Spatial keyword querying beyond the single geo-textual object granularity

Cao, Xin

2014

Cao, X. (2014). Spatial keyword querying beyond the single geo-textual object granularity.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61681>

<https://doi.org/10.32657/10356/61681>

SPATIAL KEYWORD QUERYING BEYOND THE SINGLE GEO-TEXTUAL OBJECT GRANULARITY

CAO XIN

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

2014

SPATIAL KEYWORD QUERYING BEYOND THE SINGLE GEO-TEXTUAL OBJECT GRANULARITY

CAO XIN

School of Computer Engineering

A thesis submitted to the Nanyang Technological University

in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

2014

Acknowledgments

I would like to express my sincere appreciation to my supervisor Dr. Gao Cong. I began to work with him in Aalborg University in Denmark five years ago. At that time I just graduated as a master student and knew little about research. Under his patient and inspiring supervision, I have learned not only cutting-edge knowledge on spatial databases and information retrieval, but also a serious and down-to-earth working attitude. Then, three years ago I followed him to Nanyang Technological University and studied as his first Ph.D. student researching on spatial keyword querying and other related topics. He is not merely a supervisor, but a big brother and a good friend in my life. I feel so lucky to have such a nice and knowledgeable supervisor.

I feel very grateful to Dr. Christian S. Jensen for his continuous support and insightful guidance, who supervised me when I was a research assistant in Aalborg University. Especially, I have learned many research writing and presenting skills from him.

I would also like to acknowledge my friends at Nanyang Technological University for their friendship and insightful discussions. Quan Yuan, Zhiqiang Xu, Diwen Zhu, Ning Chen, Boshen Zhang, Wenqing Lin were enrolled as Ph.D. students in NTU at the same time with me. We together have great fun in the past three years. Lisi Chen, Zhen Hai, Yaqiong Liu, Shaohua Li, Wei Wei, Shanshan Feng, Tao Guo, Kaiyu Feng, Nguyen Tuan Anh Pham, Kaiqi Zhao, and Zhida Chen are being or have been supervised by Dr. Gao Cong. I thank them for the useful discussions and pleasant collaborations during my study. I also would like to thank my lab colleagues, Shumo Chu, Zongyang Ma, Chenliang Li, Peilin Zhao, Hao Xia, Bin Li, Pengcheng Wu, Jun Zhang, Youze Tang, Sibow Wang, Yong Liu, Min Wu, Xiaogang Han, etc. It is my pleasure to get to know all of them.

I would like to express my deep gratitude to my parents and my grandparents for their support and love. I know that they must feel proud of my achievements. More especially to my dear father and my beloved grandmother who I lost when I was an undergraduate student. May their soul rest in peace in heaven. They will always be the source of my courage and a part of me. I also want to thank to my in-laws for their unconditional support. Foremost, I want to thank my wife Xue Wu, who supports me with her care and love. I dedicate this thesis to her.

Contents

Acknowledgments	i
List of Figures	v
List of Tables	viii
Summary	ix
1 Introduction	1
1.1 Background: Geo-Textual Objects	1
1.2 Spatial Keyword Queries	2
1.3 Retrieving Geo-Textual Objects beyond the Single Granularity	4
1.3.1 Problems and Research Scope	5
1.3.2 Approaches and Methodology	8
1.4 Summary of Contributions	12
1.5 Thesis Organization	13
2 Literature Survey	14
2.1 Mapping Locations with Web Contents	14
2.2 Spatial Indices and Queries	15
2.2.1 Spatial Indices	15
2.2.2 Queries over Spatial Databases and Road Networks	19
2.3 Information Retrieval Indices and Models	20
2.3.1 Inverted Files and Signature Files	20
2.3.2 Information Retrieval Models	21
2.4 Spatial Keyword Indices and Queries	23
2.4.1 Spatial Keyword Indexing Classification	23
2.4.2 Spatial Keyword Queries	24
2.5 Summary	39

3	Top-k Prestige-based Spatial Keyword Query	40
3.1	Problem definition	40
3.2	Proposed Solutions	42
3.2.1	Baseline Algorithm	42
3.2.2	Early Stop EBC Algorithm (ES-EBC)	44
3.2.3	Subgraph-Based EBC Algorithm (S-EBC)	48
3.3	Experimental Study	57
3.3.1	Experimental Settings	57
3.3.2	Experimental Results	59
3.4	Additional Related Work	64
3.5	Conclusion	65
4	Spatial Group Keyword Query	66
4.1	Problem Statement	66
4.2	Processing SUM SGK Queries	68
4.2.1	Approximation Algorithm	69
4.2.2	Exact Algorithm Without an Index	71
4.2.3	An Improved Exact Algorithm Without an Index	76
4.2.4	Exact Algorithm Using an Index	78
4.3	Processing MAX+MAX SGK Queries	83
4.3.1	Approximation Algorithm 1	83
4.3.2	Approximation Algorithm 2	84
4.3.3	Exact Algorithm 1	87
4.3.4	Exact Algorithm 2	90
4.4	Processing Top- k Spatial Group Keyword Queries	95
4.4.1	Top- k SUM SGK Query	96
4.4.2	Top- k MAX+MAX SGK Query	97
4.5	Experimental Study	98
4.5.1	Experimental Settings	98
4.5.2	Experimental Results on SGK Queries	99
4.6	Additional Related Work	111
4.7	Conclusion	112

5	Keyword-aware Optimal Route Search	113
5.1	Problem Statement	113
5.2	Algorithms	116
5.2.1	Pre-processing	116
5.2.2	Approximation Algorithm OSScaling	116
5.2.3	Approximation Algorithm BucketBound	124
5.2.4	Greedy Algorithm	128
5.2.5	Keyword-aware Top- k Optimal Route Search	130
5.3	Experimental Study	130
5.3.1	Experimental Settings	130
5.3.2	Experimental Results	132
5.4	Additional Related Work	139
5.5	Conclusion	141
6	Conclusions and Future Work	142
6.1	Conclusions	142
6.2	Future Work	143
6.2.1	Extension of k PSK Query	143
6.2.2	Extension of SGK Query	144
6.2.3	Extension of KOR Query	144
6.2.4	Other Approaches to Spatial Keyword Querying beyond the Single Object Granularity	145
	List of Publications	147
	References	149

List of Figures

1.1	Examples of geo-textual objects	2
1.2	Prestige propagation example	6
2.1	An example of an R-tree	16
2.2	An example of an R^+ -tree	16
2.3	An example of an R^* -tree	17
2.4	An example of a Quad-tree	18
2.5	An example of a grid index	18
2.6	An example dataset containing 8 geo-textual objects	25
2.7	An example of the spatial-first index	26
2.8	An example of the text-first index	27
2.9	An example of an IR-Tree	31
2.10	An example of the virtual bR^* -tree	35
3.1	Example of the sub-graph partition, hub nodes, and extended IR-tree . .	55
3.2	Varying k	59
3.3	Varying # keywords	59
3.4	Varying α	60
3.5	Varying β	60
3.6	Varying k (Web)	61
3.7	Varying # keywords (Web)	61
3.8	Varying k (Hotel)	61
3.9	Varying # keywords (Hotel)	61
3.10	Scalability with data size	62
3.11	Varying threshold γ (Hotel)	63

3.12	Varying threshold ξ (Hotel)	63
4.1	Exhaustive search space around o_i containing t_{inf}	92
4.2	Results of SUM SGK queries on Hotel	100
4.3	Results of MAX+MAX SGK queries on Hotel	102
4.4	Results of MAX+MAX SGK queries on GN	103
4.5	Results of MAX+MAX SGK queries on Web	104
4.6	Comparison of approximation algorithms on Hotel	105
4.7	Comparison of exact algorithms on Hotel	105
4.8	Comparison of approximation algorithms on GN	106
4.9	Comparison of exact algorithms on GN	107
4.10	Results on queries without frequent keywords	108
4.11	Scalability of algorithms	109
4.12	Runtime (top- k)	109
4.13	Real Examples	110
5.1	An example of $\mathcal{G}$	114
5.2	Steps of Example 5.12	121
5.3	Procedure of Algorithm 2	126
5.4	Runtime (Flickr)	132
5.5	Runtime (Flickr)	132
5.6	Runtime (Flickr)	133
5.7	Relative Ratio (Flickr)	133
5.8	Runtime (Flickr)	134
5.9	Relative ratio (Flickr)	134
5.10	Relative ratio (Flickr)	135
5.11	Relative ratio (Flickr)	135
5.12	Relative ratio (Flickr)	136
5.13	Failure percentage (Flickr)	136
5.14	Runtime (Flickr)	136
5.15	Relative ratio (Flickr)	136
5.16	Runtime (KkR)	137

5.17 Scalability	137
5.18 Runtime (Synthetic)	138
5.19 Runtime (Synthetic)	138
5.20 Example route 1	138
5.21 Example route 2	138

List of Tables

2.1	An example of an inverted file	21
2.2	Example of the KR^* -tree list	28
2.3	Content of inverted files of the IR-Tree	31
2.4	Comparison of existing geo-textual indices	35
3.1	Dataset properties	59
3.2	Index structure sizes (MB)	62
3.3	Effectiveness of different algorithms	63
4.1	Example dataset	72
4.2	Results after processing o_1	82
4.3	Results after processing o_2	82
4.4	Results after processing o_3	82
5.1	Labels contents	122

Summary

With the proliferation of geo-positioning techniques and mobile devices such as smartphones and tablet computers, the web is increasingly being used by mobile users and accurate user positioning is increasingly available. As a result, a spatial, or geographical web is emerging where contents and users are associated with locations. This leads to the fact that massive amounts of objects are available on the web that possess both a geographical location and a textual description. Such **geo-textual objects** include stores, tourist attractions, hotels, restaurants, businesses, entertainment services, public transport, etc.

The availability of substantial amounts of geo-textual objects gives prominence to **spatial keyword queries** that target these objects. Such queries exploit both locations and textual descriptions and occur in many types of mobile and traditional web services and applications, e.g., Yellow Pages and Maps services, which greatly facilitate our everyday life. For example, in Google Maps the functionality “search nearby” allows users to retrieve points of interest around a specified location. Spatial keyword querying has attracted significant industrial and academic interests. There exist many proposals on studying spatial keyword queries, which retrieve lists of geo-textual objects that are both textually relevant to the query and satisfy a spatial query predicate. However, most existing work on spatial keyword querying treats the geo-textual objects as independent. This thesis focuses on efficiently processing the spatial keyword queries beyond the single geo-textual object granularity.

First, we believe that a relevant result geo-textual object with nearby objects that are also relevant to the query is likely to be preferable over a relevant object without relevant nearby objects. We propose the concept of prestige-based relevance to capture both the textual relevance of an object to a query and the effects of nearby objects. Based on this, a new type of query, the *top-k Prestige-based Spatial Keyword* (*kPSK*) query, is proposed that retrieves the top-*k* geo-textual objects ranked according to both prestige-based relevance and location proximity. Two algorithms are devised to answer *kPSK* queries. Empirical studies with real-world datasets demonstrate that *kPSK* queries are more effective in retrieving geo-textual objects than is a previous approach without the consideration of the effects of nearby objects; the experimental results also show that the proposed algorithms are scalable and outperform a baseline approach significantly.

Next, proposals for spatial keyword search so far generally focus on finding individual objects rather than finding groups of objects where the objects in a group collectively satisfy the requirements. We define another new type of query named the *Spatial Group Keyword (SGK)* query, which retrieves a group of geo-textual objects such that the group’s keywords cover the query’s keywords and such that objects are nearest to the query location and have the lowest inter-object distances. Specifically, we study two variants of this problem, both of which are NP-hard. We devise exact solutions as well as approximate solutions with provable approximation bounds to the problems. We present empirical studies that offer insight into the efficiency and accuracy of the solutions.

Third, we consider a spatial keyword query over road networks that retrieves routes that are formed by a sequence of geo-textual objects. Identifying a preferable route is an important problem that finds applications in map services. When a user plans a trip within a city, the user may want to find “a most popular route such that it passes by *shopping mall*, *restaurant*, and *pub*, and the travel time to and from my hotel is within 4 hours.” However, no existing algorithms can be used to answer such queries. Motivated by this, we define the *Keyword-aware Optimal Route* query, denoted by **KOR**, which finds a route such that it covers a set of user-specified keywords, a specified budget constraint is satisfied, and an objective score of the route is optimal. The problem of answering **KOR** queries is NP-hard. We first devise an approximation algorithm with provable approximation bounds. Based on this algorithm, a more efficient approximation algorithm is proposed. We also design a greedy approximation algorithm. Results of empirical studies show that all the proposed algorithms are capable of answering **KOR** queries efficiently. The empirical studies also evaluate the accuracy of the proposed algorithms.

Chapter 1

Introduction

1.1 Background: Geo-Textual Objects

The web is increasingly being used by mobile users. In recognition of this development, Google announced in 2010 that the company would now develop services “mobile first,” meaning that services are developed first for mobile devices and users and only then adapted to desktop devices and users.

We are witnessing a proliferation of geo-positioning capabilities. Smartphones, navigation devices, some tablets, and other mobile devices are equipped with GPS receivers. Other available positioning technologies exploit the communication infrastructures used by mobile devices, such as Wi-Fi and 3G. Stated briefly, this can be achieved by building and maintaining a so-called location fingerprint database of pairs of a ground truth location and the base stations and cell towers seen by the radios in a mobile device from that location. It is then possible to assign a location to a device when the device reports the base stations and cell towers seen from its location. This type of technology may be used for both outdoor and indoor positioning, but offers less accurate positioning than does GPS. With such proliferation of geo-positioning, accurate user positioning is increasingly available.

As a result of the developments outlined above, a spatial, or geographical web is emerging where content and users are associated with locations that are used in a wide range of location-based services. This leads to the fact that massive amounts of objects are available on the web that have an associated geographical location and a textual description. Such **geo-textual objects** include stores, tourist attractions, hotels, restaurants, businesses, entertainment services, public transport, etc. Formally, a geo-textual object o is in format of $o = (\psi, \lambda)$, where $o.\psi$ is the textual description of o (represented by a set of keywords) and $o.\lambda$ is the location of o (represented by a point or a shape such as the rectangle). Figure 1.1 shows the examples of geo-textual objects in Google Maps.

Each balloon represents a geo-textual object—a canteen in Nanyang Technological University. Each object is located on the map, which has an address (*e.g.*, “50 Nanyang Ave. Singapore 639798”) and a description (*e.g.*, “Canteen B”).

Figure 1.1: Examples of geo-textual objects

The geo-textual objects could be obtained from location-based services provided by major commercial search engines. For example, in Google Maps¹, many geo-referenced points of interest are being associated with descriptive texts. They could also be extracted from geo-tagged web contents, such as the business in online yellow pages (the addresses are generated by service providers), the geo-tagged photos in Flickr² (where users can mark the location for a uploaded photo where it is taken), and the geo-located tweets in Twitter³ (where users can attach their current location to tweets detected automatically by the smartphones). In recent years, the emerging location-based social network applications such as Foursquare⁴ have attracted significant attention. Foursquare allows mobile users to “check in” at locations and create some “tips” which serve as suggestions for things to do, see, or eat at the location. These locations with tips generated by the users could be viewed as geo-textual objects as well.

1.2 Spatial Keyword Queries

The availability of a substantial amount of the so-called geo-textual objects brings about that a substantial fraction of queries submitted to web search engines have local intent and

¹<http://maps.google.com/>

²<http://www.flickr.com/>

³<http://twitter.com/>

⁴<http://foursquare.com/>

target these objects. One article reports that 53% of mobile searches on Bing have local intent [91]. An older, PC-centric finding from Google is that 20% of Google searches are related to locations [48]. The prosperity of mobile internet and mobile devices gives the prominence to the **spatial keyword queries** [15, 24, 30, 34, 51, 63, 88, 99, 113]. Typically, such a query takes a location and a set of keywords as arguments and returns geo-textual objects that are spatially and textually relevant to these arguments. The location component represents the local intent, and the keywords component describes the user requirements.

Due to the rich semantics of geographical space and the importance of geographical space to our daily lives, many different kinds of relevant spatial keyword query functionality may be envisioned. For example, considering a couple tourists who plan to visit Paris during the summer holidays, they may search for hotels that are kid-friendly and also close to some points-of-interests. A spatial keyword query like “find the nearest hotel to the Eiffel Tower that has a pool” could be issued, in which “hotel” and “pool” describes their requirements and the “Eiffel Tower” is the location component. A system that is able to process such queries should return a list of hotels with pools and close to the Eiffel Tower to them.

Spatial keyword queries are being supported in real-life applications. For example, in Google Maps the functionality “search nearby” allows users to retrieve points-of-interest around a specified location; in Twitter users can search for geo-tagged tweets satisfying their needs. Spatial keyword querying is also receiving increasing interest in the research community. Several proposals already exist for efficiently processing various types of spatial keyword queries. The existing studies can be categorized into different types according to the constraints specified in the query.

- (i) Some proposals (*e.g.*, [18, 34, 51]) view keywords as Boolean predicates, filtering out objects that do not contain the keywords and ranking the remaining objects based on their spatial proximity to the query. While other proposals (*e.g.*, [30, 72, 87]) combine spatial proximity and textual relevance using a linear ranking function. The textual relevance can be computed using various models, such as the vector space model [119] (*e.g.*, in work [72]), the Okapi-BM25 model [86] (*e.g.*, in work [27]), and the language model [110] (*e.g.*, in work [30]).
- (ii) The query location component also differs across existing studies. Some proposals (*e.g.*, [30, 34]) retrieve the top- k relevant geo-textual objects, and some proposals (*e.g.*, [24, 99]) retrieve the geo-textual objects within a certain range.
- (iii) In most work (*e.g.*, [30, 34]), the spatial keyword queries search for geo-textual objects represented by points. However, there also exist several proposals (*e.g.*, [24, 42, 63]) assuming the location of each object as a region (usually represented by a minimum bounding rectangle) instead of a point.

- (iv) In some proposals, the geo-textual objects are stored in spatial databases (*e.g.*, [18, 30, 34, 72]), and the R-tree [50] is usually used to index the objects. Some proposals (*e.g.*, [88]) study the spatial keyword querying over road networks. In spatial databases, the distance between objects or between query and objects is computed based on the Euclidean space, while on road networks it is computed based on road segment lengths.
- (v) The spatial keyword queries can also be categorized according to the search granularity. Most proposals retrieve single geo-textual objects (*e.g.*, [30, 34, 51, 88]). While some work [74, 113] retrieves a group of objects covering all the query keywords with the optimal cost. There also exist some proposals on trajectory keyword search (*e.g.*, [31, 116]), where each trajectory is formed by a sequence of geo-textual objects.

The existing work is still not sufficient to solve the problems in real-world applications. It can be noticed that most of the spatial keyword queries retrieve geo-textual objects independently. Single object granularity based spatial keyword querying have several limitations. First, these studies do not take into account the inter-object relationship during the retrieval. That is, the ranking of one object does not affect the ranking of another. However, if an object has many nearby neighbor objects that are relevant to the user's requirement, it would also be preferable to users and its ranking should be enhanced. Second, these studies cannot deal with the situations when user needs cannot be satisfied by a single object. For example, a tourist would be willing to do some sightseeing and then have dinner around the POIs. Such a query should be answered by a group of objects instead of one. Third, in some scenarios the user may intend to search for a route composed of several geo-textual objects. The existing studies on spatial keyword search is not able to process such queries as well.

In this thesis, three types of spatial keyword queries that retrieve the geo-textual objects beyond the single granularity are proposed to address the aforementioned problems of existing proposals.

