in CL, where $maxscore(c, q)$ is the upper value of the spatial keyword similarity between any object in cell c and a query q:

$$maxscore(c, q) = q.\alpha \cdot SimSUB(c, q.\rho) + (1 - q.\alpha) \cdot SimT(c.maxwts, q.\psi) \tag{6.10}$$

The $maxscore(c, q)$ is the upper bound, because $SimSUB(c, q.\rho)$ dominates the spatial similarity and $SimT(c.maxwts, q.\psi)$ dominates the inner product value of the keyword similarity.

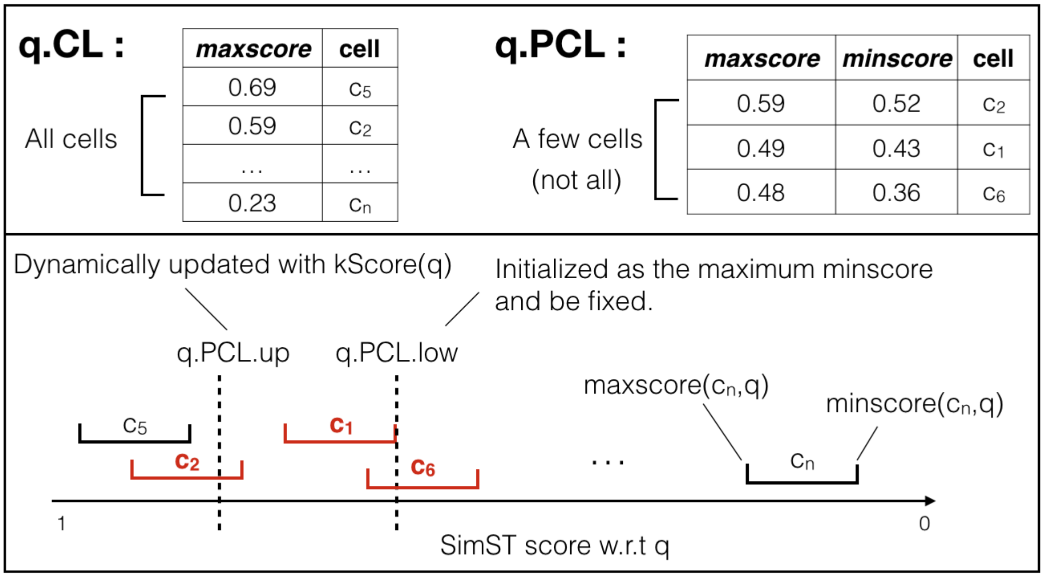
Our idea is that we can employ an efficient branch-and-bound method to find the top-k objects from the cells in CL. Since CL can be implemented as a binary-tree-like structure [3]. Algorithm 12 shows the top-k reevaluation method with CL (GCL). In Algorithm 12, CL should be updated w.r.t the cells covering the dynamic object (Line 1) before updating the top-k lists according to *InQ* (Lines 2-3) and *OutQ* (Lines 4-5).

GPCL method with the partial sorted cell list

Including the above GCL method, previous works always focus on the top-k reevaluations [66, 90]. However, these techniques contain redundant computations because they ignore the existing order and recreate a new top-k list again and again. Actually, when an object changs, only one ranking position changes, and the relative order of the other objects remains. For example, suppose that there are five objects, if the 3rd becomes the 5th, then the ranking relationship of the current 1st, 2nd, 3rd, and 4th objects do not change. Therefore, if an object changes out of the top-k, the correct top-k list can be obtained by only refilling the candidate $(k + 1)$-th object. Another limitation of GCL is that CL must maintain itself every time even though an object does not affect any queries.

Motivated by the above limitations, we propose a partial sorted cell list (PCL), which is a subset of CL. PCL always keeps the candidate $(k + 1)$-th object to refill the top-k list. We use both *maxscore* in Equation (6.10) and *minscore* to index the cells in PCL. The $minscore(c, q)$ denotes the minimum spatial keyword score between any objects in cell c and a query q. *minscore* is formed

[3] Our implementation uses std::map in C++, it is a red-black tree structure.



q.CL :

maxscore	cell
0.69	c5
0.59	c2
…	…
0.23	cn

q.PCL :

maxscore	minscore	cell
0.59	0.52	c2
0.49	0.43	c1
0.48	0.36	c6

Figure 6.5: Examples of CL (Section 6.2.4) and PCL (Section 6.2.4) buffers.

with the minimum spatial similarity ($SimSLB$) and the minimum keyword similarity ($SimTLB$).

$$minscore(c, q) = \alpha \cdot SimSLB(c, q.\rho) + (1 - \alpha) \cdot SimTLB(c.minwts, q.\rho) \tag{6.11}$$

$SimSLB$ is calculated similarly to Equation (6.6), where $maxD(.)$ represents the maximum Euclidean distance.

$$SimSLB(c, q) = 1 - \frac{maxD(c, q.\rho)}{MaxDist} \tag{6.12}$$

and $SimTLB$ is:

$$SimTLB(c.minwts, q.\rho) = MIN_{w \in q.\psi \cap c.minwts} wt(q.w) \cdot wt(c.w) \tag{6.13}$$

Partial sorted cell list (PCL)

Definition 5. *(*Partial Cell List, PCL*). Given a query q and a grid-based index. For each cell $c \in q.PCL$, it holds that c contains at least one object, and $minscore(c, q) < q.PCL.up$ and $maxscore(c, q) > q.PCL.low$. While processing, $q.PCL.up$ can be updated as the up-to-date $kScore(q)$, $q.PCL.low$ is initialized as the value of $maxMinS$ and will not change until $q.PCL$ is recreated, where $maxMinS = MAX\{minscore(c_j, q)\}$, $c_j \in grid.cells$ and $maxscore(c_j, q)$ $< kScore(q)$.*

Theorem 1. *After initialization, $q.PCL$ contains the (k+1)-th object w.r.t q.*

Proof. By contradiction. Assume that the (k+1)-th object o_{k+1} is not in the cells of PCL, we

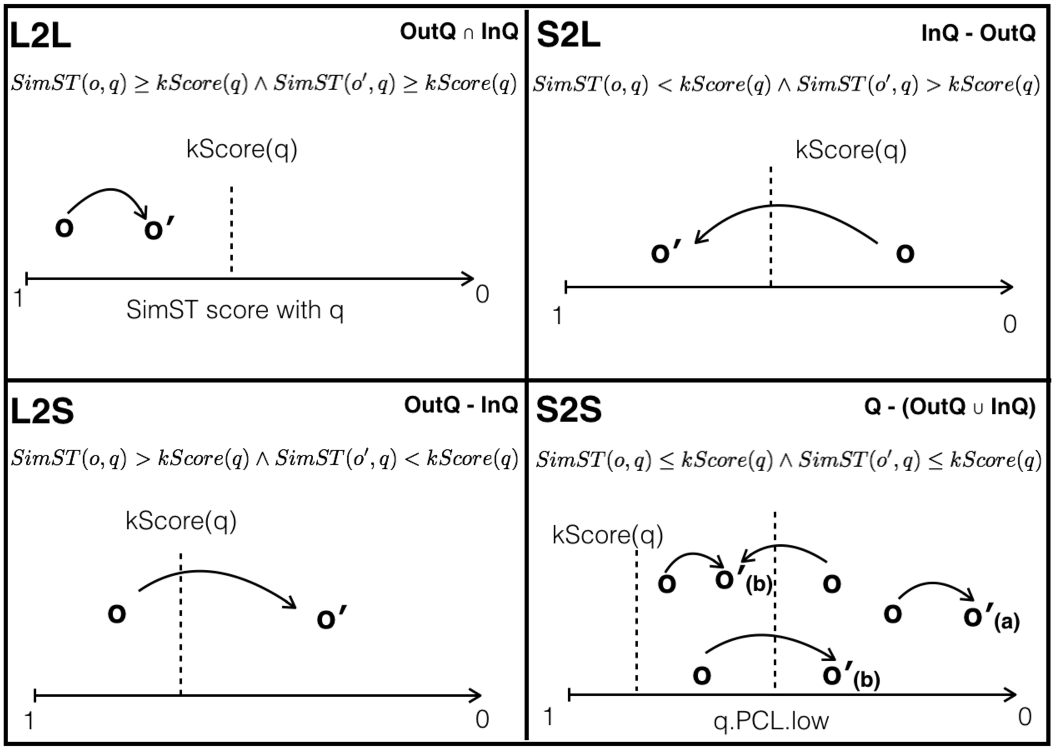

L2L
OutQ ∩ InQ
$SimST(o,q) \geq kScore(q) \wedge SimST(o',q) \geq kScore(q)$
kScore(q)
o
o′
1
0
SimST score with q
S2L
InQ - OutQ
$SimST(o,q) < kScore(q) \wedge SimST(o',q) > kScore(q)$
kScore(q)
o′
o
1
0
L2S
OutQ - InQ
$SimST(o,q) > kScore(q) \wedge SimST(o',q) < kScore(q)$
kScore(q)
o
o′
1
0
S2S
Q - (OutQ ∪ InQ)
$SimST(o,q) \leq kScore(q) \wedge SimST(o',q) \leq kScore(q)$
kScore(q)
o
o′(b)
o
o
o′(a)
o
o′(b)
1
q.PCL.low
0