1.3 Retrieving Geo-Textual Objects beyond the Single Granularity

The following spatial keyword queries are proposed: i) the top- k Prestige-based Spatial Keyword (k PSK) query that consider the inter-object relationship; ii) the Spatial Group Keyword (SGK) query where the resulting objects collectively answer the query; and iii) the Keyword-aware Optimal Route (KOR) query that searches for the optimal route such that it covers all the query keywords and satisfies the query budget constraint (such as the travel time and the financial budget).

1.3.1 Problems and Research Scope

1.3.1.1 Top- k Prestige-based Spatial Keyword Query

We often see that similar businesses locate near each other. Examples include restaurant districts (*e.g.*, China Town, Little Italy), bar districts, shopping streets, markets and bazaars (*e.g.*, farmer’s markets, antiques markets), and regions with a concentration of car sales businesses. This phenomenon seems to suggest that businesses benefit from locating near similar businesses. A possible explanation may be that this affords consumers easy access to a larger selection of products and services. For example, if no shoes in a store are attractive to a consumer, the presence of nearby shoe stores is desirable. Thus, we may expect that concentrations of similar businesses attract proportionally more consumers than do isolated stores. Put differently: co-located stores attract more customers to such an extent that this compensates for the increased competition.

Existing work treats geo-textual objects as independent when ranking them for a given query, which may not be appropriate because the co-location phenomenon is not taken into account. A relevant object whose nearby objects are also relevant to the query is preferable when compared to a relevant object without relevant nearby objects. For example, a user may prefer to visit a location with many shops instead of a location with only one shop in order to compare prices.

The problem is then how to integrate the benefits of co-location into the ranking of places in top- k spatial keyword queries. This is done by developing a notion of object “prestige” that takes into account the presence of nearby objects that are also relevant to a query. This notion of prestige is then used for the ranking of the query results. We call such a query as the top- k prestige-based spatial keyword (k PSK) query, that takes into account both location proximity and prestige-based text relevance (PR). The query retrieves a list of k objects ranked according to their spatial distances and PR scores with respect to the query.

One naive idea of computing the prestige of a given object is that, we compute the relevance scores of all objects in the neighborhood of the object, and then use the aggregated score of these objects as its prestige. However, it is hard to determine the size of the neighborhood. How to aggregate the scores efficiently given different queries is challenging as well. Therefore, we propose to adopt a PageRank-like model to capture the inter-object relationship. This is the first study on supporting this inter-object relationship in spatial keyword querying. However, in non-spatial keyword search, inter-document relationships have been exploited to improve effectiveness of document retrieval. For example, a PageRank-like algorithm is applied to the document similarity graph (built based on document similarity rather than web links), thus significantly improving the effectiveness of document retrieval [66] and question answers [32].

A further benefit of supporting this notion of prestige is that even if the description of an object does not contain the query terms, the object can still be identified as relevant. The intuition is similar to the translation model in information retrieval, where the relevant documents do not necessarily contain the query keywords, but have a high probability to be relevant to the query. This occurs if the object has a text description that matches those of nearby objects that in turn contain the query terms. For example, consider a query for “spring roll” and two close objects with descriptions “best Chinese restaurant in Boston” and “Chinese restaurant offering spring rolls.” The two descriptions are similar, and the latter contains the query term. Thus although the first object’s description does not contain the query term, it is highly likely to be relevant to the query.

To illustrate the effect of supporting prestige-based object relevance, consider the query “shoes” at location q in Figure 1.2. Circles represent shops selling shoes or jeans, with centers representing locations and areas representing relevance to the query. Existing spatial keyword search techniques (*e.g.*, [30]) rank p_5 as the top-1 result since p_5 is relevant and closest to q . However, p_4 is more attractive because it has more nearby shops that are relevant to the query and also is close to the query location.

Figure 1.2: Prestige propagation example

1.3.1.2 Spatial Group Keyword Query

Most of the existing proposals focus on searching for single objects, where a list of geo-textual objects that are textually relevant to the query keywords and close to the query location or within the query range are returned as the results. Unfortunately, sometimes user needs may exist that are not easily satisfied by a single object, but where groups

of objects may combine to meet the user needs. We propose the Spatial Group Keyword (SGK) query which returns a group of objects that satisfy the user requirements collectively.

A group of objects may better satisfy the user requirements. For example, a tourist may have particular shopping, dining, and accommodation needs that may best be met by several geo-textual objects. As another example, a user may wish to set up a project consortium of partners within a certain spatial proximity that combine to offer the capabilities required for the successful execution of the project.

To address the need for such collective answers to spatial keyword queries, we assume a database of geo-textual objects and then consider the problem of how to retrieve a group of spatial objects that collectively meet the user's needs given as location and a set of keywords: 1) the textual description of the group of objects must cover the query keywords, 2) the objects are close to the query point, and 3) the objects in the group are close to each other.

Specifically, given a set of geo-textual objects D , and a *Spatial Group Keyword* (SGK) query $q = (\lambda, \psi)$, where λ is a location and ψ is a set of keywords, we consider two instantiations of the SGK query.

- (i) We aim at finding a group of objects χ that cover the keywords in q such that the sum of their spatial distances to the query is minimized.
- (ii) We aim at finding a group of objects χ that cover the keywords in q such that the sum of the maximum distance between an object in χ and query q and the maximum distance between two objects in χ is minimized.

It turns out that the subproblems corresponding to these two instantiations are both NP-complete.

1.3.1.3 Keyword-aware Optimal Route Query

We also consider spatial keyword search on the road networks and we aim at retrieving the routes formed by a sequence of geo-textual objects. Identifying a preferable route in a road networks has been supported in real applications, such as Baidu Lvyou⁵ and Yahoo Travel⁶. However, the routes that they provide are collected from users and are thus pre-defined. This is a significant deficiency since there may not exist any pre-defined route that meets the user needs. There also exist some research studies (*e.g.*, [68,69,94]) for trip planning or route search, and they are often insufficient in offering the flexibility for users to specify their requirements on the route.

⁵<http://lvyou.baidu.com>

⁶<http://travel.yahoo.com>

Consider a user who wants to spend a day exploring a city. She is not familiar with the city and she might pose such a query: “Find the most popular route to and from my hotel such that it passes by a *shopping mall*, a *restaurant*, and a *pub*, and the time spent on the road in total is within 4 hours.”

The example query above has two hard constraints: 1) the points of interests preferred by the user, as expressed by a set of keywords that should be covered in the route (*e.g.*, “shopping mall”, “restaurant” and “pub”); 2) a budget constraint (*e.g.*, travel time) that should be satisfied by the route. The query aims to identify the optimal route under the two hard constraints, such that an objective score is optimized (*e.g.*, route popularity [25]). Note that route popularity can be estimated by the number of users traveling a route, obtained from the user traveling histories recorded in sources such as GPS trajectories or Flickr photos [25]. In general, the budget constraint and the objective score can be of various different types, such as travel duration, distance, popularity, travel budget, etc. We consider two different attributes for budget constraint and objective score because users often need to balance the trade-off of two aspects when planning their trips. For example, a popular route may be quite expensive, or a route with the shortest length is of little interests. In the example query, it is likely that the most popular route requires traveling time more than 4 hours. Hence, a route searching system should be able to balance such trade-offs according to users’ different preferences.

We refer to the aforementioned type of queries as *Keyword-aware Optimal Route* query, denoted as KOR. Formally, a KOR query is defined over a graph $\mathcal{G}$, and the input to the query consists of five parameters, v_s , v_t , ψ , Δ , and f , where v_s is the source location of the route in G , v_t is the target location, ψ is a set of keywords, Δ is a budget limit, and f is a function that calculates the objective score of a route. The query returns a path R in G starting at v_s and ending at v_t , such that R minimizes $f(R)$ under the constraints that R satisfies the budget limit Δ and passes through locations that cover the query keywords in ψ . To the best of our knowledge, none of the existing studies on trip planning or route search (*e.g.*, [68, 69, 94]) is applicable for KOR queries. Furthermore, the problem of solving KOR queries can be shown to be NP-hard by a reduction from the weighted constrained shortest path problem [38]. It can also be viewed as a generalized traveling salesman problem [52] with constraints.

1.3.2 Approaches and Methodology

1.3.2.1 Top- k Prestige-based Spatial Keyword Query

In our proposal for prestige-based relevance (denoted as “PR”) of objects to queries, the PR score of an object is affected by the PR scores of its neighbors. The notion of prestige involves mutual reinforcement: the prestige of one place is affected by the

prestige of nearby places. This motivates us to employ a PageRank-like random walk mechanism for the propagation of prestige.

Recall the query “shoes” at location q in Figure 1.2. Conceptually, the PR scores of the objects are computed as follows: We build a graph with the objects as the nodes. Two nodes are connected by an edge if the objects are close and their text descriptions are similar. Myriads of random surfers are initially placed at the nodes that contain the query term “shoes” (*i.e.*, p_4, p_5, p_6, p_7). The number of random surfers at a node is proportional to the textual relevance between the node and the query, which is the initial prestige-based relevance of the node. At each step, each random surfer either moves to an adjacent node following a link in the graph with a certain walking probability (depending on the distance between the nodes), or it randomly jumps to the initial set of nodes containing “shoes” without following any link, again with a certain probability (depending on the well-known damping factor [14]). The expected percentage of surfers at each node eventually converges, and the converged percentage of surfers at a node represents the PR score of the node.

The concept of PR is inspired by the concept of personalized PageRank [10, 56], where a subset of web pages share the initial prestige uniformly (rather than all web pages as in PageRank), and its applications to keyword search in Entity-Relation graphs [5, 19].

The above random walk process has unique features that render a direct application of PageRank [14] inadequate. PageRank is used to compute the objects’ global importance, which is query-independent, and it has no preferences for any particular nodes. Our problem is very different: each query has a set of preference objects based on the initial relevance scores, and thus random surfers start from this set of objects and jump to them. For example, given a “jeans” query at point q , p_1 is the best result rather than p_4 , which is best if the query is for “shoes.” The personalized PageRank can allow users to define their own notion of importance for each individual query, and thus it can better interpret our problem. Note that the personalized PageRank is also adopted in problems relevant to the k PSK query, such as ObjectRank [5].

The k PSK query is expensive to compute, especially due to the PR score computation. A straightforward approach to answering the k PSK query is to adapt the algorithms for computing Personalized PageRank Vectors (PPVs) [5, 10, 19, 45, 56] to compute the PR scores of all objects in the spatial object graph, to use an R-tree for computing spatial distances, and then to combine the two scores. However, this solution is expensive. First, it is expensive to compute PR scores for a large graph at query time using existing algorithms for computing PPVs. Second, it is impractical to pre-compute PPVs for each node in terms of either pre-computation time or the storage requirements [56], because $|V|^2$ space is needed to store the PPVs for all objects. Third, it is a waste to compute the spatial distance and PR scores of all objects and then rank them to find the top- k results.

We note that the spatial object graph has unique properties that render it different from the web link graph [10, 56] and the entity-relation graph [5, 19].

- Similar spatial objects often co-locate geographically. For example, shops often co-locate, as do, *e.g.*, bars. Therefore, spatial objects tend to naturally form subgraphs.
- The number of nodes in a subgraph is constrained by geography. Thus, subgraphs will be of relatively modest size.

Properties like these enable us to develop approaches that speed up PR scoring. We propose two novel algorithms for the efficient computation of the k PSK query.

ES-EBC (Early stop extended bookmark coloring): We prove that if the distance between a node and the query point exceeds a certain threshold, the node will not affect the PR scoring of the top- k objects. Therefore, we need only consider nearby nodes when propagating PR, which speeds up the PR scoring substantially. We also show how to estimate lower and upper bounds on the PR score of each object in each iteration during scoring. Utilizing these bounds, we derive conditions for when further iterations will not change the ranking order of the top- k objects and we stop iterating.

S-EBC (Subgraph-based extended bookmark coloring): We propose an approximate solution to PR scoring with performance guarantees. We organize spatial objects and their text descriptions using the external memory IR-tree [30]. Hence, the spatial objects are grouped into subgraphs based on their locations, with each subgraph corresponding to a leaf node of the IR-tree. We prove that the PR scores of the nodes in a subgraph can be computed by PR propagation within the subgraph and contributions from the border objects that connect the subgraph with other subgraphs. This enables PR scoring w.r.t. subgraphs rather than on the whole graph. Next, we propose a novel approach to estimating an upper bound on the PR scores of the objects in each subgraph. This bound together with the distance of a subgraph to the query is used to choose which subgraphs to process and in which order.

1.3.2.2 Spatial Group Keyword Query

Specifically, we study two instantiations of the spatial group keyword query, as described in Section 1.3.1.2, both of which are NP-hard.

The first subproblem can be reduced from the weighted set cover problem. We propose a greedy algorithm that provides an approximate solution to the problem. This algorithm utilizes a spatial-keyword index such as the IR-tree [30] to prune the search space. The algorithm has a provable approximation bound. Based on the assumption that in some applications, the number of keywords in a query q may not be large, we also propose two exact algorithm that exploit dynamic programming. They avoid enumerating the combinations of data objects in the database. Rather, the first algorithm

enumerates the query keywords and the second one enumerates the candidate objects to generate the optimal group. We also utilize a spatial-keyword index to further improve the performance of the exact algorithm by exploiting a series of pruning strategies.

The second subproblem can be reduced from the 3-SAT problem. We develop three approximation algorithms based on a spatial-keyword index with provable approximation bounds. The first approximation algorithm has a 3-approximation ratio, while the second algorithm and the third algorithm have a 1.8-approximation ratio. We also develop two exact algorithm that exploits a spatial-keyword index and the geometric property to prune the search space.

1.3.2.3 Keyword-aware Optimal Route Query

Since answering a KOR query is NP-hard, this leads to an interesting question: is it possible to derive efficient solutions for KOR queries?

Due to the hardness of answering KOR queries, we answer the aforementioned question affirmatively with three approximation algorithms. The first approximation algorithm has a performance bound and is denoted by **OSScaling**. In **OSScaling**, we first scale the objective value of every edge to an integer by a parameter ϵ to obtain a scaled graph denoted by $\mathcal{G}_S$. Specifically, in the scaled graph $\mathcal{G}_S$, each partial route is represented by a “label”, which records the query keywords already covered by the partial route, the scaled objective score, the original objective score, and the budget score of the route. At each node, we maintain a list of “useful” labels corresponding to the routes that go to that node. Starting from the source node, we keep creating new partial routes by extending the current “best” partial route to generate new labels, until all the potentially useful labels on the target node are generated. Finally, the route represented by the label with the best objective score at the target node is returned.

We prove that the algorithm returns routes with objective scores no worse than $\frac{1}{1-\epsilon}$ times of that of the optimal route. The worst case complexity of **OSScaling** is polynomial with $\frac{1}{\epsilon}$, the budget constraint Δ , the number of edges and nodes in $\mathcal{G}$, and it is exponential in the number of query keywords, which is usually small in our targeted applications, as it is well known that search engine queries are short, and an analysis on a large Map query log [106] shows that nearly all queries contain fewer than 5 words.

Our second algorithm improves on the algorithm **OSScaling**, which is referred to as **BucketBound**. It also returns approximate solutions to KOR queries with performance guarantees. However, it is more efficient than **OSScaling**. The algorithm can always return a route whose objective score is at most β ($\beta > 1$ is a parameter) times of the one found by **OSScaling**. The algorithm divides the traversed partial routes into different “buckets” according to the best possible objective scores they can achieve. This enables us to develop a novel way to detect if a feasible route (covering all query keywords and satisfying

the budget constraint) is in the same bucket with the one found by **OSScaling**. When we find a feasible route that falls in the same bucket as the route found by **OSScaling**, we return it as the result.

Finally, we also present a greedy approach for the problem. From the starting location, we keep selecting the next location greedily, taking into account all the three constraints in the **KOR** query. This is repeated until we reach the target location. This algorithm is efficient, although it may generate a route that violates the two hard constraints of **KOR**: covering all query keywords and satisfying the budget constraint.

1.4 Summary of Contributions

In this thesis, the problem of efficient processing spatial keyword queries beyond the single geo-textual object granularity is addressed. Specifically, we propose three different approaches.

First, we introduce a novel query, named **kPSK**, to retrieve the top- k relevant geo-textual objects considering the inter-object relationship. In such a query, an object can gain “prestige” from their neighbors, and thus objects with many neighbors that are also relevant to the query have high rankings. It is shown that **kPSK** query can better satisfy user needs. We adapt the bookmark coloring algorithm (BCA) [10] to do the PR scoring. Based on this algorithm, we propose two methods to answer the **kPSK** query: a) the ES-EBC algorithm that extends BCA and utilizes the threshold algorithm [39]; b) the S-EBC algorithm that utilizes the modified IR-tree [30] to index the geo-textual objects and partition the object graph into subgraphs. The PR scores can be computed in different subgraphs separately utilizing some pre-processing results.

Second, we propose the novel **SGK** query, that retrieves a group of geo-textual objects χ such that they collectively cover all the query keywords and the cost computed by a function is minimized. We study two instances of the **SGK** query, both of which are NP-hard. The first type of cost function computes the total distances of all the objects in χ . For this type of query, we propose one approximate algorithm and three exact algorithms using dynamic programming approach. The second type of cost function computes the sum of the maximum distance between the query and an object in χ and the maximum distance between any two objects in χ . We propose three approximation algorithms and two exact algorithms utilizing the IR-tree indexing structure [30].

Third, we define the novel **KOR** query, that searches for the optimal route consisting a sequence of geo-textual objects such that the route covers all the query keywords and satisfies the query budget constraint. The problem of answering the **KOR** query is proved to be NP-hard. We propose two approximation algorithms with performance guarantee and one heuristic algorithm. We also tackle the problem of answering the top- k keyword-aware optimal route query by extending the algorithms for the **KOR** query.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 summarizes related literatures. Chapter 3 introduces the algorithms and experimental results for the *kPSK* query. Chapter 4 presents the algorithms and empirical studies for the **SGK** query. Chapter 5 elaborates the algorithms and experimental studies for the **KOR** query. Finally, Chapter 6 concludes the thesis.

Chapter 2

Literature Survey

Spatial keyword querying incorporates techniques from spatial databases and information retrieval. Most proposals on spatial keyword querying combine spatial indices with text indices (loosely or in a hybrid manner) to index the geo-textual objects and to process the queries. This chapter reviews state-of-the-art of research studies relevant to spatial keyword search.

2.1 Mapping Locations with Web Contents

Geo-textual objects that possess both a geographical location and a textual description are gaining in prevalence. As discussed in Section 1.1, they can be obtained from commercial map services (*e.g.*, Google Maps) and from websites containing geo-tagged contents (*e.g.*, Flickr and Foursquare). Although geo-textual objects can be extracted directly from websites, there exist many proposals on geographical retrieval studying the problem of mapping the geography with web contents, which enrich the availability of geo-textual objects.

Ding et al. [36] introduce techniques for automatically computing the geographical scope of web resources. In this proposal, the geographical scope of a web resource w is defined as the geographical area that the creator of w intends to reach. They use a three-level location hierarchy to represent the United States, and both the textual content of the resources and the geographical distribution of hyperlinks to the resources are utilized to map the locations in the hierarchy to the web resources. However, because they adapt a statistical approach in their algorithms, their method cannot meaningfully work on individual web resources that contain very few places mentioned.

McCurley [80] investigates different approaches to discovering geographic context for web pages, and describes a navigational tool for browsing web resources by geographic proximity. He describes geographic indicators found in pages, including the URL, the language used, phone numbers, ZIP codes, geographic feature names, and the context

derived from hyperlinks. Location names found in the text may be looked up in gazetteers if they are places, or in the White Pages directory to extract their addresses. Hyperlinks may be followed to see if the linked pages have a strong geographic association that could be reflected back. However, his approach is not easy to apply to areas where the online information may not be complete, because the approach heavily depends on information such as place names, postal tracts and phone directories that are free and available online.

Amitay et al. [2] propose a system for associating geography with web pages. Given a web page, the goal is to identify all geographic mentions in web pages, to assign a geographic location and confidence level to each, and to derive a focus (or foci) for the entire page. They utilize several public gazetteers on the web and design algorithms to score the taxonomy nodes for a given page to find the focus of that page.

Silva et al. [95] describe methods for determining the scope of web documents in the Portuguese tumba! web search engine (not available now). They first transform web documents to a structured XML/RDF format. Then, they propose to recognize geographical references over the web documents and assign geographical scopes to the documents through a graph ranking algorithm.

2.2 Spatial Indices and Queries

Spatial keyword querying utilizes techniques in spatial databases. We first briefly review several popular spatial indices, and then we present the existing studies on processing spatial queries.

2.2.1 Spatial Indices

Jon Louis Bentley proposes the k -dimensional tree (denoted by k -d tree) [8], which can be used to index spatial points. Generally, it is a space-partitioning data structure for organizing points in a k -dimensional space. The k -d tree is a binary tree in which every 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 parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k -dimensions, with the hyperplane perpendicular to that dimension's axis.

R-tree [50] was proposed by Antonin Guttman in 1984, and it is the most popular method for indexing spatial data. Spatial objects (*e.g.*, shapes, lines, and points) are grouped using minimum bounding rectangles (MBR). The R-tree nodes are organized

Figure 2.1: An example of an R-tree

in disk pages, and each node can have a variable number of entries (up to some predefined maximum, and usually above a minimum fill). Each non-leaf node stores the identifiers and MBRs of all its child nodes. Each leaf node stores the spatial objects within this entry. The search, update, insertion and the deletion of spatial objects in the R-tree is quite efficient. When a new entry is inserted into a overflowing node containing m entries, it is necessary to split the $m + 1$ entries into two nodes. Guttman devises two cost algorithms in order to lead to the smallest increase in size, *i.e.*, the linear-cost and the quadratic-cost algorithms. Figure 2.1 illustrates the example of an R-tree that indexes five rectangles.

Figure 2.2: An example of an R^+ -tree

Sellis et al. [92] propose the R^+ -tree, which is a variation to the R-tree that avoids overlapping rectangles in intermediate nodes of the tree. The “coverage” and “overlap” significantly affect the performance of an R-tree. Coverage is the entire area to cover all related rectangles. Overlap is the entire area which is contained in two or more nodes. Minimal coverage reduces the amount of “dead space” (empty area) which is covered by the nodes of the R-tree. Minimal overlap is even more critical for the access time than

minimal coverage, since it reduces the set of search paths to the leaves. Efficient search requires both minimal coverage and overlap. The main idea of the R^+ -tree is to achieve zero overlap among intermediate node entries by allowing partitions to split rectangles. Due to the zero overlap, a single path is followed during the search, and thus fewer nodes are visited than using the R -tree. However, since rectangles may be split and stored in multiple nodes, it occupies more space than does the R -tree. As shown in the empirical study, the R^+ -tree achieves up to 50% savings in disk accesses compared to an R -tree when searching files of thousands of rectangles. Figure 2.2 shows the example of an R^+ -tree. R_4 is split into two rectangles and they are stored in nodes A and B separately.

Figure 2.3: An example of an R^* -tree

Beckmann et al. [7] propose the R^* -tree, which differs from the R -tree in the following aspects: 1) when inserting, the R^* -tree uses different strategies for leaf and non-leaf nodes. In the leaf nodes level, the entry that needs least overlap enlargement to include the new object is selected, while for the inner nodes, the entry that needs least area enlargement to include the new object is selected; 2) when splitting, the R^* -tree uses a topological split that chooses a split axis based on perimeter, and then minimizes overlap; 3) the R -tree is highly susceptible to the order in which the entries are inserted, and thus the R^* -tree tries to reinsert objects and subtrees into the tree, to obtain better tree structures than the original one built based on the insertion order. An R^* -tree as shown in Figure 2.3 may be created to index the five rectangles. Because better insertion and deletion strategies are adopted, the overlap area is reduced compared with using R -tree.

Finkel et al. [43] propose the Quad-tree, which is a tree data structure in which each internal node has exactly four children. The Quad-tree is most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. Each Quad-tree cell has a maximum capacity, and during insertion if the maximum capacity is reached, the cell is split into four lower-level cells. The work [59] compares the performance of the R -tree and the Quad-tree using spatial objects. According to this study, a Quad-tree could be recommended for update-intensive applications using simple polygon

Figure 2.4: An example of a Quad-tree

geometries, high concurrency update databases, or when specialized masks such as touch are frequently used in queries. However, users have to fine-tune the tiling level to obtain best performance. The R-tree, which does not require any such tuning, could be used in all other cases to obtain nearly equivalent or better performance. Figure 2.4 illustrates the example of a Quad-tree that indexes eight spatial points, where the maximum capacity is set to 2.

Figure 2.5: An example of a grid index

The grid-based index partitions the space into several disjoint cells. Generally, the construction of such a index entails allocation of relevant objects to their positions in the grid, then creating an index of object identifiers vs. grid cell identifiers for rapid access. The advantage is that the structure of the index can be created first, and the

insertion of new spatial objects requires no change to the index structure. The grid-based index is “space-driven”, as opposed to “data-driven” indices such as the R-tree and the Quad-tree [85]. Nievergelt et al. [82] first propose to use the grid file for accessing multi-key records, and it has proven to be useful when applied to geometric data [96]. Figure 2.5 illustrates the example of a simple 3×3 grid index that indexes eight spatial points.

The space filling curves are also useful in indexing and accessing spatial objects [41], such as the Z-order curve [81] and the Hilbert curve [53]. The space filling curves are used to impose a linear ordering on the spatial objects to be indexed. After the data are sorted according to the ordering, any one-dimensional data structure can be used such as the B-tree [29]. Specially, the resulting Z-ordering can equivalently be described as the order one would get from a depth-first traversal of a Quad-tree.

2.2.2 Queries over Spatial Databases and Road Networks

The most important and widely used spatial queries include the k nearest neighbor (k NN) query and the range query in spatial databases and road networks, which have been studied for years.

The most influential work on k NN queries is proposed by Roussopoulos et al. [89]. The authors exploit the R-tree [50] to index the points, and devise a branch-and-bound R-tree traversal algorithm to find the k nearest neighbors for a given query. The candidate R-tree nodes that contain potential k nearest neighbors are maintained in a priority queue. The nodes are organized in order of either the **MINDIST** or the **MINMAXDIST** metric, where **MINDIST** computes the minimum distance between the query and an R-tree node, and **MINMAXDIST** computes the minimum of the maximum possible distance between the query and an R-tree node. **MINDIST** produces the most optimistic possible ordering, whereas **MINMAXDIST** produces the most pessimistic ordering that ever need to be considered. Hjaltason and Samet [54] propose an incremental nearest neighbor algorithm using only **MINDIST** that is applicable to a large class of hierarchical spatial data structures. Experiments show that the incremental algorithm significantly outperforms the k nearest neighbor algorithm [89] for distance browsing queries in a spatial database that uses the R-tree as a spatial index.

The work [9] first tackles the problem of range searching on multidimensional data, where the k -d tree is exploited in query processing. The spatial range query retrieves a set of objects that overlap or are within the specified query window. All the spatial indices support efficient spatial range querying. For example, a range query is processed recursively from root node to the leaf nodes in an R-tree [50]. The nodes that do not touch the query window is filtered out. The search process is invoked on the subtrees

whose root nodes overlap with the query, and when the leaf nodes are reached, each entry is checked if it is a result.

On spatial networks, the distance between two spatial objects is computed as the road segment length of their shortest path instead of the Euclidean distance between them, and thus the algorithms for queries on spatial databases cannot be applied directly for processing queries over the road networks. Papadias et al. [84] study four types of spatial queries, *i.e.*, nearest neighbors, range search, closest pairs and e-distance joins in spatial network database. They assume a digitization process that generates a modeling graph from an input spatial network. The graph vertices include the network junctions, the starting/ending point of a road segment, and additional points depending on the application. The graph edges preserve the connectivity in the original network. A network R-tree is proposed to index the MBRs of poly-lines, in which each leaf entry contains a pointer to the disk page storing the corresponding polyline. Based on the graph and the network R-tree, the four types of spatial queries are processed exploring the spatial properties of the network.

2.3 Information Retrieval Indices and Models

2.3.1 Inverted Files and Signature Files

In many state-of-the-art large-scale IR systems such as web search engines, the inverted file [120] is used to index the documents, which is the most efficient index structure for text retrieval as shown in the study [120]. The inverted file is also referred to as postings file, and it stores a mapping from contents, such as words or numbers, to its locations in a database file, or in a document or a set of documents. Table 2.1 demonstrates an example of the inverted file structure. The left table shows the document corpus, where each document contains some keywords. The right table shows the postings lists, each of which stores the documents that contain the corresponding word.

Zobel et al. [120] compares the performance of inverted files and other indexing techniques such as the signature files [40]. The idea behind signature files is to create a quick and dirty filter that will keep all the documents that match to the query and hopefully a few ones that do not. The way this is done is by creating for each file a signature, typically a hash coded version. As demonstrated by the empirical results in the work [120], signature files are not competitive with the inverted file for information retrieval queries. Since in most cases this structure is inferior to inverted files in terms of speed, size and functionality, it is not used widely. A bitmap in which each bit represents the occurrence of a keyword can be viewed as a special signature file.

documents	words
d_1	t_1, t_2
d_2	t_2, t_3
d_3	t_1, t_3
d_4	t_1, t_5
d_5	t_2, t_4
d_6	t_4, t_6
d_7	t_4, t_5
d_8	t_1, t_5

2.1.a Document corpus

words	documents
t_1	d_1, d_3, d_4, d_8
t_2	d_1, d_2, d_5
t_3	d_2, d_3
t_4	d_5, d_6, d_7
t_5	d_4, d_7, d_8
t_6	d_6

2.1.b Postings lists

Table 2.1: An example of an inverted file

2.3.2 Information Retrieval Models

There are a variety of retrieval models proposed to meet different information retrieval needs. The most popular ones include the vector space model [119], the Okapi BM25 model [86], and the language model [110]. They are used in many applications and show excellent performance (*e.g.*, [16, 17]). Most spatial keyword queries that consider the textual relevance adopt them to compute the ranking scores of geo-textual objects (*e.g.*, [27, 30, 72]).

The vector space model is an algebraic model for representing text documents as vectors. Both documents and queries are represented as vectors, where each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is non-zero. Several different ways of computing these values, referred to as term weights, have been developed. One of the best known schemes is TF-IDF weighting, where TF stands for “term frequency” and IDF stands for “inverse document frequency”. Then, vector operations can be used to compare documents with queries, and usually the cosine of the angle between the vectors is computed to represent the similarity between the documents and queries. A popular variation of this model is proposed by Zobel et al. [119]. Given a query $\mathbf{q}$ and a document $\mathbf{d}$, the text similarity $\text{Sim}(\mathbf{q}, \mathbf{d})$ between them can be computed as follows:

$$\begin{aligned} \text{Sim}(\mathbf{q}, \mathbf{d}) &= \frac{\sum_{t \in \mathbf{q} \cap \mathbf{d}} w_{\mathbf{q},t} w_{\mathbf{d},t}}{W_{\mathbf{q}} W_{\mathbf{d}}}, \text{ where} \\ w_{\mathbf{q},t} &= \ln\left(1 + \frac{N}{f_t}\right), \quad w_{\mathbf{d},t} = 1 + \ln(tf_{t,\mathbf{d}}) \\ W_{\mathbf{q}} &= \sqrt{\sum_t w_{\mathbf{q},t}^2}, \quad W_{\mathbf{d}} = \sqrt{\sum_t w_{\mathbf{d},t}^2} \end{aligned} \quad (2.1)$$

Here N is the number of documents in the whole collection, f_t is the number of documents containing the term t , and $tf_{t,\mathbf{d}}$ is the frequency of term t in $\mathbf{d}$. The term $W_{\mathbf{q}}$ can

be neglected as it is a constant for a given query and does not affect the rankings of documents. Finally, $w_{\mathbf{q},t}$ captures the IDF of term t in the collection, and $w_{\mathbf{d},t}$ captures the TF of term t in $\mathbf{d}$. Vector space model is used for spatial keyword querying by Li et al. [72].

The vector space model favors short documents, the Okapi BM25 Model [86] takes into account the document length to overcome this problem. Given a query $\mathbf{q}$ and a document $\mathbf{d}$, the text similarity between them $\text{Sim}(\mathbf{q}, \mathbf{d})$ is computed as follows:

$$\begin{aligned} \text{Sim}(\mathbf{q}, \mathbf{d}) &= \sum_{t \in \mathbf{q} \cap \mathbf{d}} w_{\mathbf{q},t} w_{\mathbf{d},t}, \text{ where} \\ w_{\mathbf{q},t} &= \ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \frac{(k_3 + 1)tf_{t,\mathbf{q}}}{k_3 + tf_{t,\mathbf{q}}} \\ w_{\mathbf{d},t} &= \frac{(k_1 + 1)tf_{t,\mathbf{d}}}{K_{\mathbf{d}} + tf_{t,\mathbf{d}}} \\ K_{\mathbf{d}} &= k_1((1 - b) + b\frac{W_{\mathbf{d}}}{W_A}) \end{aligned} \quad (2.2)$$

Here N , f_t , and $tf_{t,\mathbf{d}}$ are as defined in the vector space model; k_1 , b , and k_3 are parameters that are set to 1.2, 0.75, and ∞ , respectively, as defined in [86] (the expression $\frac{(k_3+1)tf_{t,\mathbf{q}}}{k_3+tf_{t,\mathbf{q}}}$ is then equivalent to $tf_{t,\mathbf{q}}$); and $W_{\mathbf{d}}$ is the document length of $\mathbf{d}$ and W_A is the average document length in the collection. Okapi BM25 is used for spatial keyword querying by Christoforaki et al. [27].

A language model is associated with a document in a collection. Given a query $\mathbf{q}$, retrieved documents are ranked based on the probability that the document's language model would generate the terms of the query. The method to use language models in information retrieval is the query likelihood model. Usually, the smoothing is necessary, assigning some of the total probability mass to unseen words for a document. Given a query $\mathbf{q}$ and a document $\mathbf{d}$, the text similarity between them $\text{Sim}(\mathbf{q}, \mathbf{d})$ is computed as follows, using Jelinek-Mercer smoothing [110]:

$$\begin{aligned} \text{Sim}(\mathbf{q}, \mathbf{d}) &= \prod_{t \in \mathbf{q}} ((1 - \lambda)P_{ml}(t|\mathbf{d}) + \lambda P_{ml}(t|\mathbf{Coll})), \text{ where} \\ P_{ml}(t|\mathbf{d}) &= \frac{tf_{t,\mathbf{d}}}{\sum_{t' \in \mathbf{d}} tf_{t',\mathbf{d}}}, P_{ml}(t|\mathbf{Coll}) = \frac{tf_{t,\mathbf{Coll}}}{\sum_{t' \in \mathbf{Coll}} tf_{t',\mathbf{Coll}}} \end{aligned} \quad (2.3)$$

Here $P_{ml}(t|\mathbf{d})$ is the maximum likelihood estimate of word t in $\mathbf{d}$; $P_{ml}(t|\mathbf{Coll})$ is the maximum likelihood estimate of word t in the collection $\mathbf{Coll}$; and λ is the smoothing parameter. The language model is used for spatial keyword querying by Cong et al. [30].

2.4 Spatial Keyword Indices and Queries

A spatial keyword query is defined over a database of geo-textual objects $\mathcal{D}$. An object $o \in \mathcal{D}$ has two attributes: $\langle \psi, \lambda \rangle$, where ψ is a text description and λ encodes an accurate geo-location. Typically, it takes a location (a point or an area) and a set of keywords as arguments and returns geo-textual objects that satisfy these arguments. Both a spatial and a text index are utilized to process the spatial keyword query. We first talk about the classification of the existing spatial keyword indices in Section 2.4.1, and then we introduce the existing studies on spatial keyword querying and give more analysis and examples of the indices utilized in these studies in Section 2.4.2.

2.4.1 Spatial Keyword Indexing Classification

A geo-textual object has both a location descriptor and a text description. In order to index such objects, a spatial keyword (or spatio-textual) index incorporates the techniques of indexing spatial objects and indexing documents. It can be classified according to the following aspects: the spatial indexing scheme used, the text index employed, and the combination manner of the spatial and text indices.

Spatial indexing scheme. Several spatial indices are introduced in Section 2.2.1, and most of them have been adopted for building a spatio-textual index. For example, R-tree is used in the proposals [30, 72], R*-tree is used in the work [51, 117], Quad-tree is used in the studies [111, 115], grid is used by Vaid et al. [99], and the space filling curve is adopted in the work [24, 27].

Text indexing scheme. As introduced in Section 2.3.1, the inverted file and the signature file are the most popular text indexing techniques. The existing spatial keyword search proposals either use the inverted file or the signature file. For example, the proposals [30, 51, 72, 117] use the inverted file to index the text descriptions of objects, while the studies [18, 34] use the signature file.

Combination manner. Existing studies propose several ways of combining the spatial and the text index. A naive way of building a spatial keyword index is to build the spatial and the text index separately. The geo-textual objects are indexed twice, where the spatial component of the objects is indexed by the spatial index and the textual part is indexed by the text index. This method is studied as a baseline approach in the work [117].

However this simple way has poor performance (to be shown in Section 2.4.2.1). Therefore, a better hybrid spatial keyword index that combines the two types of indices are proposed and studied in several proposals. The two types of indices can either be combined loosely (*e.g.*, [99, 117]), or be combined in a tight manner (*e.g.*, [30, 34]).

The loose combination has two different manners: either spatial-first or text-first. In the spatial-first index, the top-level is a spatial indexing structure (*e.g.*, an R-tree) and the bottom level (*e.g.*, an R-tree leaf node) consists of the text indices (*e.g.*, inverted files). A text-first loose combination index usually employs the inverted file as the top-level index and then organize the objects in each inverted list in a spatial structure, which can be an R-tree, a grid etc. The two types of indices are studied and evaluated in several proposals (*e.g.*, [99, 117]).

The tight combination index combines a spatial and a text index tightly such that both types of information can be used to prune the search space simultaneously during query processing. One of the most well-known such type of spatial keyword index is the IR-tree, proposed by Cong et al. [30].

All existing spatial keyword indices can be categorized into some type as discussed above. More analysis and examples of spatial keyword indexing techniques are to be given in Section 2.4.2.

2.4.2 Spatial Keyword Queries

We first review and classify the existing studies on spatial keyword queries retrieve geo-textual objects at the granularity of single objects in Section 2.4.2.1. We next introduce some proposals on querying beyond the single object granularity in Section 2.4.2.2. Finally, we describe other types of spatial keyword queries in Section 2.4.2.3.

2.4.2.1 Single Granularity-based Queries

In spatial databases, the arguably most fundamental queries are range queries and k nearest neighbor queries. In information retrieval, queries may be Boolean, requiring results to contain the query keywords, or ranking-based, returning the k documents that have the highest rankings according to a text similarity function as discussed in Section 2.3.2.