Figure 6.6: Four cases of a dynamic object and a query.

can know that $SimST(o_{k+1}, q) < q.PCL.low$. According to the features of PCL, at least one object o_i whose score is greater than $q.PCL.low$ must exist. Therefore $SimST(o_i, q) > q.PCL.low$ $> SimST(o_{k+1}, q)$, and o_i should be the (k+1)-th object, leading to the contradiction. □

Example 2. *Figure 6.5 gives an illustration of CL and PCL buffers. In this Figure, the horizontal axis shows the score distribution of all cells (c_1, c_2, ..., c_n) w.r.t a specific q. A segment represents the score range of a cell. The left side is the maxscore while the right side is the minscore. CL is a cell list sorted by maxscore. Therefore. the order of $q.CL$ in this example is: $\{c_5, c_2, c_1, c_6, ..., c_n\}$. For PCL, $q.PCL.up$ is initialized by the current $kScore(q)$. On the right side of $kScore(q)$, cell c_1 has the maximum minscore. Thus, $q.PCL.low$ will be initialized as $minscore(c_1, q)$. Then $q.PCL$ contains the $\{c_2, c_1, c_6\}$ overlapping the range $(q.PCL.up, q.PCL.low)$.*

For each query, the corresponding PCL is initialized based on Definition 5. When a top-k list needs to be reevaluated, the candidate object can be searched from PCL and refilled to this top-k list. The top-k reevaluation is much more efficient than using the CL because the only top-1 search is conducted from fewer cells. To guarantee that all PCLs always contain the candidate object w.r.t the corresponding queries, we propose a sophisticated strategy to maintain PCL.

PCL Maintenance

We divide the situations between a dynamic object and a query into the following four cases. Figure 6.6 shows images and summarizes the four cases: **L2L** (large to large), **S2L** (small to large), **L2S** (large to small) and **S2S** (small to small). Because a dynamic object may affect the cells in PCL, we should maintain PCL carefully to ensure it always contains the (k+1)-th object to refill. We propose a sophisticated maintenance process. Table 6.2 summarizes the operations of these cases.

L2L and S2L. The cases of L2L and S2L are discussed together since they have a common PCL maintenance. In the cases of L2L, both o and o' are in the top-k. Thus, the order of the objects outside top-k is not changed, and the ($k+1$)-th object remains in PCL. Therefore, PCL still contains the candidate object. We just *check* the changing cells $o.cell$ and $o'.cell$ with PCL. Algorithm 13 describes the procedure of *check*. In Algorithm 13, each input cell, according to their new *maxscore* and *minscore*, will be deleted (Lines 2-3) and re-inserted (Lines 4-5) into PCL. For the case of S2L, o' is added into top-k from the outside. The previous kth object (denoted as o_k) will become the $(k + 1)$ one, so we should add the $o_k.cell$ into PCL to ensure the candidate object. Since $o.cell$ and $o'.cell$ also require a *check*, we need *check* $\{o.cell, o'.cell, o_k.cell\}$. In conclusion, we employ a *check* operation to maintain PCL in the cases of L2L and S2L.

Lemma 1. *Given a dynamic object and a query. If they are in the cases L2L and S2L, we need to maintain PCL with the operation of check.*

L2S. In the situation of L2S, we must *search* the top-1 candidate object o_{cand} from PCL. Then the scores $SimST(o', q)$ with $SimST(o_{cand}, q)$ are compared to determine whether to refill o_{cand} or not. Then, we should *trim* the PCL with a new score range $(kScore(q)', q.PCL.low)$ where $kScore(q)'$ denotes the updated k-th score. Algorithm 14 gives the details of the operation *trim*, which filters the cells that are not within the input score range (Lines 2-4). Sometimes PCL may become empty (no object) after *trim*. Then, PCL must be recreated.

Lemma 2. *Given a dynamic object and a query of case L2S, we need to maintain PCL with the operations of check and trim. Also, if PCL has empty candidates after trimming in case L2S, PCL must be recreated.*

S2S. S2S is divided into two sub-cases: S2S.a and S2S.b. In S2S.a, both o and o' are on the outside of $q.PCL.low$, so PCL does not need to be maintained. In S2S.b, we need to *check* $o.cell$ and $o'.cell$ with PCL. Similar to L2S, we will recreate PCL if it becomes empty.

Lemma 3. *Given a dynamic object and a query, if they are in case S2S.a, PCL does not need to be maintained. On the other hand in S2S.b, PCL must be maintained with the* check *operation. If PCL has no candidates, PCL must be recreated.*

Algorithm 15 gives the proposed GPCL method. It maintains the top-k and PCL buffer for the 4 cases illustrated in Figure 6.6. In L2L and S2L (Lines 1-8), the top-k lists are only updated with

Algorithm 13 q.PCL.check

```
Input: $Cells$
for each cell $c \in Cells$ do
    if $c \in PCL$ then
        delete $c$ from q.PCL
    if $(maxscore(c, q) > q.PCL.low)$ and $(minscore(c, q) < kScore(q))$ then
        $q.PCL.insert(c)$
```

Algorithm 14 *q.PCL.trim*

```
Input: $kScore(q)$, $q.PCL.Low$
$PCL_{new} = \emptyset$
for each $c \in q.PCL$ do
    if $(maxscore(c, q) > q.PCL.low)$ and $(minscore(c, q) < kScore(q))$ then
        $PCL_{new}.insert(c)$
$q.PCL = PCL_{new}$
```

the upcoming o' (Lines 3, 7). PCL will be checked with Algorithm 13 (Lines 4,8) based on Lemma 1. In L2S (Lines 9-20), a candidate object will be searched and re-fill to top-k (Lines 11-15). PCL will be trimmed with Algorithm 14 (Line 17) based on Lemma 2. In S2S (Lines 21-27), PCL will be checked based on Lemma 3. Note that when PCL becomes empty, it will be re-created (Lines 19, 27).

6.3 Discussion

6.3.1 Theoretical analysis

Compared to CL, PCL is advantageous with regards to a (k+1)-th maintenance. On the other hand, if PCL is empty, it must be recreated from all cells, which is an expensive operation ($O(n^2logn^2)$). Therefore, there is a trade-off between the number of cells and recreation. Fewer cells in PCL realize an efficient search and maintenance, but lead to a higher probability that an expensive recreation will be triggered. Obviously, the number of cells in PCL depends on the number of cells n^2 in the grid. We conduct a theoretical analysis to build a cost model that balances this trade-off by estimating the best value of n^2. The total expecting cost is the sum of the expected costs of each operation.

$$E[total] = E[check] + E[trim] + E[search] + E[recreate] \tag{6.14}$$

The expected cost of an operation can be calculated by multiplying the probability to the number of the similarity computation. According to Table 6.2, the expected costs of all operations can be calculated by the equations below. Here, $P(X)$ means the probability that event X happens, PCL_c

Algorithm 15 GPCL

```
Input: $o$, $o'$, $OutQ$, $InQ$
// L2L
for each $q \in OutQ \cap InQ$ do
    Update $top\text{-}k(q)$ with $o'$.
    $q.PCL.check(\{o.cell \cup o'.cell\})$
// S2L
for each $q \in InQ - OutQ$ do
    Update $top\text{-}k(q)$ with $o'$.
    $q.PCL.check(\{o.cell \cup o'.cell \cup k.cell\})$
// L2S
for each $q \in OutQ - InQ$ do
    $o_{cand}$ = Retrieve Top-1 from $q.PCL$
    if $SimST(o', q) > SimST(o_{cand}, q)$ then
        Update $top\text{-}k(q)$ with $o'$.
    else
        Update $top\text{-}k(q)$ with $o_{cand}$.
    $q.PCL.check(\{o.cell \cup o'.cell\})$
    $q.PCL.trim(kScore(q)', q.PCL.low)$
    if $q.PCL$ is empty then
        $q.PCL.recreate$
// S2S
for each $q_i \in Q - OutQ \cup InQ$ do
    if $SimST(o, q) < q.PCL.low$ and $SimST(o', q) < q.PCL.low$ then
        continue
    else
        $q.PCL.check(\{o.cell \cup o'.cell\})$
        if $q.PCL$ is empty then
            $q.PCL.recreate$
```

represents the cells number in PCL, and PCL_o represents the object's number in PCL.