A standard single granularity-based spatial keyword query involve all the conditions on the spatial and textual aspects of geo-textual objects. Generally, a spatial keyword query Q is in format of $\langle \psi, \lambda \rangle$, where ψ is a set of keywords describing the user requirements and λ is a location descriptor such as a point or an area. As a result, based on the spatial and textual predicates specified in a query, existing proposals can be categorized into the following four types: the Boolean range query (BRQ) that retrieves a set of objects each of which satisfies the Boolean keyword predicate and is in the query range, the Boolean k nearest neighbor query (BkQ) that retrieves k objects each of which satisfies the Boolean keyword predicate ranked according to their distances to the query, the ranking range query (RRQ) that retrieves objects in the query range ranked according

to the text similarity, and the hybrid k nearest neighbor query (HkQ) that retrieves k objects ranked according to a score that takes into consideration both spatial proximity and text relevance.

1. Boolean Range Query. Zhou et al. [117] tackle the problem of answering BRQ. They adopt the inverted file for the text index and the R*-tree for the spatial index. They first propose to build the spatial and the text index separately as a baseline, *i.e.*, web pages are indexed separately twice, once by R*-tree and once by inverted files. As for the example dataset in Figure 2.6, an R-tree like Figure 2.6.a and an inverted file like Table 2.1.b are created separately.

2.6.a Object locations

2.6.b Object descriptions

Figure 2.6: An example dataset containing 8 geo-textual objects

When a query comes, the objects are filtered either first by the spatial index and then verified by the text index or vice versa. The main advantage of this method is the ease of maintaining two separate indices. However, the main performance bottleneck lies in the number of candidate objects generated during the filtering stage. If spatial filtering is done first, many objects may lie within a query's spatial extent, but very few of them are relevant to the query keywords. This increases the disk access costs by generating a large number of candidate objects. The subsequent stage of keyword filtering becomes expensive. The same is also true if keyword filtering is done first.

To improve the efficiency, they propose another two methods: a spatial-first index—first R*-tree then inverted file, where an R*-tree is built on all web pages and in each leaf MBR the inverted file is created, and a text-first index—first inverted file then R*-tree, where each keyword points to an R*-tree indexing the web pages containing that keyword.

In the spatial-first index, the top-level is a spatial indexing structure (*e.g.*, an R-tree) and the bottom level (*e.g.*, an R-tree leaf node) consists of the text indices (*e.g.*,

Figure 2.7: An example of the spatial-first index

inverted files). Figure 2.7 exemplifies this spatial keyword index for the example dataset in Figure 2.6. The R*-tree is employed as the spatial index and the inverted file is used as the text index. For each leaf node, an inverted file is created to store the geo-textual objects (*e.g.*, the inverted file for node R_1 is shown in the figure). In such type of spatial keyword index, during the query processing, the geo-textual objects are first filtered based on the spatial index, and when the bottom level is reached, the text index is accessed to obtain the objects satisfying the query. The problem of this method is that the spatial filtering stage may still generate too many candidate objects.

In the text-first index, the inverted file is usually used in the top-level and then the postings in each inverted list are arranged in a spatial indexing structure such as an R-tree or a grid index. Figure 2.8 shows an example this spatial keyword index for the example dataset in Figure 2.6. For each keyword, an R*-tree is created to index the geo-textual objects (*e.g.*, the R*-tree for keyword t_1 is shown in the figure).

When a query comes, the spatial indices correspond to the query keywords are accessed to find the geo-textual objects that can answer the query. The problem of this method is that it does not take advantage of the association of keywords in space. An geo-textual object is contained in all the spatial indices correspond to its keywords. Hence, when query keywords are closely correlated in space, this approach suffers from paying extra disk costs accessing different spatial indices for the same geo-textual objects. As shown in the empirical study, the text-first index always outperforms the spatial index *w.r.t.* the query processing time, since the number of query keywords is usually not large.

Figure 2.8: An example of the text-first index

Vaid et al. [99] also present techniques to combine the output of a text and a spatial index to answer a spatial keyword query, which is part of the SPIRIT project [57]. Three types of queries are involved, *i.e.*, Distance, Topological, and Directional, all of which belong to BRQ. The difference from the work [117] is that the grid index is used rather than the R*-tree. Two spatial-textual indexing schemes, namely the spatial primary Index (ST for short, which is spatial-first) and the text primary index (TS for short, which is text-first), are proposed. They are the earliest grid based spatio-textual indices. As shown in the experimental study, the text-first index outperforms the spatial first index, which is consistent with the results of the work [117].

In the proposal [24], each document (web page) consists of a textual part and a page footprint, which is defined as a function that assigns a nonnegative integer to each location in the underlying geographic domain. A query consists of a set of terms and a query footprint. The problem studied is to determine the set of documents that contain all the query terms and that also have a nonempty intersection between document footprint and query footprint, and to compute a relevance score according to a given ranking function on all such documents. As a footprint may spread too large area, it is split into several “toeprints,” which are represented by MBRs of more limited size.

The authors also propose to utilize a text-first and a spatial-first as baseline methods, and the spatial-first baseline performs better. In the proposed improved algorithms, the authors suggest storing toeprints in the disk by following Hilbert curve ordering [53], which maintains the spatial closeness of objects thereby speeding up the disk access operations in retrieving the toeprints. The documents in the inverted index list are assigned IDs according to Hilbert ordering and sorted based on these IDs. A grid-based structure is built in memory to store the IDs of the toeprints in each tile of the grid. When a query is issued by the user, the relevant tiles of the grid that overlap the query

region are retrieved. The toeprints contained in the tiles are sorted and looked up against the inverted indices, and then the obtained documents are ranked according to a certain function.

words	tree nodes
t_1	$R_1, R_2, R_3, R_4, R_5, R_6$
t_2	R_1, R_2, R_5
t_3	R_2, R_4, R_5, R_6
t_4	R_1, R_2, R_4, R_5, R_6
t_5	R_2, R_3, R_5, R_6
t_6	R_4, R_6

Table 2.2: Example of the KR*-tree list

Hariharan et al. [51] present a hybrid indexing structure called the KR*-tree that consists of an R*-tree and an inverted file for the nodes of the R*-tree. This index is designed for the BRQ. All internal and leaf nodes of KR*-tree are augmented with a set of distinct keywords that appear in the space covered by the nodes. Thus, it can support both spatial and textual filtering simultaneously. Since the number of keywords that appear in each node varies, they do not store the keywords in the node. Rather, they construct a special list called KR*-tree List that stores the keywords appearing in the nodes. This list stores for each keyword, the node IDs of KR*-tree in which the keyword appears, along with its parent ID and children IDs. The KR*-tree List is similar to that of an inverted file index, the only difference is that it stores the node IDs instead of object IDs. At query time, the KR*-tree based algorithm finds the nodes that contain the query keywords and then uses these as candidates for subsequent search. This approach suffers from unnecessary overhead when there are many candidates. Table 2.2 shows an example of the KR*-tree list of the example dataset in Figure 2.6.

2. Boolean k NN Query. Felipe et al. [34] propose the IR²-tree, which tightly integrates signature files and the R-tree to answer the Boolean k NN queries. In particular, each node of an IR²-tree contains both spatial and textual information: the former in the form of a minimum bounding rectangle and the latter in the form of a signature file. The drawback of an IR²-tree is that the same signature length is used for all levels which leads to more false positives in higher levels. Hence, they propose the Multi-level IR²-tree (MIR²-tree for short) to handle this problem. The multi-level superimposed coding [20] is used in this index, which reduces the number of false positives, particularly in non-leaf nodes. They use the optimal signature length [100] for each level and superimpose the signatures of all objects in the subtree of each node, instead of the signatures of the children nodes as before. However, signature files can only determine whether a given document contains query keywords but fail to order them based on the textual relevance,

and thus cannot process **HkQ**. In addition, signature files are generally inferior to inverted files for general text retrieval [120], and it usually performs worse than indices adopting the inverted file as the text index.

Cary et al. [18] propose the hybrid Spatial-Keyword Index (SKI) to efficiently process the so-called $k-SB$ queries that finds nearest neighbor objects satisfying Boolean constraints on keywords combined with conjunctive ($\wedge$), disjunctive ($\vee$), and complement ($\neg$) logical operators. SKI uses the modified R-tree and bitmaps to store spatial and text information respectively. The parent nodes of leaf nodes are called super nodes in the modified R-tree. Each non-leaf node is augmented with a range of IDs of the super nodes under the non-leaf node. Each super node is associated with a bitmap version of inverted file. Specifically, if the fanout of the R-tree is m , a super node has at most m leaf nodes each of which contains at most m objects, and thus a super node contains at most m^2 objects. Consequently, the term bitmap of term t at super node s is defined as a fixedlength bit sequence $I(t, s)$ of size m^2 , where the i_{th} bit is set to 1 if that bit corresponds to an object whose description contains t . Then, a “Spatial Inverted File” is created based on these bit sequences of all terms at all super nodes. This inverted file is indexed by B⁺-tree and each bit sequence $I(t, s)$ is compressed using the Word-Aligned Hybrid bitmap compression method [105]. The query processing based on SKI is better than a text-first and a spatial-first approach.

Wu et al. [102] also study the problem of processing **BkQ**. They propose an index called WIR-tree which is a variant of IR-tree [30] (to be detailed later). It aims at partitioning objects into multiple groups such that each group shares as few keywords as possible. To achieve this goal, the objects are first partitioned into two groups using the most frequent word w_1 : one group whose objects contain w_1 and the other group whose objects do not. Then, each group is partitioned by the next frequent word w_2 . The process is repeated iteratively until each partition contains a certain number of objects. After partitioning, each group of objects becomes the leaf node of the WIR-tree. Finally, the tree is constructed following the structure of IR-tree. They further improve the WIR-tree by replacing the inverted files with inverted bitmaps, which reduces the storage and query I/O. As studied in this work, WIR-tree outperforms IR-tree in terms of query processing time.

3. Ranking Range Query. Christoforaki et al. [27] propose several hybrid indices for processing **RRQ**, where the Okapi-BM25 model is used to compute the text relevance. It is required that the returned objects must contain all query keywords and are in the query range, and the objects are ranked according to the similarity computed by the BM25 model.

They first design several baselines following either spatial-first or text-first manner. Next, two improved methods are proposed. The first method utilizes the k -d tree [8] to partition the geographic space into a small number of k regions. Then, either the k

inverted lists for each term are stored sequentially (CSP-SEQ) or each region builds its own inverted file independently (CSP-SUB). The query processing also follows a spatial-first manner: first determine all regions that intersect the query rectangle and then fetching list data from disk if needed in each intersecting region.

The second method uses the Z-order curve [81] to assign to each geo-textual object an ID that corresponds to the position of its location on the Z-curve, and then build a standard block compressed inverted index. The idea is that all objects in the query rectangle are likely close to each other on the Z-curve, and thus close to each other in the inverted list. This allows skipping large parts of the list, and two approaches are explored. The first approach uses an in-memory Quad-tree to enable skipping, denoted by SFC-QUAD. When a query comes, the Quad-tree is first traversed to obtain m ID ranges that contain all objects intersecting the query range, and then the disk-resident inverted lists are swept to fetch the needed parts. In the second approach, each block stores a minimum bounding rectangle encompassing all of the objects contained in the block. During query processing, any block whose MBR does not intersect with the query rectangle can be skipped. This approach is denoted by SFC-SKIP. As shown in the empirical study, SFC-QUAD performs the best under all system settings.

4. Hybrid k NN Query. Martins et al. [79] discuss possible structures for geographic indexing and query processing, including inverted indexes and R-trees, and propose future work in this direction. This work computes text relevancy and location proximity independently and then combine the two ranking scores in four possible different ways: the linear combination, the product, the maximum similarity, and the step-linear function. However, they do not propose any concrete algorithms.

Cong et al. [30] first study the problem of answering the **HkQ** that considers both text similarity and the spatial proximity. Given a query q , it retrieves top- k objects ranked according to the following function:

$$RS(o, q) = \eta \cdot \text{Rel}_S(o.\lambda, q.\lambda) + (1 - \eta) \cdot \text{Rel}_T(o.\psi, q.\psi) \quad (2.4)$$

where $\text{Rel}_S(o.\rho, q.\rho)$ computes the spatial proximity between $o.\lambda$ and $q.\lambda$, $\text{Rel}_T(o.\psi, q.\psi)$ measures the text relevance between $o.\psi$ and $q.\psi$, and $\eta \in (0, 1)$ is a query preference parameter that makes it possible to balance the spatial proximity and text relevance according to user requirements. If users are not satisfied by the results, they can tune this parameter to set their own preference and to obtain better results. The spatial proximity is defined as the normalized Euclidian distance $\text{Rel}_S(o.\rho, q.\rho) = \frac{\text{Dist}(o.\lambda, q.\lambda)}{D_{max}}$ where $\text{Dist}(o.\lambda, q.\lambda)$ is the Euclidian distance between o and q , and D_{max} is the maximum distance between any two objects in the database D . The text relevance $\text{Rel}_T(o.\psi, q.\psi)$ is computed by $\text{Rel}_T(o.\psi, q.\psi) = 1 - \text{Sim}(o.\psi, q.\psi)$, where $\text{Sim}(o.\psi, q.\psi)$ is computed by a language model as defined in Equation 2.3.

Figure 2.9: An example of an IR-Tree

<i>Root</i>	<i>R₅</i>	<i>R₆</i>	<i>R₁</i>	<i>R₂</i>	<i>R₃</i>	<i>R₄</i>
$t_1: R_5, R_6$	$t_1: R_1, R_2$	$t_1: R_3, R_4$	$t_1: o_1$	$t_1: o_8$	$t_1: o_4$	$t_1: o_3$
$t_2: R_5$	$t_2: R_1, R_2$	$t_3: R_4$	$t_2: o_1, o_5$	$t_2: o_2$	$t_5: o_4$	$t_3: o_3$
$t_3: R_5, R_6$	$t_3: R_2$	$t_4: R_4$	$t_4: o_5$	$t_3: o_2$		$t_4: o_6$
$t_4: R_5, R_6$	$t_4: R_1, R_2$	$t_5: R_3$		$t_4: o_7$		$t_6: o_6$
$t_5: R_5, R_6$	$t_5: R_2$	$t_6: R_4$		$t_5: o_7, o_8$		
$t_6: R_6$						

Table 2.3: Content of inverted files of the IR-Tree

The IR-tree index is proposed, which is essentially an R-tree, each node of which is enriched with reference to an inverted file for the objects contained in the sub-tree rooted at the node. Each leaf node in the IR-tree contains entries of the form $(o, o.\lambda, o.di)$, where o refers to an object in dataset D , $o.\lambda$ is the minimum bounding rectangle or the geographical position (when objects are represented by points) of o , and $o.di$ is an identifier of the description of o . Each leaf node also contains a pointer to an inverted file with the keywords of the objects stored in the node, which is stored separately. An inverted file consists of the following two main components.

- A vocabulary of all distinct words appearing in the description of an object.
- A set of posting lists, each of which relates to a term t . Each posting list is a sequence of pairs $\langle d, w_{d,t} \rangle$, where d refers to the description of an object which contains t , and $w_{d,t}$ is the weight of term t in d .

Each non-leaf node in the IR-tree contains a number of entries of the form $(cp, cp.\lambda, cp.di)$, where cp is the address of a child node of the non-leaf node, $cp.\lambda$ is the minimum bounding rectangle of all rectangles in entries of the child node, and $cp.di$ is an identifier of a pseudo text description that is the union of all text descriptions in the entries of the

child node. Each non-leaf nodes also has an inverted file, which stores the identifiers of child nodes containing each term t . The weight of each term t in the pseudo text description referenced by $cp.di$ is the maximum weight of the term in the documents contained in the subtree rooted at node cp . As an example, Figure 2.9 illustrates the corresponding IR-tree, and Table 2.3 shows the content of the inverted files associated with the nodes (term weights are not shown for simplicity). During the query processing both the spatial and text pruning can be applied simultaneously.

The authors also propose several methods to optimize the IR-tree, including the DIR-tree, the CIR-tree and the CDIR-tree. The DIR-tree takes both spatial and textual information into account during the tree construction by optimizing for a combination of minimizing the areas of MBRs and maximizing the text similarities between the objects of the enclosing rectangles. A parameter β is introduced to balance the weights on the two parts. The CIR-tree optimizes the IR-tree by grouping objects into a number of clusters based on their text descriptions. Compared with the IR-tree, the CIR-tree includes a summary of text content for each cluster in its posting lists. The CDIR-tree is a combination of the DIR-tree and the CIR-tree, which shows the best performance as shown in the experimental study.

In their subsequent work [101], Wu et al. also consider processing the hybrid k NN query where the location is represented by a rectangle instead of a point, which can also be answered using the proposed indices in the work [30]. In addition, two improved indices are devised—the ClusterMBR and TermMBR enhancements are applied to the CDIR-tree, denoted as the CM-CDIR-tree and TM-CDIR-tree, respectively. The ClusterMBR enhancement refines the MBR of each cluster by constructing an MBR for each cluster at a node. This enables to estimate a tighter bound on the distance from the query to the node. The TermMBR enhancement refines the MBR of each node by constructing an MBR for each term in this node. As reported, both enhancements outperform the CDIR-tree, which is the best in the earlier work [30], and the CM-CDIR-tree performs even better than the TM-CDIR-tree.

Li et al. [72] presents a hybrid index structure which is also called IR-tree (to distinguish with the index proposed in the work [30], it is denoted by LiIR-tree) for processing **HkQ**. The first difference between it and the work [30] is that the vector space model using TF-IDF term weighting is employed to compute the text relevance between the query and objects, rather than the language model. Second, the IR-tree [30] stores the inverted files for each node separately. However, only the leaf nodes in the LiIR-tree [72] has their own inverted files. For all the non-leaf nodes, LiIR-tree stores DF/TF pairs related to the same word but for different nodes together in the same memory block, which refers to a linked list of pages. The DF/TF pair of a node i *w.r.t.* a word w is in format of $\{df_{w,D_i}, TF_{w,D_i}\}$, where D_i is the set of text descriptions beneath node i , df_{w,D_i} represents the number of documents in D_i that contain w , and TF_{w,D_i} is the aggregated

information about the term frequency of w in D_i . Specifically, the authors investigate two different representations of TF_{w,D_i} . The fashion that the LiIR-tree organizes the inverted file is similar to that of the KR*-tree [51], and the difference lies in that the KR*-tree does not store the weight of terms. As compared in the works [23, 101], the LiIR-tree has similar performance with the IR-tree, and thus performs worse than the indices improved on IR-tree, *i.e.*, the CIR-tree, the DIR-tree, the CDIR-tree [30], the CM-CDIR-tree, and the TM-CDIR-tree [101].

Khodaei et al. [63] propose a new index structure called Spatial-Keyword Inverted File (SKIF) to handle location-based web searches in an integrated manner, where the location of each object is represented by a region. A new distance measure called spatial TF-IDF is defined, and based on this four variants of HkQ are proposed. The whole space is partitioned into several grid cells, and each geo-textual object can be associated with a set of cell identifiers overlapping with it. With spatial TF-IDF, the overlap of a cell with the object is analogous to the existence of a keyword in document with TF-IDF. The overlap area between each cell and the object provides a measure of how well that cell describes the object. Specifically, the frequency of cell c in an object o is defined as the area of overlap between o and c divided by the area of cell c . Similarly, the spatial inverse document frequency is also defined to assign less weight to cells that appear in many documents. SKIF employs the inverted file structure to store both spatial and textual information for objects. The vocabulary of the inverted file includes both the keywords and the cells, and the posting lists store the object IDs that contain the keyword or overlap with the cell with the corresponding term or cell frequency. In their later work [64], the geo-textual objects are represented by points, and the spatial term frequency is measured by some spatial decay functions. As reported by the authors, their index performs better than the CDIR-tree [30] when the text relevance plays an less important role during query processing.