$$E[check] = (1 - P(S2S.a)) \cdot log(PCL_c) \tag{6.15}$$

$$E[trim] = P(L2S) \cdot PCL_c \tag{6.16}$$

$$E[search] = P(L2S) \cdot log(PCL_o) \tag{6.17}$$

$$E[recreate] = P(PCL.empty) \cdot n^2 \cdot logn^2 \tag{6.18}$$

To estimate $E[total]$, we need to find the probabilities of $P(S2S.a)$, $P(L2S)$ and $P(PCL.empty)$. We implement a theoretical analysis based on the following assumption. The queries and objects follow a uniform distribution in [0,1) space. With n^2 cells, the length of a cell is $l = \frac{1}{n}$, and the average number of objects in a cell is $\frac{|O|}{n^2}$, where $|O|$ is the size of the object set. For an object, we assume that a spatial (maximum) step in space is δ_s. To evaluate the similarities on spatial and

	Operation on PCL			
Case	Search	Check	Trim	Re-create
L2L	-	✓	-	-
S2L	-	✓	-	-
L2S	✓	✓	✓	✓ (if empty)
S2S.a	-	-	-	-
S2S.b	-	✓	-	✓ (if empty)

Table 6.2: Summary of the operations for PCL maintenance.

keywords, we convert the value on one to the other. δ_t is calculated by computing the change on the keyword similarity and converting it to space. We use w_q and w_o to represent the average number of keywords in an object and a query, respectively. The change in the keyword similarity is:

$$SimT(o'.\psi, q.\psi) - SimT(o.\psi, q.\psi) = \sum_{w \in o.\psi \cap q.\psi} wt(q.w) \cdot (wt(o'.w) - wt(o.w)) \approx \frac{1}{w_q^2 \cdot w_o^2} \tag{6.19}$$

By recalling that the distance in [0,1) space ranges is from 0 to $\sqrt{2}$, while the similarity on the keywords is the cosine value, we can convert the change of keyword similarity to that on the space by multiplying the ratio between the two ranges as

$$\delta_t = \frac{\sqrt{2}}{w_q^2 \cdot w_o^2}. \tag{6.20}$$

Then we estimate the probabilities. Recall that L2S is the case where an object belongs to top-k and changes out of top-k. As shown in Figure 6.7a, $P(L2S)$ can be estimated as the ratio of the areas, which is expressed as:

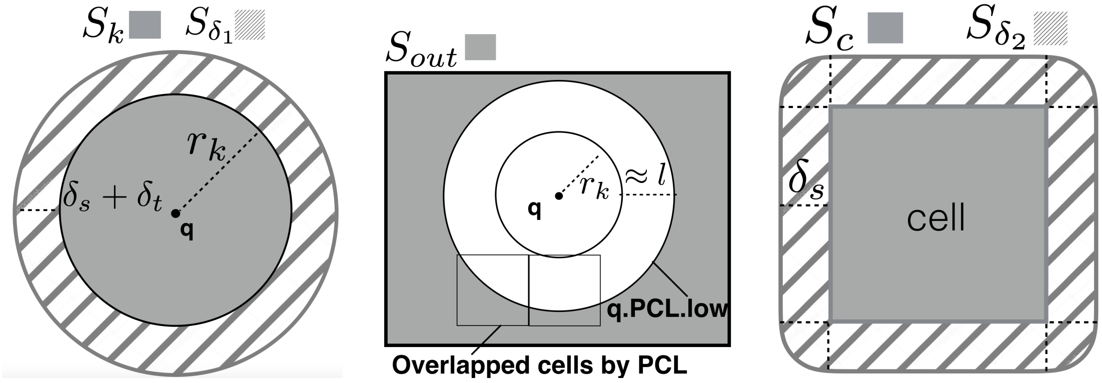
$$P(L2S) = \frac{S_{\delta_1}}{S_k + S_{\delta_1}} \cdot S_k = \pi {r_k}^2 - \frac{\pi r_k^4}{(r_k + \delta_s + \delta_t)^2} \tag{6.21}$$

Since we assume that all points follow a uniform distribution, the ratio of the area between the top-k circle and the whole space represents the ratio of the object's numbers. Hence, the radius r_k in Equation (6.21) can be calculated as:

$$\frac{\pi {r_k}^2}{1} = \frac{k}{|O|} \implies r_k = \sqrt{\frac{k}{\pi |O|}} \tag{6.22}$$

According to Figure 6.7b, P(S2S.a) is computed based on the areas in a similar way.

$$P(S2S.a) = {S_{out}}^2 = (1 - \pi(r_k + l)^2)^2 \tag{6.23}$$



(a) L2S.

(b) S2S.a.

(c) Out of a cell.

Figure 6.7: Area of each case.

The number of cells in PCL, PCL_c, is estimated by the number of cells that overlap the circle of q.PCL.low. As shown in Figure 6.7b, the number of overlapped cells is approximately the value of the perimeter divided by the side length of a cell (l). That is, $PCL_c = \frac{2\pi(r_k+l)}{l}$. Then we can calculate the objects number $PCL_o = PCL_c \cdot \frac{|O|}{n^2}$. Because PCL has PCL_o objects, the probability that PCL only has one object can be simply approximated as $\frac{1}{PCL_o}$. When PCL contains only one object, PCL may become empty when this object moves out of its cell. Note that moving out of a cell depends only on δ_s. Figure 6.7c shows the areas of this situation. In conclusion, $P(PCL.empty)$ is calculated as:

$$P(PCL.empty) = \frac{1}{PCL_o} \cdot \frac{S_{\delta_2}}{S_{\delta_2} + S_c} = \frac{1}{PCL_o} \cdot \frac{4l\delta_s + \pi\delta_s^2}{4l\delta_s + \pi\delta_S^2 + l^2} \tag{6.24}$$

Eventually, with the parameters $|O|$, k, δ_s, w_q and w_o, the only variable in Equation (6.14) is n. We used the gradient descent to figure out the n by computing the extreme value to minimize $E[total]$.

6.3.2 Batch process

If an application requires a massive number of object to be updated frequently, it may become inefficient to process the new status of the objects one by one. In such cases, batch processing is a good choice to combine the reduplicative updating top-k list with common queries. Our methods can be extended to support a batch process easily. Suppose that we received a batch of objects and aim to carry out a batch process. In *affected queries finder* module, we retrieve InQ and OutQ for a specific object when processed singly. To identify InQ and OutQ to build a batch of objects, we need to count the number of times of a query belongs to InQ and OutQ in this batch. If the times of a query belongs to InQ is larger or equal to the times that of OutQ, then we mark this query as

Datasets	YELP	TWITTER	SYN
Data size	1.1M	4.2M	12M
Default # of selected objects	220K	500K	1M
Default # of selected queries	156K	250K	1M
# of keywords	819K	3.5M	819K
# of average keywords	5.9	4.5	3

Table 6.3: Datasets statistics

a InQ query. Otherwise, it should be a OutQ query. In *Top-k refiller* module, we need to initialize PCL to maintain k-candidate objects so that we can utilize a top-k search on the PCL buffer. The PCL buffer can be maintained with a set of cells in which a batch of objects arrive in (depart from). The operations and methodologies for PCL maintenance in Section 6.2.4 also support a batch of cells directly.

On the other hand, aiming at a real-time processing problem, we need to consider the response time. It is well-known that large size of batch would result in a faster average processing time, it is unrealistic to let the client wait too long when filling up the batch. Therefore, a balance needs to be made in practice by considering several items such as computing resource, update rate, and the user requirement. Intuitively, $b_n \sim f(b_r, b_p)$, where b_n, b_r, and b_p are batch size, the user-defined response time and processing time, respectively. And, f maps b_r, b_p to tune the best batch size in practice. As an example, adding b_r to Fig. 16.b we can figure out the appropriate b_n for different processing time.

6.4 Experiments

All algorithms were implemented in C++. All indices, buffers, and algorithms were run on in-memory of a Mac with a 2.2GHz Intel Core i7 CPU and 32GB memory.

6.4.1 Setting

Dataset. We used two real datasets and one synthetic dataset. Table 6.3 shows the statistics of the datasets.

YELP is an open source dataset provided by YELP.com[4]. It contains 1.1M of reviews on 156K businesses by 220K users. We set the businesses with their locations and descriptions as queries. For the keyword attributes of queries, we randomly selected 1 to 5 keywords from the description. We set the first review of each user as the initial states of an object. Since the reviews did not contain users' locations, we intuitively set the initial location of the objects as the business location in the review. For the future status of the dynamic objects, we set a simple random walk that randomly

[4] https://www.yelp.com/dataset

±0.05 to the previous coordinates. Then we used the contents of other reviews from the same user as the changes in the keyword attribute.

TWITTER is the dataset with 4.2M geo-tag tweets from the United States[5]. In TWITTER, there are 1.2M unique users each of which has at least three geo-tag tweets. First, we selected 100K to 500K random users and used their first tweets as the queries in the experiment. For each query, we set the spatial attribute as their geo-tag. Then select 1 to 5 words from the tweets as the keyword attributes. On the other hand, we randomly selected 200K to 1M from the remaining unique users and initialized the objects with their first geo-tag tweets. The rest of the tweets from selected users were treated as the continuous states of moving objects.

SYN is a synthetic data containing 12M spatial keyword tuples. we used the data of moving points[6] for spatial attributes, which were generated by the BerlinMOD benchmark [34]. For the keyword attributes, we randomly selected 1 to 5 keywords from the TWITTER dataset and assigned them to each point. Objects and queries were selected from the SYN tuples.

Algorithms. For the *affected queries finder* module, we compared the following methods:

- **IGPT.** The group pruning techniques in [90].
- **AQF.** The proposed method with a influence circle and a group pruning technique.

For the *top-k refiller* module, we compared:

- **kmax**. The method with *kmax* buffer in [102]. For the size of the buffer, we tuned the value of *kmax* from k to $5k$. The experimental results lead us to set $3k$ as the default value of *kmax*. The tuning result is similar to related work [90].
- **GCL.** The proposed method with CL (Algorithm 12).
- **GPCL.** The proposed method with PCL (Algorithm 15).

We randomly selected over 10,000 status of dynamic objects and reported the average processing time and memory usage among different methods. The default result size k was 100 and the parameter α was a random value in (0,1). The default grid size is calculated by Equation (6.14), e.g., the grid size for the TWITTER data with $k = 100$ is 16^2. The default window size m for the keywords of an object ($o.\psi$) was 1.

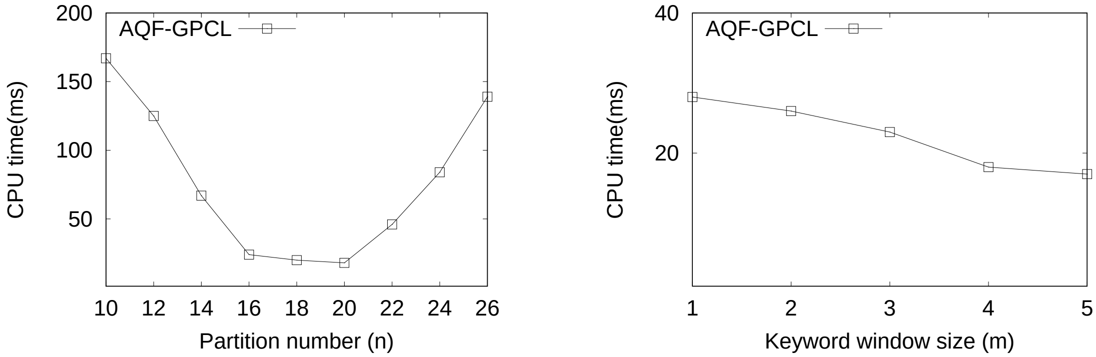
6.4.2 Experimental Results

Effect on varying n. In Figure 6.8a, we varied n^2 to observe the processing performance for testing our cost model in Section 6.3.1. The optimal theoretical results of n by minimizing Equation (6.14) is 15.44. According to experimental results, the best value is 20, while the results of 16-18 are close to the optimal value. We conclude that our cost model estimates the n well and helps to optimize the performance.

Effect on varying m. In Figure 6.8b, each object keeps five recent status of keywords. A

[5] https://datorium.gesis.org/xmlui/handle/10.7802/1166

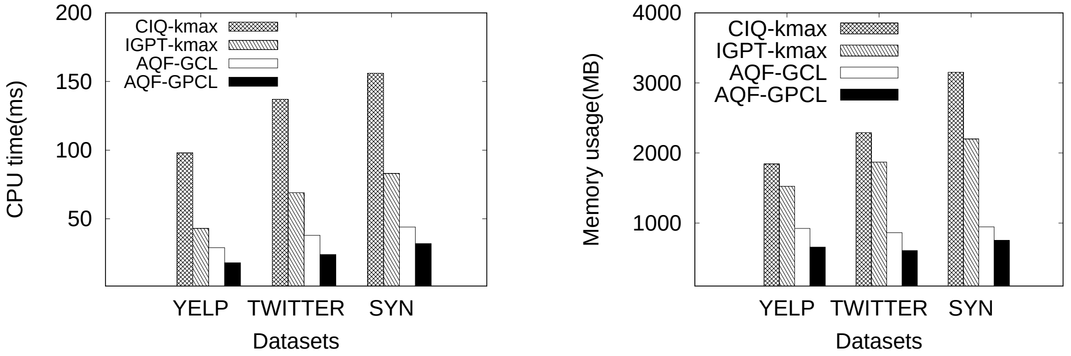
[6] http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html



(a) Varying n.

(b) Varying m.

Figure 6.8: TWITTER data.



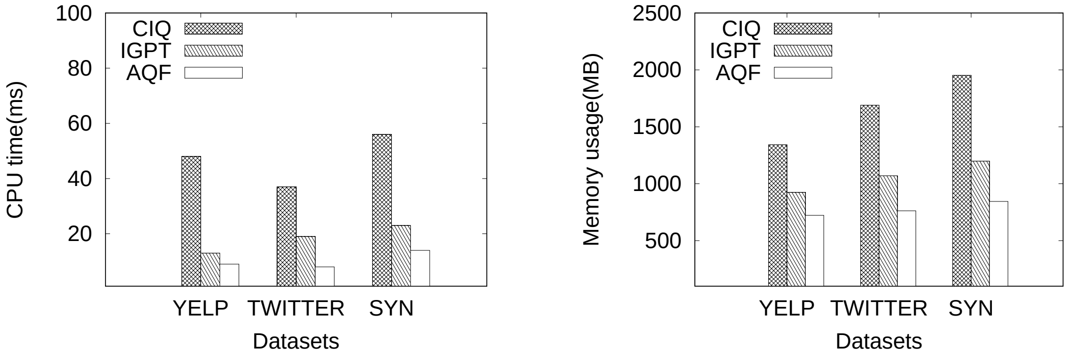
(a) Average processing time.

(b) Memory usage.

Figure 6.9: Overall processing.

Larger m leads a slighter change on the keyword attribute and a better performance on processing. Compared to the situation of keyword changes discretely ($m = 1$), we have less processing time when keywords attribute changes consecutively (i.e., to keep the previous keywords). The reason is that our PCL has a higher probability to offer the candidate results and avoid recreating.

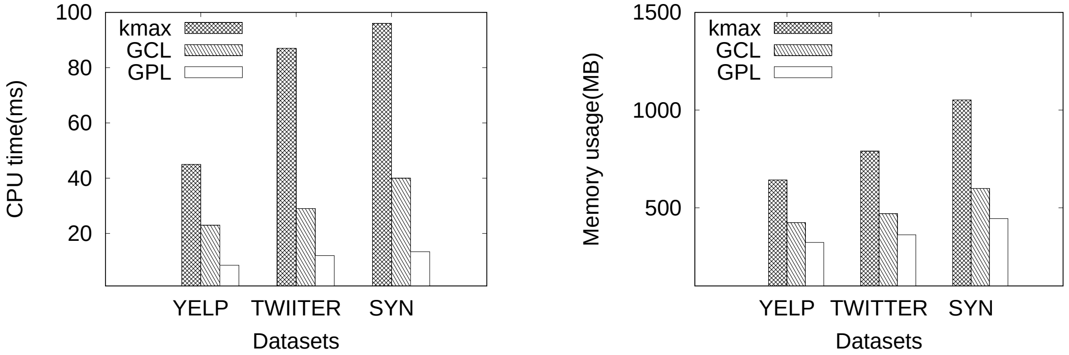
Overall processing. Figure 6.9 shows the comparison results for the overall processing with the default size of objects and queries shown in Table 6.3. Specifically, we compared the two proposed methods, AQF-GCL and AQF-GPCL to two related works. Note that AQF-GCL means that we imported the AQF algorithm as the *affected queries finder* module, and utilize the GCL algorithm as the *top-k refiller* module. For the related works IGPT and CIQ, we adjusted their techniques with the *kmax* method so that they could deal with our problem of dynamic objects. Both of our proposed methods, AQF-GCL and AQF-GPCL, have better performances than the others w.r.t the processing time and memory cost.