Rocha-Junior et al. [87] proposed an index called Spatial Inverted Index (S2I) to process the same type of query in the work [30], which ranks objects by linearly combining text relevance and spatial proximity. S2I employs two different strategies for indexing frequent keywords and infrequent keywords. The threshold to distinguish frequent terms from infrequent terms needs to be set empirically. Specifically, the S2I maps objects containing each frequent term to an aggregate R-tree [83] (aR-tree), in which each node stores an aggregated value that captures the maximum impact (in terms of the text relevance score) of the term on the objects in the subtree rooted at the node. The S2I organizes the objects containing each infrequent term by an inverted file. For each object, the object ID, the object location, and the impact of term t in the object are stored. Then, S2I organize the blocks and trees by the vocabulary, which stores, for each distinct term, the number of objects in which the term appears, a flag indicating the type of storage used by the term (block or tree), and a pointer to a block or the aR-tree

that stores the objects containing the given term. As shown in the experimental study, it outperforms DIR-tree [30] in terms of query time. However, the work [23] states that it costs too much disk space, since an object is repeatedly stored in many trees and objects.

Recently, both works [111, 115] adopt the Quad-tree and the inverted file to build a spatial keyword index, where the Quad-tree is created for each keyword. Such index is more suitable when geo-textual objects are inserted and deleted frequently, because the Quad-tree is recommended for update intensive applications [59] and it provides a uniform space decomposition mechanism for all the keywords. The index in the proposal [115] is called I^3 . Each object is presented by a tuple that stores its ID and geographic information. The maximum capacity of the Quad-tree cell is computed based on the disk page size. When a page cannot store any more tuples, the cell is split. The index is composed by the lookup table that stores the page information for each keyword, the head file that contains the information for pruning, and the data file that contains a sequence of fixed-size pages for storing the tuples. The index in the proposal [111] is called the inverted linear Quad-tree (IL-Quadtree). The difference is that the authors utilize the linear Quad-tree [47], in which only the non-empty leaf node of the Quad-tree is stored in an auxiliary disk-based one dimensional structure (*e.g.*, B^+ -tree), and each node is encoded by the Z-order curve. The Z-order code of a node is encoded based on the path of the node in the Quad-tree, and the code of a particular node (region) in the space is unique. Both works outperform the previous methods on HkQ , such as the IR-tree [30] and S2I [87].

5. Summary and Comparison.

Table 2.4 summarizes the above proposals on single granularity-based spatial keyword querying, including the query supported, the spatial and text index utilized, and the combination manner of the two types of indices.

The indexing structures for the queries with Boolean keyword predicate cannot be applied to answer queries with ranking-based keyword predicate, but the indices for RRQ and HkQ can be easily modified to process BRQ and BkQ , respectively. As studied in the work [23], the index SFC-Quad proposed by Christoforaki et al. [27] always performs the best for answering BRQ and RRQ . For answering BkQ , if both space cost and query processing time are important, the WIR-tree proposed by Wu et al. [102] is the best choice. If only focusing on the query processing time, S2I proposed in the work [87] has the best performance. S2I also achieves the best efficiency for answering HkQ , if the space cost is not a big issue. Otherwise, the CIDR-tree, a variant of the IR-tree, proposed by Cong et al. [30] is suggested.

Proposal	Query	Spatial part	Textual part	Combination Scheme
Zhou et al. [117]	BRQ	R*-tree	Inverted file	Spatial-first and text-first
Vaid et al. [99]	BRQ	Grid	Inverted File	Spatial-first and text-first
Chen et al. [24]	BRQ	Hilbert curve	Inverted File	Spatial-first
Hariharan et al. [51]	BRQ	R*-tree	Inverted File	Tightly combined
Felipe et al. [34]	BkQ	R-tree	Signature File	Tightly combined
Cary et al. [18]	BkQ	R-tree	Bitmap	Tightly combined
Wu et al. [102]	BkQ	R-tree	Inverted File	Tightly combined
Christoforaki et al. [27]	RRQ	Z-order curve	Inverted File	Tightly combined
Cong et al. [30]	HkQ	R-tree	Inverted File	Tightly combined
Li et al. [72]	HkQ	R-tree	Inverted File	Tightly combined
Khodaei et al. [63]	HkQ	Grid	Inverted File	Tightly combined
Rocha-Junior et al. [87]	HkQ	aR-tree	Inverted File	Text-first
Zhang et al. (I^3) [115]	HkQ	Quad-tree	Inverted File	Tightly combined
Zhang et al. [111]	HkQ	Quad-tree	Inverted File	Tightly combined

Table 2.4: Comparison of existing geo-textual indices

2.4.2.2 Querying beyond the Single Object Granularity

Zhang et al. [113] propose a type of query called the m -closest keyword query (mCK). It searches for the closest objects containing at least m specified query keywords. A hybrid index structure that combines the R*-tree and bitmap indexing named bR*-tree is developed to process such queries. Each R*-tree node is augmented with a bitmap indicating the keywords contained in the objects rooted at this node. Besides the keyword bitmap, they also store the keyword MBR in each node which is the minimum bounding rectangle of the objects containing a keyword in the node.

Figure 2.10: An example of the virtual bR*-tree

In their subsequent work [114] an improved version of bR*-tree called virtual bR*-tree is proposed. The objects are indexed by R*-tree, where each node is assigned a label

indicating the path to the root node. Thus, given a node label, it can be judged where the node is located in the R^* -tree without accessing the tree. Then, an inverted index is built along with the R^* -tree. It maintains inverted lists for all the tags in the database. Each element in the list consists of the node label derived from the construction of the R^* -tree and the actual location. Figure 2.10 shows an example of the R^* -tree with labels and the steps of building a virtual bR^* -tree given a query $\{t_3, t_6\}$, using the example dataset in Figure 2.6.

Long et al. [74] also study the processing of **MAX+MAX SGK** queries, where the cost is computed as the sum of the maximum distance between the query and the group and the maximum distance between any pair in the group. They propose an approximation algorithm as well as an exact algorithm. As shown in Section 4.5.2.2, our approximate algorithms run much faster with only slightly worse accuracy, and our exact algorithm performs much better, comparing with the algorithms proposed in the work [74]. They also study a variant of **SGK**, which includes the query point when computing the cost for a group and is processed similarly by the algorithms for **SGK**.

There are also some proposals on the keyword-aware trajectory search such as the works [93, 116]. Shang et al. [93] propose and investigate a novel problem called User Oriented Trajectory Search (UOTS). The UOTS query contains a set of intended places given by the traveler and a set of textual attributes describing the traveler's preference. If a trajectory in the given trajectory dataset is connecting/close to the specified query locations, and the textual attributes of the trajectory are similar to the traveler's preference, it will be recommended to the traveler for reference. Zheng et al. [116] study the problem of efficient similarity search on activity (represented by keywords) trajectory database. They propose the Activity Trajectory Similarity Query (ATSQ). Given a sequence of query locations, each associated with a set of desired activities (keywords), an ATSQ returns k trajectories that cover the query activities and yield the shortest minimum match distance.

2.4.2.3 Other Types of Queries

Fan et al. [42] study the spatial keyword query on geo-textual objects represented by rectangles. The aim is to find the similar objects by considering spatial overlap and textual similarity. They define the spatial Jaccard similarity using the regions of two objects, and the text similarity is computed by the weighted Jaccard coefficient. Given a query consists of a rectangle region, a set of keyword, a spatial similarity threshold τ_R , and a text similarity threshold τ_T , the objects whose spatial similarity with the query is larger than τ_R and text similarity with the query is larger than τ_T are returned. A filter-and-verification framework is introduced to compute the answers. In the filter step, signatures are generated for the objects and the query, and the signatures are utilized to

generate candidates whose signatures are similar to that of the query. In the verification step, the candidates are verified to find the final answers.

Wu et al. [104] study the efficient processing of continuously moving top- k spatial keyword (MkSK) queries. This query enables a mobile user (or driver) to be continuously aware of the k geo-textual objects that best match a query with respect to location and text relevancy. The ranking score of an object *w.r.t.* a query is defined by the Euclidean distance between the object and the query divided by the text relevance between them. The aim includes: (i) the client can receive correct results at any point in time, (ii) the computational server-side cost is optimized, and (iii) the client/server communication cost is optimized. They utilize the IR-tree [30] for query processing. In their subsequent work [103], more analysis on processing the MkSK query is provided.

Lu et al. [76] study the problem of Reverse Spatial and Textual k Nearest Neighbor (RST k NN) query, which finds geo-textual objects that have the query object q_o in their lists of top- k objects ranked by a linear ranking function that takes both spatial proximity and text relevance into consideration. They propose the Intersection-Union R-tree (IUR-tree) index structure, which is a combination of textual vectors and an R-tree. Each node of an IUR-tree contains a minimum bounding rectangle(MBR) and two textual vectors: an intersection vector and a union vector. The weight of each item in the intersection (resp. union) textual vector is the minimum (resp. maximum) weight of the items in the documents contained in the subtree rooted at this node. They then estimate the lower and upper bounds between IUR-tree nodes which are used in the branch-and-bound search algorithm.

The work [88] studies the problem of answering spatial keyword queries on road networks. The ranking score of an object *w.r.t.* the query is defined as:

$$RS(o, q) = \frac{\theta(o.\psi, q.\psi)}{1 + \alpha \cdot \delta(o.\lambda, q.\lambda)},$$

where $\delta(o.\lambda, q.\lambda)$ represents the network proximity between the query q and the object o , and $\theta(o.\psi, q.\psi)$ represents the textual relevance of o to q . The basic indexing architecture combines the IR-tree [30] and the road network framework proposed by Papadias et al. [84]. They also design an enhanced index consists of two components: i) the mapping component that maps a key composed by the pair of edge ID and term ID to the inverted list; ii) the inverted file component that contains inverted lists, each of which stores the objects lying on the edge (v, v') that have a term t in their description. For each object, the inverted list stores the network distance between the object and the reference vertex of the edge (v, v') and the impact of the term t in the description of the object. The algorithm based on the enhanced index shows better performance than that using the basic index.

Bouros et al. [13] propose to answer the spatio-textual similarity joins query (ST-SJOIN). Given a dataset of geo-textual objects R , a distance threshold ϵ , and a text similarity threshold θ , the ST-SJOIN query aims to find all pairs (x, y) with $x, y \in R$, such that the distance between x and y is smaller than ϵ and the text similarity between x and y is larger than θ . The work combines ideas from state-of-the-art spatial distance join and set similarity join methods and propose efficient algorithms that take into account both spatial and textual constraints. In addition, a batch processing technique is designed to boost the performance of the proposed approaches.

Liu et al. [73] study a similar problem: given two sets of geo-textual objects R and S , find all similar pairs (x, y) with $x \in R$ and $y \in S$, such that the spatial proximity between x and y is larger than a given spatial similarity threshold and the text similarity between x and y is larger than a given textual similarity threshold. They develop a filter-and-refine framework, which is similar to that in the work [42]. First, they generate spatial and textual signatures for the objects in R and S and build inverted index on top of these signatures. Then, they generate candidate pairs using the inverted lists of signatures. Finally, the candidates are verified to generate the final result. The geo-textual objects are represented by rectangles in this proposal, while Bouros et al. [13] consider the spatio-textual similarity join problem over objects represented by points.

Chen et al. [22] study the problem of matching a stream of incoming Boolean Range Continuous (BRC) queries over a stream of incoming geo-textual objects in real time. The BRC query consists of three components: a boolean keyword expression, a spatial region, and a time interval. The answer to a BRC query comprises such geo-textual objects that satisfy the boolean keyword expression, fall in the specified spatial regions, and are generated during the specified time interval. They propose a hybrid index, called Inverted File Quad-tree (IQ-tree), and novel cost models for managing a stream of incoming BRC queries. The IQ-tree is essentially a Quad-tree extended with inverted files. It integrates the Quad-tree for organizing the spatial region information of BRC queries and the inverted file for organizing the keyword expression of BRC queries. Each node of the IQ-tree is associated with an inverted file that organizes the keyword expression of the BRC queries associated with the node. When a query comes, it is inserted into the IQ-tree according to some cost models. When a geo-textual object comes, it is sent to the queries that it can satisfy.

The proposal [70] studies the problem of direction-aware spatial keyword query. Besides a location point and a set of keywords, the query also contains a direction constraint $[\alpha, \beta]$, which denotes that the user is only interested in the objects with directions to q in $[\alpha, \beta]$. An object is an answer of a query only if it covers all the query keywords and satisfies the direction constraint.

These proposals consider both the spatial and textual factors in their problems, but are different from traditional spatial keyword queries because the objectives are totally different.

2.5 Summary

This section reviews the existing studies on or related to spatial keyword querying. Comparing with the proposed three different approaches in this thesis, which do the spatial keyword querying beyond the single object granularity, it can be concluded that: 1) None of the existing work on spatial keyword querying has taken into account the inter-object relationship as we do in the *kPSK* query; 2) The work [74] studies one instance of our *SGK* query. However, our exact algorithm outperforms their exact algorithm; and 3) Our *KOR* query cannot be answered by using the existing approaches on spatial keyword search over road networks.

Chapter 3

Top-k Prestige-based Spatial Keyword Query

This chapter is organized as follows: Section 3.1 defines the *kPSK* query. Section 3.2 details the proposed algorithms. In Section 3.3, we report on the empirical studies. Section 3.4 reviews more related works, and Section 3.5 concludes this chapter.

3.1 Problem definition

Let $\mathcal{D}$ be a set of geo-textual objects. Each object o in $\mathcal{D}$ has a text description $o.\psi$ and a location $o.\lambda$. Similarly, a *top-k Prestige-based Spatial Keyword (kPSK)* query $Q = \langle \psi, \lambda \rangle$ has a location $Q.\lambda$ and a set of keywords $Q.\psi$.

Let $\text{Dist}(Q, o)$ denote the Euclidean distance between the locations of query Q and object o , and let $\text{Sim}(Q, o)$ denote the text relevance between the keyword component of query Q and the text description of object o . We use the vector space model (VSM) [119], one of the most popular ranking functions, for computing $\text{Sim}(Q, o)$, while using the TF-IDF weighting scheme to represent the text descriptions of objects. The vector space model is defined as Equation 2.1.

To compute the PR scores of objects, we define a weighted, undirected graph $\mathcal{G} = (V, E)$ over $\mathcal{D}$, where each node in V corresponds to a geo-textual object and edge set E includes an edge $\langle o_i, o_j \rangle$ iff the following two conditions are satisfied: 1) $\text{Dist}(o_i, o_j) \leq \gamma$ and 2) $\text{Sim}(o_i, o_j) \geq \xi$, where γ and ξ are threshold parameters. The weight of edge $\langle o_i, o_j \rangle$ in E is $\text{Dist}(o_i, o_j)$.

Prestige-Based Relevance (PR). The final PR vector $\vec{p}$ fulfills the following equation:

$$\begin{aligned} \vec{p} &= (1 - \alpha)\mathbf{C}^T \vec{p} + \alpha \vec{u}_Q, \\ \vec{u}_Q &= [v_1, \dots, v_{|\mathcal{D}|}]^T, v_i = \text{Sim}(Q, o_i), 1 \leq i \leq |\mathcal{D}|, \end{aligned} \quad (3.1)$$

where C is the normalized adjacency matrix of graph $\mathcal{G}$ such that $\sum_{j \in V} C(b, j) = 1$, where $C(b, j)$ represents the normalized weight from node b to node j ; and column vector u_Q is the initial PR vector in which each element is the relevance of an object.

Parameter α represents the probability of a random surfer jumping to the set of initially relevant spatial objects ($v_i > 0$) instead of following the edges in the graph. Interestingly, parameter α can be used to balance the relevance of an object and the effect of its relevant neighbors, *i.e.*, the parameter allows for tuning according to user-specific requirements. In particular, smaller values of α favor objects with nearby relevant objects, while larger values of α favor objects with high initial PR scores. Users can set the value of α according to their preference. If they would like to obtain results more relevant to the query and not affected by neighbors too much, they can set α at a larger value, and vice versa.

To understand the random walk process for each object b , its PR score $\vec{p}(b)$ can be rewritten as follows:

$$\vec{p}(b) = \alpha \vec{u}_Q(b) + (1 - \alpha) \sum_{j \in V} C(j, b) \vec{p}(j), \quad (3.2)$$

where $\vec{u}_Q(b)$ is the initial PR score of b .

The iterative computation diffuses the PR score of each object across the graph. In the beginning, each object gets its initial PR score according to its text relevance to the query. At each step, an α fraction of the PR score is held by each node, while the remaining $(1 - \alpha)$ flows by following the links of the graph. This propagation continues until all the prestige is distributed cross the graph. The final PR scores take into account both the original relevance scores and the effect of neighbor nodes. The PR vector is inspired by the personalized PageRank vector (PPV) [56] and the bookmark coloring vector (BCA) [10].

k PSK Query Definition. Intuitively, an k PSK query retrieves k objects from database $\mathcal{D}$ ranked according to a combination of their distances to the query location and their PR scores for the query. Formally, given a query $Q = (\psi, \lambda)$, where $Q.\psi$ is a location descriptor and $Q.\lambda$ is a set of keywords, the objects returned are ranked according to a ranking function $f(\text{Dist}(Q, o), \text{Pr}(Q, o))$, where $\text{Dist}(Q, o)$ is the Euclidian distance between Q and o and $\text{Pr}(Q, o)$ is the PR score of o with respect to Q . A k PSK query becomes an LkT query [30] in the extreme case of $\alpha = 1$ (Equation 3.2), *i.e.*, we disregard the effects of nearby relevant objects.

Problem Statement. We address the problem of efficiently answering k PSK queries.

The k PSK query is applicable to a wide range of ranking functions that are monotone with respect to the distance proximity $\text{Dist}(Q, o)$ and the PR score $\text{Pr}(Q, o)$. We follow

existing work [30, 79] and use a linear combination of the normalized factors for ranking an object o with respect to a query Q :

$$\text{RS}(Q, o) = (1 - \beta)(1 - \text{Pr}(Q, o)) + \beta \frac{\text{Dist}(Q, o)}{\text{max}D} \quad (3.3)$$

where $\beta \in (0, 1)$ is used to balance the PR score and the location proximity (if users would like to obtain results not too far away from their current location, they can set β at a smaller value, and vice versa); Euclidian distance $\text{Dist}(Q, o)$ between query Q and object o is normalized to a value between 0 and 1 by a constant $\text{max}D$, which can be the maximum distance between two objects in $\mathcal{D}$ or the maximum distance that can be accepted by the users; and $\text{Pr}(Q, o)$ is the PR score of object o *w.r.t.* query Q and usually takes a value between 0 and 1. This function computes the ranking score of each object given a $k\text{PSK}$ query.

Note that parameter β allows to set the preference between the PR score and the location proximity at query time.

3.2 Proposed Solutions

3.2.1 Baseline Algorithm

As a baseline, we present an improvement of the straightforward solution mentioned in Section 1.3.1.1.

In the straightforward solution, we compute PR scores for all objects and the distances between all objects and the query, upon which we rank the objects based on the combined scores. We focus on computing the costly PR scores.

We choose to adapt the bookmark-coloring algorithm (BCA) [10], an elegant algorithm for computing BCVs, to computing the PR scores. We first extend the in-memory BCA algorithm to work in secondary memory. We read the graph in large blocks that each exploit the memory available, and do the iterative propagation in a per-block manner. The computation stops when the termination condition for the graph is met. A block is likely to be read and written multiple times since it may receive PR scores from other blocks that need to be distributed. Second, we extend BCA, which works on unweighed graph for a single preferred object, to support PR score computation for a general preference vector $\vec{u}_Q$ on a weighted graph.

Algorithm 1 details the resulting Extended BCA (EBC) algorithm. Let $\vec{p}$ denote the PR score that each object already has, let $\vec{q}$ denote the PR score that each object needs to distribute, and let outPR be the vector of the sum of the PR scores that need to be distributed in each graph block.

Algorithm 1: EBC(Q)

```

1  compute the text relevance of each object to  $Q$  and compute  $u_{\vec{Q}}$ ;
2   $\vec{q} \leftarrow u_{\vec{Q}}$ ,  $\vec{p} \leftarrow \vec{0}$ ,  $outPR \leftarrow \vec{0}$ ;
3   $blockQueue \leftarrow \text{NewPriorityQueue}()$ ;
4  foreach object  $b$  do  $outPR(b.block) \leftarrow outPR(b.block) + \vec{q}(b)$ ;
5  foreach block  $bg$  do  $blockQueue.Enqueue(bg)$ ;
6  while  $\|\vec{q}\|_1 \geq \varepsilon$  do
7       $bg_i = blockQueue.Dequeue()$ ;
8      read the graph block  $bg_i$ ;
9       $outPR(bg_i) \leftarrow 0$ ;
10      $Queue \leftarrow \text{NewQueue}()$ ;
11     foreach object  $n$  s.t.  $n \in bg_i$  and  $\vec{q}(n) > 0$  do
12          $Queue.Enqueue(n)$ ;
13     while not  $Queue.Empty()$  do
14          $b \leftarrow Queue.Dequeue()$ ;
15         if  $\vec{q}(b) > \epsilon$  then
16              $\vec{p}(b) \leftarrow \vec{p}(b) + \alpha \vec{q}(b)$ ;
17             foreach out-neighbor  $j$  of  $b$  do
18                 if  $\vec{q}(j) = 0$  and  $j \in bg_i$  then
19                      $Queue.Enqueue(j)$ ;
20                  $outV \leftarrow (1 - \alpha)C(b, j)\vec{q}(b)$ ;
21                  $\vec{q}(j) \leftarrow \vec{q}(j) + outV$ ;
22                 if  $j \notin bg_i$  then
23                      $outPR(j.block) \leftarrow outPR(j.block) + outV$ ;
24              $\vec{q}(b) \leftarrow 0$ ;
25          $blockQueue.Update()$ ;
26 return  $\vec{p}$ 

```

We compute the text relevance of each object to the query Q using an inverted list index and then construct the preference vector $u_{\vec{Q}}$ according to Equation 3.1 (line 1). We use a priority queue $blockQueue$ whose key is the accumulated outgoing PR score that needs to be propagated in each block (lines 3–5). In each graph block, we modify the propagation mechanism of BCA to accommodate edge weights and multiple objects in the preference vector.

We use a queue $Queue$ to store the objects that have PR scores that need to be distributed (lines 10–12). Specifically, for the PR $\vec{q}(b)$ that needs to be distributed at an object b , we assign $\alpha \vec{q}(b)$ to b (line 16) and $(1 - \alpha)\vec{q}(b)$ to its neighbors according to the edge weights (lines 17–20). We update the PR score that needs to be distributed for each block (lines 21–22).

When the PR score of each object that needs to be distributed is smaller than the propagation threshold ϵ , we stop the propagation within a block (line 15). When the

PR score to be distributed is smaller than the tolerance threshold ε (line 6), we stop the propagation over the graph and return PR vector $\vec{p}$, each element of which represents the PR score $\text{Pr}(Q, o)$ of object o . We thus use ε and ϵ as the termination conditions, as in BCA [10] and its variant [49].

This method is inefficient because it computes PR scores and distances for all objects. The PR scores of objects are inter-dependent and need to be computed together, while the distance scores can be computed individually. Thus, inspired by the Threshold algorithm [39], we develop two improved baseline algorithms that avoid unnecessary distance score computations.

Baseline 1: This algorithm computes PR scores of all objects and uses an R*-tree to incrementally compute the nearest neighbor in a second stage. When computing the PR scores, the algorithm obtains a list L_{PR} that ranks the objects involved in ascending order of their scores. The algorithm then incrementally finds nearest neighbors [54] using the R*-tree and checks the PR scores of the objects in L_{PR} , until further objects will not become top- k results.

The tricky part is when to stop finding nearest neighbors. The algorithm maintains the minimum PR score in L_{PR} , denoted by $\min_{PR}$, that has not been “seen” so far, and it maintains the combined ranking score (defined in Equation 3.3) of the current k -th object, denoted by ϕ .

For a newly “seen” object with spatial distance d , if the combined score (the lower the score, the better) computed from d and the current $\min_{PR}$ exceeds ϕ , the algorithm stops since it is guaranteed that all “unseen” objects will not have lower scores than the current k -th object (and thus cannot be in the result).

Baseline 2. This algorithm computes the PR scores of all objects, thus obtaining a list L_{PR} that ranks them in ascending order of their scores. The list is then scanned to compute the spatial proximity to the query until further scanning will not generate top- k results.

During the scan, the algorithm keeps track of the combined ranking score of the current k -th object, denoted by ϕ . For a new object o , if its PR score exceeds ϕ , the algorithm stops since all objects after o in L_{PR} will have a score that exceeds ϕ ; otherwise, we retrieve its location, compute its combined ranking score (Equation 3.3), and compare with ϕ to update ϕ if needed.

3.2.2 Early Stop EBC Algorithm (ES-EBC)

PR scores are much more costly to compute than distances because they require iterations over possibly large graphs. We thus proceed to propose two new techniques for speeding up the computation of PR scores. First, we show how to stop the iterative PR score

computation early, using a new stopping condition (Theorem 3.1). Second, we show how to disregard objects further away from the query than a certain distance (Theorem 3.2).

As before, let $\vec{p}$ denote the vector of the PR score that each object already has, and let $\vec{q}$ denote the vector of the outgoing PR score that each object needs to distribute.

Lemma 3.1 *Let $\vec{p}_i(b)$ denote the PR score of object b in the i -th iteration during PR scoring. Then $\vec{p}_i(b) \leq \vec{p}_{i+1}(b)$.*

Lemma 3.2 *Given an object b , the final PR value $\vec{p}(b)$ of b and the value $\vec{p}_i(b)$ of b in the i -th iteration fulfill the following:*

$$\vec{p}_i(b) \leq \vec{p}(b) \leq \vec{p}_i(b) + \alpha^2 \max(\vec{q}_i) + (1 - \alpha) \|\vec{q}_i\|_1,$$

where $\max(\vec{q}_i)$ is the maximum element in $\vec{q}_i$ in the i -th iteration, and $\|\cdot\|_1$ is the 1-norm.

Lemma 3.1 holds because in each PR scoring iteration, a node will increase its current PR score. The proof of Lemma 3.2 follows previous work [49]. The current ranking score $\text{CRS}(b)$ of an object b is computed according to Equation 3.3 at the current (i -th) iteration of PR scoring. We estimate lower and upper bounds on the ranking score for each object in the i -th iteration as follows:

$$\begin{aligned} \text{Upper}(b) &= \text{CRS}(b) \\ \text{Lower}(b) &= \text{CRS}(b) - (1 - \beta)(\alpha^2 \max(\vec{q}_i) + (1 - \alpha) \|\vec{q}_i\|_1) \end{aligned} \tag{3.4}$$

The upper bound holds because the distance between an object and the query is constant, while its PR score increases in each iteration (Lemma 3.1). We obtain the lower bound based on Lemma 3.2 and Equation 3.3.

Theorem 3.1 *Let priority queue L_r record the current top- $(k + 1)$ objects seen, the key being the objects' PR scores. It is guaranteed that the top- k objects have been found if the k -th and the $(k + 1)$ -st objects, represented by $L_r(k)$ and $L_r(k + 1)$, respectively, satisfy the following condition:*

$$\text{Upper}(L_r(k)) < \text{Lower}(L_r(k + 1))$$

Proof: In each PR scoring iteration, a node increases its current PR score. Given a query Q , we have $\text{RS}(Q, L_r(k + 1)) \geq \text{Lower}(L_r(k + 1))$ and $\text{RS}(Q, L_r(k)) \leq \text{Upper}(L_r(k))$.

Together with the condition in the theorem, we also have that $\text{RS}(Q, L_r(k)) < \text{RS}(Q, L_r(k + 1))$. Consider an object m in $L_r[1, k - 1]$ and an object n not in L_r . Because $\text{CRS}(m) \leq \text{CRS}(L_r(k)) \leq \text{CRS}(L_r(k + 1)) \leq \text{CRS}(n)$ and because of Equation 3.4, we have $\text{Upper}(m) \leq \text{Upper}(L_r(k)) < \text{Lower}(L_r(k + 1)) \leq \text{Lower}(n)$. Thus, we have $\text{RS}(Q, m) < \text{RS}(Q, n)$.

The PR score computation stops iterating when the condition in Theorem 3.1 is satisfied, *i.e.*, the current top- k objects all have smaller final ranking scores than those of all other nodes.

Theorem 3.2 *Given a query Q , a set of candidate objects C , and a spatial cell Ω_i , objects contained in Ω_i can be disregarded during propagation if the following is satisfied:*

$$\text{minDist}(Q, \Omega_i) > \gamma \log_{1-\alpha} \frac{\epsilon}{\text{outC}(\Omega_i)} + \text{Dist}(Q, o_s),$$

where $\text{minDist}(Q, \Omega_i)$ is the minimum distance between Q and Ω_i , γ is the distance threshold used when building the object graph; α and ϵ are as explained in Algorithm 1; o_s is the object in C furthest away from query Q ; and $\text{outC}(\Omega_i)$ stores the aggregated PR score that needs to be distributed in Ω_i .

Proof: According to the triangle inequality, we have $\text{Dist}(o_s, \Omega_i) \geq |\text{Dist}(Q, \Omega_i) - \text{Dist}(Q, o_s)|$. Hence, the minimum number of edges in the path from O_s to gb can be computed by $\frac{|\text{Dist}(Q, \Omega_i) - \text{Dist}(Q, o_s)|}{\gamma}$.

In each propagation, an object will distribute the fraction $(1 - \alpha)$ of its PR score to other objects by following its out-edges. Therefore, it follows that an upper bound on the effect of Ω_i on o_s is $\text{outPR}(\Omega_i)(1 - \alpha)^{\frac{|\text{Dist}(Q, gb) - \text{Dist}(Q, o_s)|}{\gamma}}$. If the upper bound is smaller than ϵ , the effect of graph block Ω_i on o_s can be ignored (the effect on other objects in C is even smaller). From the inequality relationship between the upper bound and ϵ , when a block graph is far away, we can get:

$$\text{Dist}(Q, \Omega_i) > \gamma \log_{1-\alpha} \frac{\epsilon}{\text{outPR}(\Omega_i)} + \text{Dist}(Q, o_s),$$

which completes the proof.

The condition stated in Theorem 3.2 guarantees that the objects in cell Ω_i will neither become top- k results nor affect the PR scores of the top- k results. Thus, the objects in the cell can be disregarded during the propagation of scores.

The algorithm that exploits the early stopping conditions first divides the graph into blocks according to the locations of the spatial objects such that each block fits into memory. Each block is further divided into a grid of spatial cells. Then nearest neighbors are retrieved incrementally [54] using the R*-tree [7]. For each nearest neighbor object o , the block graph containing o is read cell by cell: for each cell the algorithm checks whether it can be pruned according to Theorem 3.2; if it cannot, it reads the part of the graph corresponding to the cell. Then it iterates in the block to get the local PR scores for the objects in the block.

The algorithm keeps track of the current top- $(k + 1)$ objects. When the ranking score of the k -th object is smaller than the lower bound (the minimum possible) ranking score

of the current nearest-neighbor object o , i.e., $\Delta = \beta \frac{\text{Dist}(Q, L_r(k))}{\max D}$, the nearest-neighbor retrieval stops because no unseen object has a lower ranking score than has object o (since unseen objects no closer to the query than o) and thus cannot be a top- k object.

This way, we obtain a set of candidate top- k objects. However, these do not necessary constitute the final result since PR scores were only propagated inside a block. We need to propagate the PR scores across blocks while still using Theorems 3.1 and 3.2.

Algorithm 2: ES-EBC ($rtreeIndex, k, Q$)

```

1  compute the text relevance of each object to  $Q$  and  $u_{\vec{Q}}$ ;
2   $\vec{q} \leftarrow u_{\vec{Q}}$ ,  $\vec{p} \leftarrow \vec{0}$ ,  $L_d \leftarrow \vec{0}$ ,  $outPR \leftarrow \vec{0}$ ;
3  foreach object  $b$  do  $outPR(b.block) \leftarrow outPR(b.block) + \vec{q}(b)$ ;
4   $L_r \leftarrow \text{NewPriorityQueue}()$ ;
5  Initialize  $L_r$  with  $k + 1$  objects whose key values are  $\infty$ ;
6  while true do
7       $b \leftarrow rtreeIndex.\text{NextNearestNeighbor}(Q)$ ;
8       $L_d(b) \leftarrow \text{Dist}(Q, b)$ ;
9      if  $\text{CRS}(L_r(k)) \leq \beta * \frac{L_d(b)}{\max D}$  then break;
10      $bg_i \leftarrow b.block$ ;
11      $O_s \leftarrow$  furthest object in the current top- $k$ ;
12      $pruneDist \leftarrow \gamma \log_{1-\alpha} \frac{\epsilon}{outPR(bg_i)} + L_d(O_s)$ ;
13     if  $bg_i$  is not processed and  $\text{Dist}(Q, bg_i) < pruneDist$  then
14         do propagation on  $bg_i$  as in lines 8–23 of Algorithm 1;
15         foreach object  $o$  s.t.  $o \in bg_i$  and  $L_d(o) > 0$  do
16              $\text{CRS}(o) \leftarrow (1 - \beta) * (1 - \vec{p}(o)) + \beta * \frac{L_d(o)}{\max D}$ ;
17             update  $L_r$  with  $o$  and  $\text{CRS}(o)$ ;
18  $blockQueue \leftarrow \text{NewPriorityQueue}()$ ;
19 foreach block  $bg$  do  $blockQueue.\text{Enqueue}(bg)$ ;
20 while  $\|\vec{q}\|_1 \geq \epsilon$  do
21      $bg_i = blockQueue.\text{Dequeue}()$ ;
22      $O_s \leftarrow$  the furthest object in the current top- $k$ ;
23      $pruneDist \leftarrow \gamma \log_{1-\alpha} \frac{\epsilon}{outPR(bg_i)} + L_d(O_s)$ ;
24      $outPR(bg_i) \leftarrow 0$ ;
25     if  $\text{Dist}(Q, bg_i) < pruneDist$  then
26         do propagation on  $bg_i$  as in lines 8–23 of Algorithm 1;
27         foreach object  $o$  s.t.  $o \in bg_i$  and  $L_d(o) > 0$  do
28              $\text{CRS}(o) \leftarrow (1 - \beta) * (1 - \vec{p}(o)) + \beta * \frac{L_d(o)}{\max D}$ ;
29             update  $L_r$  with  $o$  and  $\text{CRS}(o)$ ;
30     if  $\text{CRS}(L_r(k)) < \text{CRS}(L_r(k + 1)) - (1 - \beta) * (\alpha^2 \max(\vec{q}) + (1 - \alpha)\|\vec{q}\|_1)$  then
31         break;
32      $blockQueue.\text{Update}()$ ;
33 return  $L_r$ ;
    
```

The pseudo-code of Early Stop EBC (ES-EBC) is shown in Algorithm 2. The variables $\vec{p}$, $\vec{q}$, and $outPR$ are as in Algorithm 1. We use a priority queue L_r with the ranking score as its key to keep track of the current top- $(k+1)$ objects, and we use a vector $\vec{L}_d$ to store the distances of the objects to the query. We use an R*-tree index to incrementally find the next nearest object b to query Q (line 7). Termination occurs when the smallest possible ranking score (its distance to query) of the next object is larger than the current ranking score of the k -th object (lines 8–9). We check whether the graph block containing b can be pruned according to Theorem 3.2 (lines 10–12). If the block graph bg_i cannot be pruned, we do the propagation within the block using the same propagation mechanism as in lines 8–23 of Algorithm 1, and maintain the top- $(k+1)$ objects. If object o is a nearest neighbor that has been accessed (*i.e.*, $\vec{L}_d(o) > 0$), we compute its ranking score and update L_r (lines 15–17). After we find the set of candidate top- k objects, the algorithm proceeds to propagate the outgoing PR of each block in the graph (lines 18–32). When the upper bound of the k -th object is smaller than the lower bound of the $(k+1)$ -st object (line 30), we will stop the propagation and return the top- k objects in L_r according to Theorem 3.1. Note that although Theorem 3.2 is used in line 12, it is also applied in line 23 to prune block graphs since $outPR(bg_i)$ can be changed with the propagation.

This algorithm does not require any pre-processing, and thus only the graph itself and the R-tree need to be stored in the disk.

3.2.3 Subgraph-Based EBC Algorithm (S-EBC)

3.2.3.1 Overview of the Algorithm

Recall that the baseline and ES-EBC algorithms need to propagate PR scores on the whole graph. To compute PR scores within some selected subgraphs, we develop several techniques.

First, we show that PR scores can be computed by the combination of two parts: the PR score propagation within a subgraph and the PR contributed by the propagations from other subgraphs, which can be computed from pre-computed distribution vectors of the border nodes that connect the subgraph with other subgraphs (see Section 3.2.3.2). This enables us to compute PR scores with respect to a subgraph.

Second, we propose an approach to identifying the subgraphs that need to be checked to find the top- k results, thus avoiding checking all subgraphs. This is enabled by a novel approach to estimating upper bound PR scores of objects in a subgraph, given a query.

Specifically, we organize the spatial objects by extending the external memory IR-tree [30] as introduced in Section 2.4.2.1. The spatial objects are grouped into subgraphs so that each subgraph corresponds to a leaf node of the IR-tree. We enrich the nodes of

the tree with pre-computed information (see Section 3.2.3.3) and show that by utilizing this information, we can compute an upper bound PR score at each node for a query.

The upper bound PR scores together with the distance of a subgraph to the query are used to choose which subgraphs to process at query time. If the best estimated ranking score of nodes in a subgraph exceeds (the smaller a score, the better) the score of the k -th object, the subgraph cannot contribute to the top- k results and can be pruned.

Based on this, we propose an approximate algorithm with performance guarantees for answering k PSK queries (see Section 3.2.3.4). The approximation occurs because we do not process all subgraphs.

3.2.3.2 Subgraph-Based PR Scoring

We present a decomposition method that enables us to compute PR scores with regard to subgraphs.

Assume that we have already partitioned the graph $\mathcal{G}$ into m subgraphs $\mathcal{G}_1, \dots, \mathcal{G}_m$ (to be discussed in Section 3.2.3.3). Let $\text{border}(\mathcal{G}_i)$ be the set of border objects of $\mathcal{G}_i$ that connect $\mathcal{G}_i$ with other subgraphs. Also, let H be the set of the border objects of all subgraphs, *i.e.*, $H = \bigcup_{i \in [1, m]} \text{border}(\mathcal{G}_i)$.