(a) Average processing time.

(b) Memory usage.

Figure 6.10: *Affected queries finder* module.



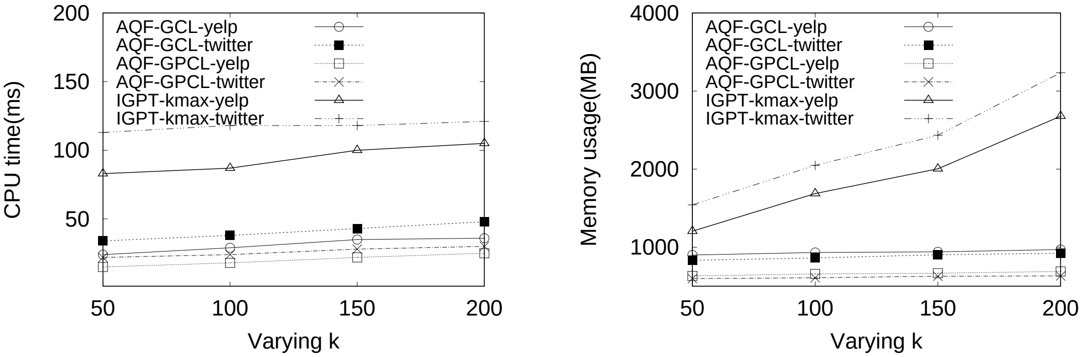
(a) Average processing time.

(b) Memory usage.

Figure 6.11: *Top-k refiller* module.

Affected queries finder. Figure 6.10 shows the comparison results of finding affected queries. Our method AQF is at least 1.5 times faster than the other methods. The reason is because IGPT and CIQ required overhead costs on the traversal of the quad-tree index from the root node. In contrast, our grid-based index can index only a few candidate queries and retrieve them directly (o(1) complexity). Moreover, the group pruning technique in AQF also boosts the performance by filtering unnecessary queries. The memory usage of our grid-index for queries is the smallest one (Figure 6.10b). We only index each query once, while IGPT indexes queries several times with keywords in the inverted files. CIQ indexes queries in multiple nodes of quad-tree.

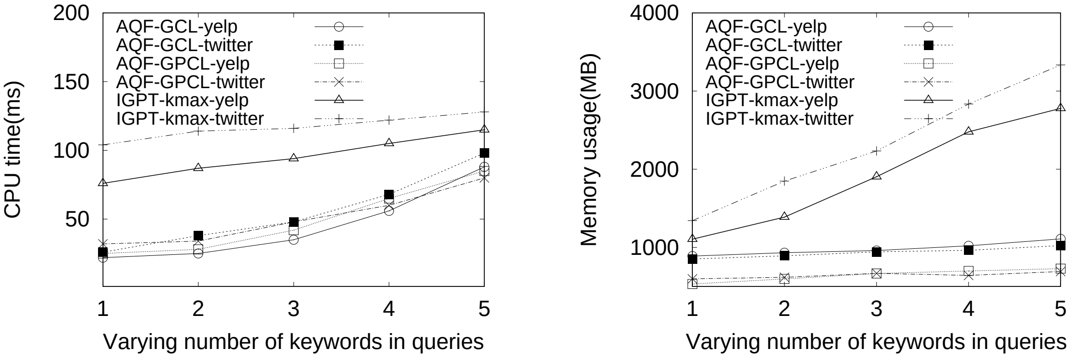
Top-k refiller. To test our *top-k refiller*, we compare the proposed GCL and GPCL methods with the *kmax* method. The results include two parts: the processing time of the top-k reevaluation and the maintenance time. According to Figure 6.11, our proposed methods are at least twice as fast as the *kmax*. The GPCL method is better than GCL since it uses the candidate refilling strategy



(a) Average processing time.

(b) Memory usage.

Figure 6.12: Varying k.



(a) Average processing time.

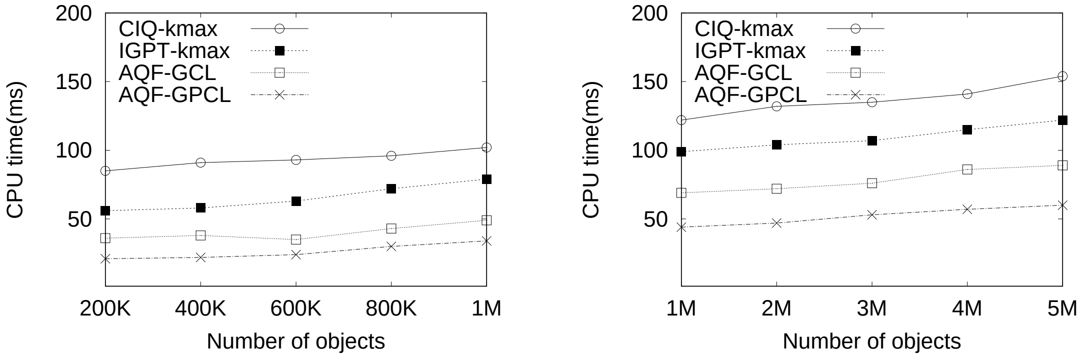
(b) Memory usage.

Figure 6.13: Varying number of keywords in queries.

rather than reevaluating the whole top-k. PCL also balances the trade-off as it only maintains a few cells. Therefore, GPCL has the least memory usage.

Effect on varying k. According to the Figure 6.12a, Only AQF-GPCL is not influenced by k since the mechanism of GPCL refills the candidate (k+1)-th object rather than reevaluating the whole top-k. From the memory usage of indices in Figure 6.12b, a large k leads to a bigger index for *kmax*-based methods. However, our proposed methods are unaffected by k since we index the identity of the cells. As we introduced previously, PCL can be seen as a small subset of CL. Hence, AQF-GPCL has a smaller memory cost than AQF-GCL.

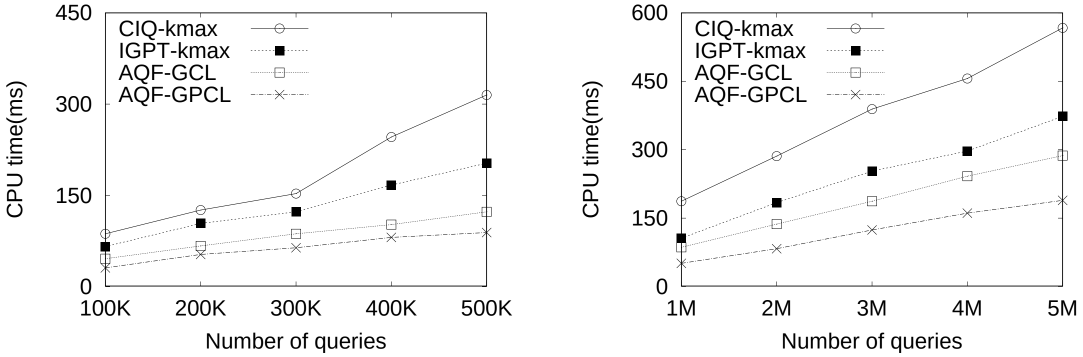
Effect on the number of query keywords. Figure 6.13 shows the processing performance among different methods with a varying number of keywords in queries. Our methods are better than the other methods in all situations for a given number of keywords. The processing time of our proposed methods is close to other methods as the number of keywords increases because many



(a) TWITTER data.

(b) SYN data.

Figure 6.14: Varying number of objects.



(a) TWITTER data.