For each border object b , we pre-compute and store a vector $\vec{\text{GP}}_b$ that describes how to distribute the unit initial PR score from b over the whole graph. Note that the number of border objects is much smaller than the number of objects in the database. In a vector $\vec{\text{Pr}}_b$, most of the elements are 0 since in spatial graphs, a node b usually only affects the objects in nearby subgraphs. We do not need to store the value 0.

We proceed to show that the PR score vector of an object, which is assigned the unit initial PR score, can be computed by propagation within its subgraph together with the propagations contributed by the border nodes, which we capture in pre-computed PR score vectors of border nodes. We denote the PR scores computed within a subgraph as the local PR scores ($\vec{\text{LP}}$), and we denote the PR scores on the whole graph as the (global) PR scores ($\vec{\text{Pr}}$). We have the following lemma and theorem:

Lemma 3.3 *Given a node b and a subgraph $\mathcal{G}_i$ containing object b , we can compute the PR score vector of b as follows:*

$$\vec{\text{Pr}}_b = \vec{\text{LP}}_b + \sum_{h \in \text{border}(\mathcal{G}_i)} \vec{\text{AP}}_b(h) \cdot \vec{\text{Pr}}_h,$$

where $\vec{\text{AP}}_b(h)$ is the accumulated PR score of border node h during the local propagation within subgraph $\mathcal{G}_i$.

Proof: We have that $\vec{\text{Pr}}_b = \vec{\text{Pr}}_b(\mathcal{G}_i) + \vec{\text{Pr}}_b(\mathcal{G} - \mathcal{G}_i)$, where $\vec{\text{Pr}}_b(\mathcal{G}_i)$ represents the distribution to nodes excluding the border nodes in $\mathcal{G}_i$, and $\vec{\text{Pr}}_b(\mathcal{G} - \mathcal{G}_i)$ represents the distribution to the rest nodes.

When we finish distributing the PR in subgraph $\mathcal{G}_i$, all the border nodes in $\mathcal{G}_i$ hold their accumulated PR scores that have not yet distributed to other subgraphs. The nodes in other subgraphs ($\mathcal{G} - \mathcal{G}_i$) are affected by these border nodes, and thus $\vec{\text{Pr}}_b(\mathcal{G} - \mathcal{G}_i) = \sum_{h \in \text{border}(\mathcal{G}_i)} \vec{\text{AP}}_b(h) \cdot \vec{\text{Pr}}_h(\mathcal{G} - \mathcal{G}_i)$. The PR scores of the nodes in $\mathcal{G}_i$ come from two parts: the PR distribution within $\mathcal{G}_i$, and the distribution of the accumulated PR scores of the border nodes in $\mathcal{G}_i$. Hence, we have $\vec{\text{Pr}}_b(\mathcal{G}_i) = \vec{\text{LP}}_b + \sum_{h \in \text{border}(\mathcal{G}_i)} \vec{\text{AP}}_b(h) \cdot \vec{\text{Pr}}_h(\mathcal{G}_i)$. We get the proof by adding $\vec{\text{Pr}}_b(\mathcal{G}_i)$ and $\vec{\text{Pr}}_b(\mathcal{G} - \mathcal{G}_i)$.

Theorem 3.3 Given a query Q , its PR score vector is computed as:

$$\vec{\text{Pr}}_Q = \sum_{j=1}^m \sum_{o \in \mathcal{G}_j} \text{Sim}(Q, o) (\vec{\text{LP}}_o + \sum_{h \in \text{border}(\mathcal{G}_j)} \vec{\text{AP}}_o(h) \cdot \vec{\text{Pr}}_h)$$

$\text{Sim}(Q, o)$ is the similarity of query Q and the description of object o according to the vector space model.

Proof: Similar to the linearity property of BCV [10], the linearity property also holds for the PR score vector. Hence, we can compute $\vec{\text{Pr}}_Q$ as follows:

$$\vec{\text{Pr}}_Q = \sum_{O \in \mathcal{D}} \text{Sim}(Q, O) \vec{\text{Pr}}_O$$

According to Lemma 3.3, we obtain:

$$\begin{aligned} \vec{\text{Pr}}_Q &= \sum_{j=1}^m \sum_{O \in \mathcal{G}_j} \text{Sim}(Q, O) \vec{\text{Pr}}_O \\ &= \sum_{j=1}^m \sum_{O \in \mathcal{G}_j} \text{Sim}(Q, O) (\vec{\text{LP}}_O + \sum_{h \in \text{border}(\mathcal{G}_j)} \vec{\text{AP}}_O(h) \cdot \vec{\text{Pr}}_h) \end{aligned}$$

Theorem 3.3 allows us to decompose the computation of PR scoring. Given a query Q and a subgraph, we compute the text relevance to Q of each object in the subgraph. We distribute these scores following the links within the subgraph: when we reach a border node, the node accumulates the value distributed to it; if we meet a non-border node in the subgraph, we increase its PR score by a portion of the scores and distribute the rest to its out-neighbors, as in Algorithm 1. Having processed all subgraphs (*i.e.*, we distribute PR scores within each subgraph), we use the global PR vector of each border object in a subgraph to update the PR scores for all the objects according to Theorem 3.3.

3.2.3.3 Indexing and Ranking Score Estimation

To efficiently process an k PSK query, we need to organize the spatial objects into subgraphs. We proceed to briefly introduce the index structure used for organizing objects and then focus on how to estimate an upper bound PR score for each node for a single-term as well as a multi-term query.

Index structure—IR-tree. We extend the IR-tree [30] index structure to organize spatial objects and capture the pre-computed information needed for upper bound estimation. It is also used to partition the graph.

A leaf node L in the IR-tree contains a number of entries of the form $\langle o, o.\lambda \rangle$, where o is the identifier of an object and $o.\lambda$ is the bounding rectangle of the object. A leaf node corresponds to a subgraph and contains a pointer to the subgraph at the node and the global PR score vector of the border objects of the subgraph.

A non-leaf node R contains a number of entries of the form $\langle cp, cp.\lambda \rangle$, where cp points to a child node and $cp.\lambda$ is the minimum bounding rectangle of all rectangles of entries in the child node.

Each node contains a pointer to an inverted file that describes the objects in the subtree rooted at the node. The inverted file for a node X contains: 1) A vocabulary of all distinct terms in the text descriptions of the objects in the subtree rooted at X . 2) A set of posting lists, each of which relates to a term t . Each posting list is a sequence of pairs $\langle cp, wt_{cp,t} \rangle$, where cp is a child of X and $wt_{cp,t}$ is the upper bound PR score of objects in the subtree rooted at cp for term t .

It is challenging to develop an effective approach to estimating the upper bound PR score of a node for a given k PSK query. It is hard even for a single-keyword query. It is computationally prohibitive to pre-compute the exact PPVs or BCVs for each object [10, 56], and this also holds for PR score vectors since they need similar computation. Note that the purpose of pre-computing and storing the upper bounds is that we can then utilize them to prune the search space at query time for an k PSK query (this will become clear shortly).

Upper bound PR score for single-keyword queries. We first consider a leaf node; it is straightforward to derive an upper bound for a non-leaf node from those of its child nodes. We estimate the upper bound for a leaf node by the sum of the upper bound PR scores from the propagation within the node and the maximum contribution from other subgraphs. Let L be a leaf node that corresponds to a subgraph $\mathcal{G}_i$. Let $\max\text{GPr}(t, L)$ denote the estimated upper bound PR score of the objects in L for (query) term t . To estimate $\max\text{GPr}(t, L)$, we need the initial PR score of a subgraph $\mathcal{G}_i$ for a query term t , denoted as $\text{IPS}(t, \mathcal{G}_i)$.

Definition 3.1 The initial PR score of $\mathcal{G}_i$ for t is computed as follows:

$$\text{IPS}(t, \mathcal{G}_i) = \sum_{o \in \mathcal{G}_i} \text{Sim}(t, o)$$

Here, $\text{Sim}(t, o)$ is the similarity between term t and the description of object o according to the Vector Space Model.

We first present a lemma on how to estimate a maximum local PR score within a subgraph $\mathcal{G}_i$ given a query term t . This is the PR score without considering the effects of other subgraphs.

Lemma 3.4 The maximum local PR score $\text{maxLPr}(t, \mathcal{G}_i)$ can be estimated as:

$$\text{maxLPr}(t, \mathcal{G}_i) = \frac{1 + \alpha - \alpha^2}{2 - \alpha} \max_{o \in \mathcal{G}_i} \{\text{Sim}(t, o)\} + \frac{1 - \alpha}{2 - \alpha} \text{IPS}(t, \mathcal{G}_i)$$

Here, $\max_{o \in \mathcal{G}_i} \{\text{Sim}(t, o)\}$ is the largest initial PR score in subgraph $\mathcal{G}_i$ for term t .

Proof: The largest PR score is generated in the following situation: One node has the largest initial PR $\max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\}$. It distributes this value, and its out-neighbors gain at most $\text{IPS}(t, \mathcal{G}_i) - \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\}$ PR; then the out-neighbors propagate at most $(1 - \alpha) (\text{IPS}(t, \mathcal{G}_i) - \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\})$ to the node, and the node keeps the fraction α of this value, and it distributes the fraction $(1 - \alpha)$. In the next propagation, the out-neighbors send back $(1 - \alpha)^3 (\text{IPS}(t, \mathcal{G}_i) - \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\})$. This process continues until no PR needs to be propagated. The total PR the node finally holds is:

$$\begin{aligned} \text{maxLPr}(t, \mathcal{G}_i) &= \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\} + \alpha((1 - \alpha) + \dots + (1 - \alpha)^{2n+1} + \dots) \\ &\quad (\text{IPS}(t, \mathcal{G}_i) - \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\}) \\ &= \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\} + \frac{\alpha * (1 - \alpha)}{1 - (1 - \alpha)^2} (\text{IPS}(t, \mathcal{G}_i) - \alpha \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\}) \\ &= \frac{1 + \alpha - \alpha^2}{2 - \alpha} \max_{O \in \mathcal{G}_i} \{\text{Sim}(t, O)\} + \frac{1 - \alpha}{2 - \alpha} \text{IPS}(t, \mathcal{G}_i) \end{aligned}$$

Based on this lemma, we get the following theorem.

Theorem 3.4 We estimate $\text{maxGPr}(t, \mathcal{G}_i)$, the global upper bound PR score of an object in $\mathcal{G}_i$ for t , as follows:

$$\text{maxGPr}(t, \mathcal{G}_i) = \text{maxLPr}(t, \mathcal{G}_i) + \sum_{j \neq i} \text{IPS}(t, \mathcal{G}_j) \cdot \max_{bn \in \text{border}(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{\text{Pr}}_{bn}(b)\}$$

$\text{IPS}(t, \mathcal{G}_i)$ is computed according to Definition 3.1, and $\text{IPS}(t, \mathcal{G}_j) \cdot \max_{bn \in \text{border}(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{\text{Pr}}_{bn}(b)\}$ represents the maximum possible PR score that can be propagated from $\mathcal{G}_j$ to a node in $\mathcal{G}_i$.

Proof: According to Lemma 3, the largest PR score from the propagation within $\mathcal{G}_i$ is $\max \text{LPr}(t, \mathcal{G}_i)$. We next consider the effect of other subgraphs. The maximum effect of subgraph $\mathcal{G}_j$ on $\mathcal{G}_i$ occurs if a certain border node in $\mathcal{G}_j$ gets the initial prestige $\text{IPS}(t, \mathcal{G}_i)$. This is because all the effect of $\mathcal{G}_j$ on $\mathcal{G}_i$ is from border nodes. Each global PR vector of a border node in $\mathcal{G}_j$ describes its effect on nodes in $\mathcal{G}_i$. We find the largest value from all the global PR vectors of border nodes, i.e., $\max_{bn \in \text{border}(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{\text{Pr}}_{bn}(b)\}$, and this is the maximum possible PR that can be propagated from $\mathcal{G}_j$ to a node in $\mathcal{G}_i$. Combining the two parts completes the proof.

We can now explain the weight $wt_{cp,t}$ in the inverted file. At a leaf node X , the upper bound of each object cp is computed as $\text{Sim}(cp.\psi, t)$ (defined in Section 3.1); At the parent node X of a leaf node, $wt_{cp,t} = \max \text{GPr}(t, G_{cp})$ (computed according to Theorem 3.4). For other nodes, $wt_{cp,t}$ is the largest PR score among the child nodes of cp , i.e., $\max_{R \in cp.\text{children}()} \{wt_{t,R}\}$.

Upper bound PR score for multi-keyword queries. Based on the pre-computed upper bound PR score for a single keyword query, we propose an approach to estimating the upper bound PR score for a multi-keyword query.

Lemma 3.5 *Given a subgraph $\mathcal{G}_i$, we compute its initial PR score (IPS) for a query Q as follows:*

$$\text{IPS}(Q, \mathcal{G}_i) = \sum_{t \in Q.\psi \cap \mathcal{G}_i.\psi} \frac{w_{Q.\psi,t}}{W_{Q.\psi}} \text{IPS}(t, \mathcal{G}_i),$$

where $w_{Q.\psi,t}$ and $W_{Q.\psi}$ are defined in Section 3.1 and $\text{IPS}(t, \mathcal{G}_i)$ is computed according to Definition 3.1.

Proof:

$$\begin{aligned} \text{IPS}(Q, \mathcal{G}_i) &= \sum_{O \in \mathcal{G}_i} \text{Sim}(Q, O) \\ &= \sum_{O \in \mathcal{G}_i} \sum_{t \in Q.\psi \cap O.\psi} \frac{w_{Q.\psi,t} w_{O.\psi,t}}{W_{Q.\psi} W_{O.\psi}} \\ &= \sum_{t \in Q.\psi \cap \mathcal{G}_i.\psi} \sum_{O \in \mathcal{G}_i} \frac{w_{Q.\psi,t} w_{O.\psi,t}}{W_{Q.\psi} W_{O.\psi}} \\ &= \sum_{t \in Q.\psi \cap \mathcal{G}_i.\psi} \frac{w_{Q.\psi,t}}{W_{Q.\psi}} \sum_{O \in \mathcal{G}_i} \frac{w_{O.\psi,t}}{W_{O.\psi}} \end{aligned}$$

and we know $\text{Sim}(t, O) = \frac{w_{t,t} w_{O.\psi,t}}{w_{t,t} W_{O.\psi}} = \frac{w_{O.\psi,t}}{W_{O.\psi}}$, hence:

$$\text{IPS}(Q, \mathcal{G}_i) = \sum_{t \in Q.\psi \cap \mathcal{G}_i.\psi} \frac{w_{Q.\psi,t}}{W_{Q.\psi}} \sum_{O \in \mathcal{G}_i} \text{Sim}(t, O) = \sum_{t \in Q.\psi \cap \mathcal{G}_i.\psi} \frac{w_{Q.\psi,t}}{W_{Q.\psi}} \text{IPS}(t, \mathcal{G}_i)$$

Lemma 3.5 provides a way of computing the initial PR scores of a query Q in a subgraph $\mathcal{G}_i$. We proceed to present how to estimate the upper bound PR score of each node in the IR-tree for a query Q .

Definition 3.2 *Given a query Q and a node X in an IR-tree, the largest possible PR score of objects in X , $\max\text{Pr}(Q, X)$, is defined as:*

$$\max\text{Pr}(Q, X) = \sum_{t \in Q.\psi \cap X.\psi} \frac{w_{Q.\psi, t}}{W_{Q.\psi}} \max\text{GPr}(t, X),$$

where $w_{Q.\psi, t}$ and $W_{Q.\psi}$ are defined in Section 3.1.

Theorem 3.5 *Given a query Q and a leaf node X that encloses a set of objects $XO = \{o_1, \dots, o_m\}$, the following holds:*

$$\forall o \in XO; (\max\text{Pr}(Q, X) \geq \text{Pr}(Q, o))$$

Proof: *Given any object $o \in XO$ and any term t , it holds true that $\max\text{GPr}(t, X) \geq \text{Pr}(t, o)$. Therefore,*

$$\max\text{Pr}(Q, X) = \sum_{t \in Q.\psi \cap X.\psi} \frac{w_{Q.\psi, t}}{W_{Q.\psi}} \max\text{GPr}(t, X) \geq \sum_{t \in Q.\psi \cap o.\psi} \frac{w_{Q.\psi, t}}{W_{Q.\psi}} \text{Pr}(t, o) = \text{Pr}(Q, o)$$

We proceed to present the *minimum spatial-PR score distance*, $\min\text{RS}$, which is needed for the query processing. Given a query Q and a node X in the IR-tree, the metric $\min\text{RS}$ offers a lower bound on the actual spatial-PR score distance between query Q and the objects in node X . This bound can be used to order and efficiently prune the search space in the index.

Definition 3.3 *Given a query Q and a node X , the minimum spatial-PR distance, denoted by $\min\text{RS}(Q, X)$, is defined as:*

$$\min\text{RS}(Q, X) = (1 - \beta)(1 - \max\text{Pr}(Q, X)) + \beta \frac{\text{Dist}(Q.\lambda, X.\Omega)}{\max D}$$

Here, $\max\text{Pr}(Q, X)$ is the upper bound PR score of objects in X for query Q (cf. Definition 3.2).

Theorem 3.6 *Given a query Q and a node X whose rectangle encloses a set of objects $XO = \{o_1, \dots, o_m\}$, the following is true:*

$$\forall o \in XO; (\min\text{RS}(Q, X) \leq \text{RS}(Q, o))$$

Proof: *Given any object $o \in XO$, we know that $\max\text{Pr}(Q, X) \geq \text{Pr}(Q, o)$, according to Theorem 3.5. Because o is contained in the region $X.\Omega$, we have $\text{Dist}(Q.\lambda, X.\Omega) \leq \text{Dist}(Q, o)$. Hence we get $\min\text{RS}(Q, X) \leq \text{RS}(Q, o)$.*

Figure 3.1: Example of the sub-graph partition, hub nodes, and extended IR-tree

The extended IR-tree used and the original IR-tree [30] share a similar data structure. However, the inverted files in the two indexes store different contents. The novelty of the extended IR-tree is its approach to estimating upper bound PR scores.

Example 3.1 Recall the example dataset in Figure 2.6. According to the textual relevance and distance between two objects, a graph can be built as shown in Figure 3.1. The graph is split into several subgraphs according to the IR-tree construction, and these subgraphs correspond to the leaf nodes R_1, R_2, R_3 , and R_4 . The objects o_1, o_2, o_3 , and o_4 are hub nodes, because they are connected to nodes in other subgraphs. On each hub node, we compute its global PR vector and store it in the disk.

3.2.3.4 Subgraph-Based EBC Algorithm (S-EBC)

We proceed to describe the S-EBC algorithm that exploits the techniques just presented. The main idea is to choose subgraphs that are more likely to contain top- k results for a query Q and then compute the PR scores in the selected subgraphs. For a node X in the IR-tree, we estimate its largest possible PR score according to Definition 3.2, and we compute its distance to the query. Thus, we can compute the smallest possible ranking score ($\text{minRS}(X, Q)$ in Definition 3.3; the smaller the score, the better).

We use a priority queue *queue* to keep track of the nodes that have yet to be visited; the smallest possible ranking score is used as the key. When the head of the queue is a leaf node, *i.e.*, its corresponding subgraph $\mathcal{G}_i$ has the lowest possible ranking score, we process the subgraph using the approach from Section 3.2.3.2. When the propagation within the subgraph completes, we have a local PR score for each object in $\mathcal{G}_i$, and we use the PR scores held by the subgraph's border objects to update the global PR scores of all the objects (for object o , according to Theorem 3.3).

We proceed to process the next subgraph using the priority queue. The processing continues until the smallest possible ranking score of the unvisited head node of the

priority queue exceeds the ranking score of the current k -th result; we can then stop since no unvisited object can become a top- k result.