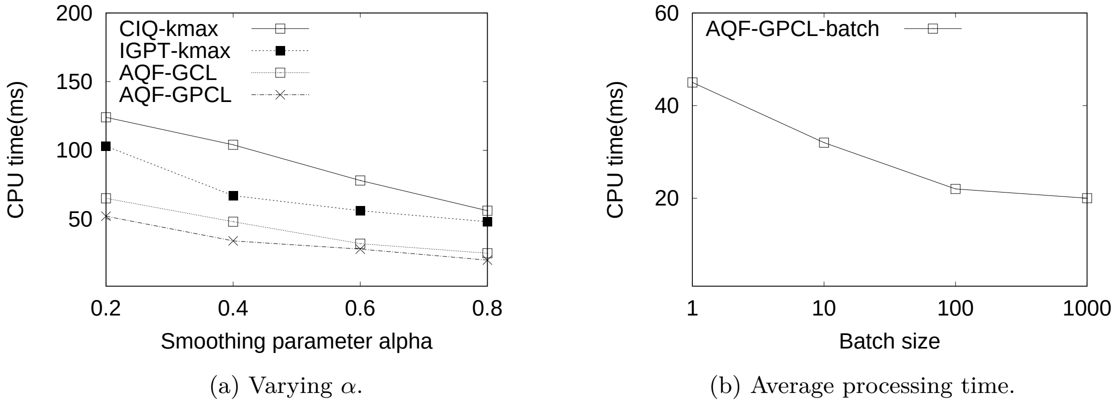
(b) SYN data.

Figure 6.15: Varying number of queries.

keywords will loosen the bound of spatial keyword similarity of a cell in Equations (6.10) and (6.11). Figure 6.13b shows the memory usage of varying number of keywords in queries. Our methods keep their superiority and AQF-GPCL costs the least memory among all indices (buffers). The inverted-file leads IGPT-based and CIQ-based methods cost more space to index data with more keywords.

Effect on number of objects and queries. Figure 6.14 shows the effects of varying the cardinality of objects with TWITTER data and SYN data. Since all algorithms keep objects with spatial indexes and implement a buffer to maintain the candidate objects, they are not affected too much by the large size of the objects. However, as the Figure 6.15 shows, a large size of queries affects the efficacy of all algorithms because more queries are affected by a dynamic object, which subsequently triggers more processes to update the results.

Effect on varying α. Figure 6.16a shows the results of varying α. All algorithms have better



(a) Varying α.

(b) Average processing time.

Figure 6.16: TWITTER data.

performance in larger α since the pruning power is mainly on the spatial attribute.

Batch process. Figure 6.16b shows the average processing time of the batch process with different batch sizes on TWITTER data. Note that batch size 1 is the case of the previous singular process of proposed AQF-GPCL. We set that a new status of an object arrives every 10ms (100 geo-tagged tweets/s [14]). The reported results also contain the waiting time that objects arrive in. Obviously, the batch process can enhance the throughput. As discussed in Section 6.3.2, we can tune a batch size with this result. e.g.: we input the 500ms as the user maximum tolerate time b_r, and we can get $b_n = 10$ is a proper balanced value for batch size.

6.5 Summary

In this chapter, we investigated a novel problem that searches dynamic spatial keyword objects continuously and propose a solution system. We employed a grid-based index to handle both dynamic objects and queries. To efficiently detect the affected queries by an object efficiently, we proposed a group pruning strategy in our *affected queries finder* module. To maintain the top-k results with a quick-response and low-cost, we proposed a sophisticated buffer called the partial cell list (PCL) to efficiently refill the top-k results in our *top-k refiller* module. We also extended the proposed methods to treat the batch process. The experiments confirmed that our proposal has a good performance compared with the baselines and related works.

As for future work, we plan to research deeply of the batch process on our problem. We also plan to propose a distributed process based solution.

Chapter 7

Conclusion and Future works

In this chapter, we conclude the main contributions of this dissertation and point out some directions of the future works.

7.1 Conclusions

In this dissertation, we studied the rank-aware query processing on multidimensional data in two data models: the user product model and spatial keyword model. Three different query problems: (a) *Aggregate Reverse Rank Query*; (b) *Weighted Aggregate Reverse Rank Query* and (c) *Continuous Search on Dynamic Spatial Keyword Objects*, are proposed in this dissertation.

7.1.1 Aggregate reverse rank query

Reverse rank queries have become important tools in marketing analysis. However, related research on reverse rank queries has focused on only a single product, which cannot deal with the common sale strategy, product bundling.

In Chapter 4, we proposed the aggregate reverse rank query (*ARR*) to address the situation of product bundling where multiple query products exist. Three different aggregate rank functions (SUM, MIN, MAX) were defined to target potential users in three normal views. To solve *ARR* efficiently, we devise a novel bound-and-filter framework to with low-dimensional data. In bound-and-filter framework, queries are bounded to calculate an approximate aggregate rank value efficiently, then tree-based structures are used to filter data in processing. For the situation of high-dimensional data, we proposed a grid index method which uses pre-calculated score bounds to reduce multiplications in the simple scan. We compared the methods through experiments on both synthetic data and real data.

7.1.2 Weighted aggregate reverse rank query

In most cases, people buy a product bundling because they appreciate a special part of the bundling. Inspired by the necessity of general aggregate reverse rank query for unequal thinking, in Chapter 5, we proposed a general, weighted aggregate reverse rank (*WARR*) query. To *WARR*, aggregate reverse rank (*ARR*) query is only a simple, special case in which all query points are treated with equal importance. *WARR* query can be critical in various applications, such as finding potential customers and analyzing marketing via different views for a set of products.

We proposed three solutions for solving *WARR* query efficiently. SFM is a straightforward way to use tree-based methods for reducing the computation of product data. The extended filtering method (EFM) adapts the previous bound-and-filter framework and is made able to solve *WARR* queries by filtering the pairwise computation from both product and preferences data. To optimize the bound, we designed a new bounding strategy, then developed and implemented an optimal bounding method (OBM). We theoretically proved the optimum of the bounds in OBM and compared the performance of the above three methods with both synthetic and real data. The results show that OBM is the most efficient of these algorithms.

7.1.3 Continuous search on dynamic spatial keyword objects

For the spatial keyword data, in Chapter 6, we investigated a novel problem that searches dynamic spatial keyword objects continuously and propose a solution system. We employed a grid-based index to handle both dynamic objects and queries. To efficiently detect the affected queries by an object efficiently, we proposed a group pruning strategy in our *affected queries finder* module. To maintain the top-k results with a quick-response and low-cost, we proposed a sophisticated buffer called the partial cell list (PCL) to efficiently refill the top-k results in our *top-k refiller* module. We also extended the proposed methods to treat the batch process. The experiments confirmed that our proposal has a good performance compared with the baselines and related works.

7.2 Future works

We have three future works regarding our works of the aggregate reverse rank query and the continuous spatial keyword search, respectively.

7.2.1 Query improvement with aggregate reverse rank queries

Our works about aggregate reverse rank query can evaluate inputted query points and find the objects with the highest ranking values. In the user product model, the aggregate reverse rank query can help manufacturers to assemble a best package that let users rank them as a higher aggregate rank. It can also apply to the application of team evaluation. For example, regarding idol group selection, aggregate reverse rank queries can help to build a popular team by adjusting the members to reach a higher aggregate rank from audience.

The problem is formalized as: Given a set of objects, a candidate set, a set of queries and a restriction, we adjust the candidate set based on the restriction, and make a higher aggregate rank to the queries.

7.2.2 Continuous spatial keyword search on road network

The road network system is consist of the graph structure, the vertices in the graph represent a place and the edges represent roads between two place. Therefore, the data of road network can be formalized as spatial keyword model. Adding a description (textual contents, or keywords) to a vertex is very common so that associating nearby business like hotel or restaurant to this place.

We plan to propose a continuous dynamic spatial keyword search on the road network. The places (vertices) are set as the monitoring spots, and to monitor moving cars with specific keywords for nearby business. Different from the work in Chapter 6 which calculate spatial similarity with the Euclidean distance, we should use graph-based distance in the road network. Moreover, we also plan to design an indexing structure considering the direction of the graph, which can help to solve the monitor processing efficiently.

7.2.3 Optimal trajectory planning for multiple spatial keyword top-k queries

People always want to save more money by getting more coupons. As the above statement, it is common to formalize the business on (or nearby) the road network with the spatial keyword model. A node of the road network can represent a continuous spatial keyword query that searches the nearby users. We plan to propose a spatial keyword query which retrieves an optimal trajectory on the road network, and this trajectory can help a user to hit the maximum number of spatial keyword queries.

The problem is formalized as: Given a spatial keyword moving object, a set of spatial keyword queries with the graph structure, a starting place which is a node of the graph, we retrieve an optimal trajectory for this object and let it hits maximum numbers of spatial keyword queries.

Bibliography

[1] Akbarinia, R., Pacitti, E., and Valduriez, P. Best position algorithms for top-k queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007* (2007), pp. 495–506.

[2] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. 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.* (1990), pp. 322–331.

[3] Benetis, R., Jensen, C. S., Karciauskas, G., and Saltenis, S. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering & Applications Symposium, IDEAS'02, July 17-19, 2002, Edmonton, Canada, Proceedings* (2002), pp. 44–53.

[4] Benetis, R., Jensen, C. S., Karciauskas, G., and Saltenis, S. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J. 15*, 3 (2006), 229–249.

[5] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[6] Berchtold, S., Keim, D. A., and Kriegel, H. 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* (1996), pp. 28–39.

[7] Butz, A. R. Convergence with hilbert's space filling curve. *J. Comput. Syst. Sci. 3*, 2 (1969), 128–146.

[8] Cao, X., Cong, G., Jensen, C. S., and Ooi, B. C. Collective spatial keyword querying. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011* (2011), pp. 373–384.

[9] Chang, Y.-C., Bergman, L. D., Castelli, V., Li, C.-S., Lo, M.-L., and Smith, J. R. The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference* (2000), pp. 391–402.

[10] Chaudhuri, S., and Gravano, L. Evaluating top-k selection queries. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), pp. 397–410.

[11] Cheema, M. A., Lin, X., Zhang, W., and Zhang, Y. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *Proceedings of the 27th, ICDE 2011* (2011), pp. 577–588.

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

[13] Chen, L., Cong, G., and Cao, X. An efficient query indexing mechanism for filtering geo-textual data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 749–760.

[14] Chen, L., Cong, G., Cao, X., and Tan, K. Temporal spatial-keyword top-k publish/subscribe. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015* (2015), pp. 255–266.

[15] Chen, L., Cong, G., Jensen, C. S., and Wu, D. Spatial keyword query processing: An experimental evaluation. *PVLDB 6*, 3 (2013), 217–228.

[16] Chen, Y., Suel, T., and Markowetz, A. Efficient query processing in geographic web search engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006* (2006), pp. 277–288.

[17] Chen, Z., Cong, G., Zhang, Z., Fu, T. Z. J., and Chen, L. Distributed publish/subscribe query processing on the spatio-textual data stream. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017* (2017), pp. 1095–1106.

[18] Chester, S., Thomo, A., Venkatesh, S., and Whitesides, S. 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* (2013), pp. 201–208.

[19] Cheung, K. L., and Fu, A. W. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record 27*, 3 (1998), 16–21.

[20] Choi, D.-W., and Chung, C.-W. Nearest neighborhood search in spatial databases. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (2015), IEEE, pp. 699–710.

[21] Christoforaki, M., He, J., Dimopoulos, C., Markowetz, A., and Suel, T. Text vs. space: efficient geo-search query processing. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011* (2011), pp. 423–432.

[22] Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece* (1997), pp. 426–435.

[23] Cong, G., and Jensen, C. S. Querying geo-textual data: Spatial keyword queries and beyond. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 2207–2212.

[24] Cong, G., Jensen, C. S., and Wu, D. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB 2*, 1 (2009), 337–348.

[25] Das, G., Gunopulos, D., Koudas, N., and Sarkas, N. Ad-hoc top-k query answering for data streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007* (2007), pp. 183–194.

[26] Dellis, E., and Seeger, B. Efficient computation of reverse skyline queries. In *Proceedings of the 33rd International Conference on VLDB* (2007), pp. 291–302.

[27] Deng, K., Sadiq, S. W., Zhou, X., Xu, H., Fung, G. P. C., and Lu, Y. On group nearest group query processing. *IEEE Trans. Knowl. Data Eng. 24*, 2 (2012), 295–308.

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

[29] Dong, Y., Chen, H., Furuse, K., and Kitagawa, H. Efficient processing of aggregate reverse rank queries. In *Database and Expert Systems Applications - 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I* (2017), pp. 159–166.

[30] Dong, Y., Chen, H., Furuse, K., and Kitagawa, H. Bound-and-filter framework for aggregate reverse rank queries. *T. Large-Scale Data- and Knowledge-Centered Systems 38* (2018), 1–26.

[31] Dong, Y., Chen, H., Furuse, K., and Kitagawa, H. Efficient methods for aggregate reverse rank queries. *IEICE Transactions 101-D*, 4 (2018), 1012–1020.

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

[33] Dong, Y., Chen, H., Yu, J. X., Furuse, K., and Kitagawa, H. Weighted aggregate reverse rank queries. *ACM Trans. Spatial Algorithms and Systems 4*, 2 (2018), 5:1–5:23.

[34] Düntgen, C., Behr, T., and Güting, R. H. Berlinmod: a benchmark for moving object databases. *VLDB J. 18*, 6 (2009), 1335–1368.

[35] Fagin, R. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci. 58*, 1 (1999), 83–99.

[36] Fagin, R., Lotem, A., and Naor, M. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 66*, 4 (2003), 614–656.

[37] Felipe, I. D., Hristidis, V., and Rishe, N. Keyword search on spatial databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico* (2008), pp. 656–665.

[38] Finkel, R. A., and Bentley, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Inf. 4* (1974), 1–9.

[39] Gao, Y., Liu, Q., Chen, G., Zheng, B., and Zhou, L. Answering why-not questions on reverse top-k queries. *PVLDB 8*, 7 (2015), 738–749.

[40] Guo, L., Shao, J., Aung, H. H., and Tan, K. Efficient continuous top-k spatial keyword queries on road networks. *GeoInformatica 19*, 1 (2015), 29–60.

[41] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984* (1984), pp. 47–57.

[42] Hjaltason, G. R., and Samet, H. Distance browsing in spatial databases. *ACM Trans. Database Syst. 24*, 2 (1999), 265–318.

[43] Hristidis, V., Koudas, N., and Papakonstantinou, Y. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD* (2001), pp. 259–270.

[44] Huang, W., Li, G., Tan, K., and Feng, J. Efficient safe-region construction for moving top-k spatial keyword queries. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012* (2012), pp. 932–941.

[45] Ilyas, I. F., and Aref, W. G. Rank-aware query processing and optimization. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* (2005), p. 1144.

[46] Ilyas, I. F., Aref, W. G., Elmagarmid, A. K., Elmongui, H. G., Shah, R., and Vitter, J. S. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst. 31*, 4 (2006), 1257–1304.

[47] Ilyas, I. F., Beskales, G., and Soliman, M. A. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv. 40*, 4 (2008).

[48] Ilyas, I. F., Shah, R., Aref, W. G., Vitter, J. S., and Elmagarmid, A. K. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004* (2004), pp. 203–214.

[49] Kanth, K. V. R., Ravada, S., and Abugov, D. Quadtree and r-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data , Madison, Wisconsin, June 3-6, 2002* (2002), pp. 546–557.

[50] Khodaei, A., Shahabi, C., and Li, C. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *Database and Expert Systems Applications, 21st International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part I* (2010), pp. 450–466.

[51] Korn, F., and Muthukrishnan, S. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD* (2000), pp. 201–212.

[52] Li, G., Feng, J., and Xu, J. DESKS: direction-aware spatial keyword search. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012* (2012), pp. 474–485.

[53] Li, G., Wang, Y., Wang, T., and Feng, J. Location-aware publish/subscribe. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013* (2013), pp. 802–810.

[54] Lian, X., and Chen, L. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proceedings of the ACM SIGMOD* (2008), pp. 213–226.

[55] Liu, Q., Gao, Y., Chen, G., Zheng, B., and Zhou, L. Answering why-not and why questions on reverse top-k queries. *VLDB J. 25*, 6 (2016), 867–892.

[56] Lu, J., Lu, Y., and Cong, G. Reverse spatial and textual k nearest neighbor search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011* (2011), pp. 349–360.

[57] Mahmood, A. R., Aly, A. M., Qadah, T., Rezig, E. K., Daghistani, A., Madkour, A., Abdelhamid, A. S., Hassan, M. S., Aref, W. G., and Basalamah, S. M. Tornado: A distributed spatio-textual stream processing system. *PVLDB 8*, 12 (2015), 2020–2023.

[58] Mahmood, A. R., and Aref, W. G. Query processing techniques for big spatial-keyword data. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), pp. 1777–1782.

[59] Mahmood, A. R., Aref, W. G., and Aly, A. M. FAST: frequency-aware spatio-textual indexing for in-memory continuous filter query processing. *CoRR abs/1709.02529* (2017).

[60] Mahmood, A. R., Daghistani, A., Aly, A. M., Tang, M., Basalamah, S. M., Prabhakar, S., and Aref, W. G. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018* (2018), pp. 219–228.

[61] Marian, A., Bruno, N., and Gravano, L. Evaluating top-*k* queries over web-accessible databases. *ACM Trans. Database Syst. 29*, 2 (2004), 319–362.

[62] McAuley, J. J., Pandey, R., and Leskovec, J. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015* (2015), pp. 785–794.

[63] McAuley, J. J., Targett, C., Shi, Q., and van den Hengel, A. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015* (2015), pp. 43–52.

[64] Morton, G. M. A computer oriented geodetic data base and a new technique in file sequencing.

[65] Mouratidis, K. Geometric approaches for top-k queries. *PVLDB 10*, 12 (2017), 1985–1987.

[66] Mouratidis, K., Bakiras, S., and Papadias, D. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data ,Chicago, Illinois, USA, June 27-29, 2006* (2006), pp. 635–646.

[67] Mouratidis, K., Hadjieleftheriou, M., and Papadias, D. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005* (2005), pp. 634–645.

[68] Mouratidis, K., Zhang, J., and Pang, H. Maximum rank query. *PVLDB 8*, 12 (2015), 1554–1565.

[69] Omohundro, S. M. *Five balltree construction algorithms.* International Computer Science Institute Berkeley, 1989.

[70] Papadias, D., Shen, Q., Tao, Y., and Mouratidis, K. Group nearest neighbor queries. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA* (2004), pp. 301–312.

[71] Papadias, D., Tao, Y., Mouratidis, K., and Hui, C. K. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst. 30*, 2 (2005), 529–576.

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

[73] Qian, Y., Li, H., Mamoulis, N., Liu, Y., and Cheung, D. W. Reverse k-ranks queries on large graphs. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* (2017), pp. 37–48.

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

[75] Rocha-Junior, J. B., Gkorgkas, O., Jonassen, S., and Nørvåg, K. Efficient processing of top-k spatial keyword queries. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings* (2011), pp. 205–222.

[76] Roussopoulos, N., Kelley, S., and Vincent, F. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995.* (1995), pp. 71–79.

[77] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England* (1987), pp. 507–518.

[78] Stanoi, I., Agrawal, D., and El Abbadi, A. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop* (2000), pp. 44–53.

[79] Tao, Y., Hristidis, V., Papadias, D., and Papakonstantinou, Y. Branch-and-bound processing of ranked queries. *Inf. Syst. 32*, 3 (2007), 424–445.

[80] Tao, Y., Papadias, D., and Lian, X. Reverse knn search in arbitrary dimensionality. In *Proceedings of the 13th International Conference on VLDB* (2004), pp. 744–755.

[81] Tao, Y., Papadias, D., Lian, X., and Xiao, X. Multidimensional reverse *k* NN search. *VLDB J. 16*, 3 (2007), 293–316.

[82] Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., and Srivastava, D. Ranked join indices. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India* (2003), pp. 277–288.

[83] U, L. H., Zhang, J., Mouratidis, K., and Li, Y. Continuous top-k monitoring on document streams. *IEEE Trans. Knowl. Data Eng. 29*, 5 (2017), 991–1003.

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

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

[86] Vlachou, A., Doulkeridis, C., and Yannis Kotidis, e. Monochromatic and bichromatic reverse top-k queries. pp. 1215–1229.

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

[88] Vlachou, A., and .el, C. D. Monitoring reverse top-k queries over mobile devices. In *MobiDE* (2011), pp. 17–24.

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

[90] Wang, X., Zhang, Y., Zhang, W., Lin, X., and Huang, Z. SKYPE: top-k spatial-keyword publish/subscribe over sliding window. *PVLDB 9*, 7 (2016), 588–599.

[91] Wang, X., Zhang, Y., Zhang, W., Lin, X., and Wang, W. Ap-tree: efficiently support location-aware publish/subscribe. *VLDB J. 24*, 6 (2015), 823–848.

[92] Weber, R., Schek, H., and Blott, S. 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* (1998), pp. 194–205.

[93] Wu, D., Cong, G., and Jensen, C. S. A framework for efficient spatial web object retrieval. *VLDB J. 21*, 6 (2012), 797–822.

[94] Wu, D., Yiu, M. L., and Jensen, C. S. Moving spatial keyword queries: Formulation, methods, and analysis. *ACM Trans. Database Syst. 38*, 1 (2013), 7:1–7:47.

[95] Wu, D., Yiu, M. L., Jensen, C. S., and Cong, G. Efficient continuously moving top-k spatial keyword query processing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE, 2011, April 11-16, 2011, Hannover, Germany* (2011), pp. 541–552.

[96] Xiao, G., Li, K., Zhou, X., and Li, K. Efficient monochromatic and bichromatic probabilistic reverse top-k query processing for uncertain big data. *J. Comput. Syst. Sci. 89* (2017), 92–113.

[97] Xiong, X., Mokbel, M. F., and Aref, W. G. SEA-CNN: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* (2005), pp. 643–654.

[98] Yan, H., Ding, S., and Suel, T. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009* (2009), pp. 401–410.

[99] Yang, S., Cheema, M. A., Lin, X., and Wang, W. Reverse k nearest neighbors query processing: Experiments and analysis. *PVLDB 8*, 5 (2015), 605–616.

[100] Yang, S., Cheema, M. A., Lin, X., and Zhang, Y. SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In *IEEE 30th International Conference on Data Engineering, ICDE* (2014), pp. 760–771.

[101] Yao, B., Li, F., and Kumar, P. Reverse furthest neighbors in spatial databases. In *Proceedings of the 25th ICDE* (2009), pp. 664–675.

[102] Yi, K., Yu, H., Yang, J., Xia, G., and Chen, Y. Efficient maintenance of materialized top-k views. In *Proceedings of the 19th International Conference on Data Engineering, ICDE, March 5-8, 2003, Bangalore, India* (2003), pp. 189–200.

[103] Yiu, M. L., Mamoulis, N., and Papadias, D. Aggregate nearest neighbor queries in road networks. *IEEE Trans. Knowl. Data Eng. 17*, 6 (2005), 820–833.

[104] Yu, A., Agarwal, P. K., and Yang, J. Processing a large number of continuous preference top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), pp. 397–408.

[105] Yu, X., Pu, K. Q., and Koudas, N. Monitoring k-nearest neighbor queries over moving objects. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* (2005), pp. 631–642.

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

[107] Zheng, B., Zheng, K., Xiao, X., Su, H., Yin, H., Zhou, X., and Li, G. Keyword-aware continuous knn query on road networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (2016), pp. 871–882.

List of Publications

Refereed journal papers

- Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Efficient Methods for Aggregate Reverse Rank Queries"
  IEICE TRANSACTIONS on Information and Systems. Volume E101-D No.4, pp. 1012-1020. 2018.

- Yuyang Dong, Hanxiong Chen, Jeffery Xu Yu, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Weighted Aggregate Reverse Rank Queries"
  ACM Transactions on Spatial Algorithms and Systems (TSAS). Volume 4 Issue 2 Article 5. 2018.

- Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Bound-and-filter Framework for Aggregate Reverse Rank Queries"
  Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS). XXXVIII Special Issue on Database and Expert Systems Applications, pp. 1-26, 2018

Refereed international conference papers

- Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Aggregate Reverse Rank Queries"
  Proc.27th International Conference on Database and Expert Systems Applications (DEXA 2016). pp.87-101, Porto, Portugal, September 5-8, 2016. (Best Paper Award).

- Yuyang Dong, Hanxiong Chen, Jeffery Xu Yu, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Grid-Index algorithm for reverse rank queries"
  Proc. 20th International Conference on Extending Database Technology (EDBT 2017). pp 306-317,Venice, Italy, March 21-24, 2017.

- Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "Efficient Processing of Aggregate Reverse Rank Queries"
  Proc.28th International Conference on Database and Expert Systems Applications (DEXA 2017). pp.159-166, Lyon, France, August 28-31, 2017.

- Yuyang Dong, Hanxiong Chen, Hiroyuki Kitagawa.
  "Continuous Search on Dynamic Spatial Keyword Objects"
  Proc. 35th IEEE International Conference on Data Engineering (ICDE 2019). pp., Macau SAR, China, April 8-12, 2019. (to appear)

Non-refereed domestic conference Papers

- Yuyang Dong, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "A Branch-and-Bound Method for Group Reverse Queries," in *The 8th Forum on Data Engineering and Information Management (DEIM 2016)*, D7-1, Fukuoka - Japan, Feb 29 - March 2, 2016.

- Yuyang Dong, Song Wang, Hanxiong Chen, Kazutaka Furuse, Hiroyuki Kitagawa.
  "A High-dimensional Solution for Aggregate Reverse Rank Query" in *The 9th Forum on Data Engineering and Information Management (DEIM 2017)*, G1-1, Takayama - Japan, March 6-8, 2017.