It is guaranteed that the unprocessed subgraphs (leaf nodes) do not contain top- k objects. However, they may affect the PR scores of the current top- k objects. To ensure this effect is within a certain bound, some postprocessing is needed. When building an IR-tree, we append the following pre-computed information to each leaf node (subgraph $\mathcal{G}_i$) of the IR-tree: a set of the IDs of the subgraphs that affect $\mathcal{G}_i$, denoted by $\mathcal{G}_i.Near$ and a factor describing the maximum possible effect of a subgraph on an object in $\mathcal{G}_i$ (e.g., for a subgraph $\mathcal{G}_j$, the factor is $\max_{bn \in border(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{Pr}_{bn}(b)\}$, according to Theorem 3.4).

In the postprocessing, we find the set SS of subgraphs containing the current top- k objects. For each subgraph $\mathcal{G}_i$ in SS , we then find $\mathcal{G}_i.Near$, the set of subgraphs that affect the PR scores of objects in $\mathcal{G}_i$. For each subgraph $\mathcal{G}_j$ in $\mathcal{G}_i.Near$, if it is not yet processed, we compute its maximum possible effect on an object in $\mathcal{G}_i$, denoted by $\max EF(\mathcal{G}_j, \mathcal{G}_i)$, according to Theorem 3.7. We sort the subgraphs in $\mathcal{G}_i.Near$ in ascending order of $\max EF(\mathcal{G}_j, \mathcal{G}_i)$, and we then find the m -th subgraph for which $SumErr = \sum_{j=[1, m-1]} \max EF(\mathcal{G}_j, \mathcal{G}_i) < \sigma$ and $SumErr + \max EF(\mathcal{G}_m, \mathcal{G}_i) \geq \sigma$.

Then starting from the m -th subgraph, for each subgraph in the sorted $\mathcal{G}_i.Near$, we do local propagation and update the PR scores of objects. We then update the list of the current top- k objects. If new objects are in the top- k and their corresponding subgraphs are not in SS , we include these subgraphs in SS and repeat the above steps until we have processed all subgraphs in SS . The postprocessing ensures that the maximal possible error in the ranking score of each top- k object is smaller than the error bound σ .

Theorem 3.7 *Given a query Q , the maximum possible PR score that subgraph $\mathcal{G}_j$ can propagate to an object in $\mathcal{G}_i$ is:*

$$\max EF(\mathcal{G}_j, \mathcal{G}_i) = \text{IPS}(Q, \mathcal{G}_j) \max_{bn \in border(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{Pr}_{bn}(b)\}$$

Proof: *It holds that the maximum possible effect of $\mathcal{G}_j$ on $\mathcal{G}_i$ is a factor of $\max_{bn \in border(\mathcal{G}_j), b \in \mathcal{G}_i} \{\vec{Pr}_{bn}(b)\}$. Multiplied by the total prestige of $\mathcal{G}_j$, we can get the maximum PR that $\mathcal{G}_j$ can propagate to an object in $\mathcal{G}_i$.*

The pseudo-code of Subgraph-based EBC (S-EBC) is given in Algorithm 3. We use $\vec{p}$ to store the current PR score of each object, $\vec{s}$ to store the accumulated PR score (to be distributed) of the border nodes, and a priority queue *queue* to keep track of the nodes to be visited (lines 1–3). In each step, we dequeue a node X from *queue*. If the minimum possible ranking score *spRS* of this node exceeds the ranking score of the current k -th object, we terminate the algorithm and return the results (lines 7–8). If the node is a non-leaf node, we compute the smallest possible ranking score for each

of its child nodes and enqueue them in *queue* (lines 10–12). Otherwise, we process the subgraph corresponding to the leaf node X by doing the PR score propagation within the subgraph and accumulating the PR scores for border objects (lines 17–27). In lines 28–32 the global PR score vectors of border objects are used to distribute the accumulated PR scores at the border objects. We update the rankings of all the objects that have been accessed (lines 33–35) because the propagation in the current subgraph $\mathcal{G}_i$ may increase the PR of the objects in other subgraphs. To ensure that the maximal possible error in the ranking score of each top- k object is smaller than the error bound σ , we do postprocessing as discussed in Section 3.2.3.4 (line 36).

In this algorithm, besides the graph and the R-tree index, the pre-computed PR vectors of the hub nodes are also stored in the disk. For each hub node, the length of its PR vector is at most $|V|$. Denote the set of the hub nodes by H . The addition space cost is $O(|H| \cdot |V|)$. The number of the hub nodes is determined by the parameter ξ and γ that are used to build the graph. As the value of γ increases and the value of ξ decreases, the density of the graph becomes higher, and the number of hub nodes increases as well. This leads to more space cost.

3.3 Experimental Study

3.3.1 Experimental Settings

Algorithms. In addition to the two proposed algorithms, ES-EBC and S-EBC, we compare with the two baseline approaches in Section 3.2.1. As Baseline 1 outperforms Baseline 2 significantly, we only report results for Baseline 1 and refer to this as “Baseline.”

Data and queries. We use three datasets that are real or based on real datasets. Table 3.1 shows some properties of the datasets.

Dataset **GN** is from the U.S. Board on Geographic Names¹. An object is a location with a geographic name (*e.g.*, valley). Dataset **Web** is generated from two datasets. One is WEBSPAM-UK2007² that consists of a large number of web documents; the other is a spatial dataset containing the tiger Census blocks in Iowa, Kansas, Missouri, and Nebraska³. We randomly combine web documents and spatial objects to get the **Web** dataset. Dataset **Hotel** contains spatial objects that represent hotels in the US⁴. Each object has a location and a set of words that describe the hotel (*e.g.*, restaurant, pool). **Hotel** is a small dataset while **GN** is much bigger. The objects in both datasets have

¹<http://geonames.usgs.gov>

²<http://barcelona.research.yahoo.net/webspam/datasets/uk2007>

³<http://www.rtreeportal.org>

⁴<http://www.allstays.com>

Algorithm 3: S-EBC($index, k, Q$)

```

1   $\vec{p} \leftarrow \vec{0}, \vec{s} \leftarrow \vec{0}, seenObjects = \emptyset$ , and initialize a list  $L_r$ ;
2   $queue \leftarrow \text{NewPriorityQueue}()$ ;
3   $queue.Enqueue(index.rootNode, 0)$ ;
4  while not  $queue.Empty()$  do
5       $X \leftarrow queue.Dequeue()$ ;
6       $spRS \leftarrow \text{minRS}(Q, X)$ ;
7      if  $spRS \geq L_r(k)$  then
8          break;
9      if  $X$  is a non-leaf node then
10         foreach entry child in  $X$  do
11              $spRS \leftarrow \text{minRS}(Q, child)$ ;
12              $queue.Enqueue(child, spRS)$ ;
13     else
14         read the corresponding graph  $\mathcal{G}_i$  of  $X$ ;
15          $seenObjects \leftarrow seenObjects \cup \{o | o \in X\}$ ;
16          $\vec{q} \leftarrow \vec{u}(\mathcal{G}_i)$ ;
17         while  $\|\vec{q}\|_1 \geq \varepsilon$  do
18             Pick an object  $b$  in  $\mathcal{G}_i$ ;
19             if  $b \in \text{border}(\mathcal{G}_i)$  then
20                  $\vec{s}(b) \leftarrow \vec{s}(b) + \vec{q}(b)$ ;
21             else
22                  $\vec{p}(b) \leftarrow \vec{p}(b) + \alpha \vec{q}(b)$ ;
23                 foreach out-neighbor  $j$  of  $b$  do
24                      $\vec{q}(j) \leftarrow \vec{q}(j) + (1 - \alpha)C(b, j)\vec{q}(b)$ ;
25                  $\text{CRS}(b) \leftarrow (1 - \beta)(1 - \vec{p}(b)) + \beta \frac{\text{Dist}(Q, b)}{\text{maxD}}$ ;
26                 update the position of  $b$  in  $L_r$ ;
27              $\vec{q}(b) \leftarrow 0$ ;
28         foreach object  $o$  in  $\text{border}(\mathcal{G}_i)$  do
29             read the global PR vector  $\vec{GP}_o$  of object  $o$ ;
30              $\vec{p} \leftarrow \vec{p} + \vec{s}(o)\vec{GP}_o$ ;
31              $\text{CRS}(o) \leftarrow (1 - \beta)(1 - \vec{p}(o)) + \beta \frac{\text{Dist}(Q, o)}{\text{maxD}}$ ;
32             update the position of  $o$  in  $L_r$ ;
33         foreach node  $o$  in  $seenObjects$  do
34              $\text{CRS}(o) \leftarrow (1 - \beta)(1 - \vec{p}(n)) + \beta \frac{\text{Dist}(Q, o)}{\text{maxD}}$ ;
35             update the position of  $o$  in  $L_r$ ;
36 do the postprocessing and then return  $L_r$ ;
    
```

short descriptions. **Web** is a medium-sized dataset whose objects have long descriptions. We evaluate our approaches on these three different datasets.

We generate 4 query sets, in which the number of keywords is 1, 2, 3, and 4, respectively, in the space of **GN**, and we generate 4 similar query sets for the space of **Web**

Property	Hotel	Web	GN
Total number of objects	20,790	552,163	1,777,598
Total number of unique words	602	1,267,595	222,407
Total number of words	59,159	110,523,880	5,971,356
Maximum frequency of words	13,352	379,308	236,536
Maximum number of words attached to an object	52	12,060	752

Table 3.1: Dataset properties

and **Hotel**. Each set comprises 200 queries, and each query is randomly generated. We report average costs of the queries in each query set.

Setup. The IR-tree index structure is disk resident, and the page size is 16KB. The number of children of a node in the IR-tree is computed given the fact that each node occupies a page. This translates to 400 children per node in our implementation. The default values for parameters are as follows: k is 10, the number of query keywords is 2, α is 0.5 (Equation 3.2), and β is 0.5 (Equation 3.3) for all algorithms. S-EBC needs an extra parameter σ (to control its error bound; Section 3.2.3.4) that is set to 0.0001. Two threshold parameters for building graphs γ and ξ (Section 3.1) are set at 2 km and 0.5, respectively.

All algorithms were implemented in VC++, and run on an Intel(R) Xeon(R) CPU X5650 @2.66GHz with 4GB RAM.

3.3.2 Experimental Results

3.3.2.1 Experiments on GN

Varying k in k PSK. Figure 3.2 show the results of varying k when using the default settings for the other parameters.. Note that the y-axis uses a logarithmic scale.

Figure 3.2: Varying k

Figure 3.3: Varying # keywords

 Figure 3.4: Varying α

 Figure 3.5: Varying β

We can see that ES-EBC and S-EBC significantly outperform (by an order of magnitude) the baseline for all values of k . ES-EBC performs better than the baseline due to the early stopping of propagation and punning of cells during score propagation. S-EBC outperforms the other two methods since it computes PR scores *w.r.t.* selected subgraphs rather than the whole graph as do the other two methods. As expected, the runtimes of all the approaches increase slightly with increasing k .

Varying the number of keywords. Figure 3.3 shows that ES-EBC and S-EBC outperform the baseline for different numbers of keywords. All algorithms need more time as the number of keywords increases since they need to process more words.

Varying α . Figure 3.4 shows that ES-EBC and S-EBC significantly outperform the baseline for all values of α . We also note that the runtime increases as α decreases. This is because it takes longer for the propagation to converge with a smaller α .

Recall that parameter α can balance the relevance of an object versus the effect of its relevant neighbors. In particular, smaller values of α favor nodes with nearby relevant nodes, while larger values of α favor nodes with high initial PR scores. At one extreme, when $\alpha = 1$, the k PSK query is essentially the same as LkT query [30] that does not consider the effect of nearby relevant objects. Hence, we can see the overhead of considering the inter-relationships between objects by comparing with the runtime at $\alpha = 1$.

Varying β . Figure 3.5 shows the results. Parameter β in Equation 3.3 allows users to set their preferences between the PR score and spatial proximity. A large β means that the spatial distance is more important, while a small β means that the PR score is more important.

As expected, ES-EBC and S-EBC perform better for larger β —they benefit from spatial proximity being given higher weight. When spatial proximity is given very low weight, ES-EBC nearly cannot prune any cell, and S-EBC nearly cannot prune any subgraph and needs process the entire IR-tree.

 Figure 3.6: Varying k (**Web**)

 Figure 3.7: Varying # keywords (**Web**)

 Figure 3.8: Varying k (**Hotel**)

 Figure 3.9: Varying # keywords (**Hotel**)

3.3.2.2 Experiments on Web and Hotel

We give the results when varying k and the number of keywords on **Web** and **Hotel** in Figures 3.6-3.9. It is shown that ES-EBC and S-EBC significantly outperform the baseline on the two datasets. The results on the two datasets are consistent with those on **GN**.

3.3.2.3 Scalability.

To evaluate scalability, we generate 5 datasets containing from 2 to 10 million data points: we generate new locations by copying the locations in GN to nearby locations while maintaining the real distribution of the objects; and for each new location, a document is selected at random from the text descriptions of the objects in GN. Figure 3.10 shows that ES-EBC and S-EBC scale linearly with the size of the dataset, but that Baseline does not scale.

Figure 3.10: Scalability with data size

3.3.2.4 Space Requirements

Table 3.2 shows the total sizes of the index structures used by each method for data set GN. Baseline and ES-EBC use the same indexes (inverted list and R*-tree) and object graph. S-EBC needs more disk space to store the PR vectors for border objects and the inverted lists in non-leaf nodes. The inverted files in the leaf nodes of the IR-tree are roughly the inverted file used in the baseline approach.

Baseline	ES-EBC	S-EBC
201	201	1423

Table 3.2: Index structure sizes (MB)

3.3.2.5 Effectiveness of k PSK Queries

To study the utility of k PSK queries, we compare with the LkT query [30]. The difference between k PSK and LkT [30] is that the former considers the effect of nearby relevant objects, while the latter does not.

The utility of k PSK queries. The lack of a publicly available test data, including both annotated resources and relevant queries, renders the comparison of the different approaches particularly challenging. To enable comparison, we collected a real spatial data set from the region of Aalborg, Denmark using a local Yellow Page service (www.degulesider.dk), where each object has category (*e.g.*, restaurant, hotel) and a description; we geocoded the objects using the Google Maps API. The dataset contains 4,951 objects with a total of 39,505 descriptive words. This dataset has the benefit that we can find expert annotators for it, who are undergraduate students living in the area of Aalborg.

We randomly generate 50 locations in the space and ask annotators to choose keywords for each, thus obtaining 50 queries. To evaluate the quality of query results, we use

a well-known metric, the nDCG [55]. The top 5 objects returned by ES-EBC, S-EBC, and LkT [30] are merged into a single list, shuffled, and then given to three annotators for judgment. Numerical scores of 0, 1, 2, and 3 are collected and averaged to reflect the annotators' opinions as to whether an object belongs in the top 5.

In *kPSK* (ES-EBC and S-EBC), α and β are set as to their default value of 0.5, and in LkT [30], the parameter that balances distance and text relevance (corresponding to β in *kPSK*) is set to 0.5.

Table 3.3 depicts the results. Both ES-EBC and S-EBC perform significantly better than LkT queries that do not take into account the effects of nearby relevant objects. The approximate S-EBC algorithm performs slightly worse than ES-EBC.

	ES-EBC	S-EBC	LkT [30]
nDCG@5	0.8873	0.8524	0.7061

Table 3.3: Effectiveness of different algorithms

3.3.2.6 Effects of Parameters on Graph Building

Figures 3.11 and 3.12 show the runtime when we vary γ and ξ on Hotel. The runtime increases as we increase γ or decrease ξ . The reason is that the graph becomes denser, making it take longer time to propagate PR scores.

Figure 3.11: Varying threshold γ (**Hotel**) Figure 3.12: Varying threshold ξ (**Hotel**)

3.3.2.7 Summarization of Empirical Study

According to the experiments on the three datasets, i.e., **GN**, **Web**, and **Hotel**, we can get the following conclusions: ES-EBC and S-EBC significantly outperform the baseline method (by orders of magnitude) in terms of query processing time, and S-EBC achieves

the best efficiency. ES-EBC performs better than Baseline due to the early stopping of propagation and punning of cells during score propagation. S-EBC performs the best because it computes PR scores *w.r.t.* selected subgraphs rather than the whole graph as done in Baseline and ES-EBC. ES-EBC and S-EBC scale linearly with the size of the dataset, but the baseline method does not scale. Due to the PR score pre-computation, S-EBC requires more space than ES-EBC and Baseline do. According to the effectiveness study, the k PSK query can better satisfy users' needs by taking into account the effects of nearby relevant objects.

3.4 Additional Related Work

Personalized PageRank. Jeh and Widom [56] propose the personalized PageRank concept. In contrast to PageRank [14] that computes the global importance of nodes in a graph, personalized PageRank allows users to favor a set P of preferred nodes. The nodes in the preference set make a unit preference vector u where $u(p) = 1/|P|$ if $p \in P$ and $u(p) = 0$ if $p \notin P$, rather than distributing the unit preference score uniformly over all nodes in PageRank. Pre-computing and storing all PPVs is impractical, as is computing PPVs at query time, since the computation of PPV needs an iterative computation over the web graph.

Several algorithms [10, 45, 49, 56] have been proposed to compute the personalized PageRank vector (PPV).

Jeh and Widom propose a remarkable Hub Decomposition algorithm [56] that pre-computes the partial vectors for the nodes in a hub set of top-ranked pages. This algorithm can only compute the PPVs of the nodes in the hub set. To process the k PSK query, we need to pre-compute PR for every node, which renders the Hub Decomposition algorithm impractical in our problem.

Fogaras et al. [45] propose a fingerprint-based algorithm that simulates random walks. The idea is to compute and store short random walks from each node in order to compute PPVs at query time. This works well to compute random walks from every node in the graph. However, this cannot be applied to computing a random walk from an arbitrary node, which is prohibitive at query time. This renders the proposal impractical for computing the PR scores in our problem.

More recently, Berkhin proposes a bookmark-coloring algorithm (BCA) [10] that perhaps fits the best with our problem among the existing algorithms for computing PPVs. Its main idea is to diffuse scores in preference vector across the graph. A unit amount of score (called paint) is injected into a selected node (the bookmark node); a fraction of the paint is held by this node, and the rest flows by following the links of the graph. This propagation continues until the paint is distributed over the whole graph. In practice, the algorithm terminates when the paint to be distributed is smaller than a threshold.

PPVs are also used for keyword queries in entity-relation graphs. In ObjectRank [5], a PPV for each keyword in a graph database is pre-computed. However, it is impractical to pre-compute the PPVs for each keyword when the vocabulary size is large [19].

Chakrabarti [19] apply and extend PPVs to the keyword query on entity-relation graphs. This work is novel in how it chooses a set of nodes as hub nodes based on query logs; and it adopts the approach of Fogaras et al. [45] to store approximate PPVs in the form of fingerprints.

Block PageRank: There is a large body of work on global PageRank computation. Some works consider computations of global PageRank values over subgraphs (e.g., [6, 58]). The problem of computing global PageRank is different from computing Prestige-base Relevance, and these proposals are therefore not directly applicable to our problem.

A final note is that two works [32, 66] that employ the PageRank algorithm to do propagation on document similarity graphs focus on effectiveness without considering efficiency issues.

3.5 Conclusion

We propose a new type of query, the *kPSK* query, that retrieves the top-*k* geo-textual objects ranked according to both location proximity and so-called prestige-based relevance that considers both the text relevance of an object to a query and the presence of nearby objects that are relevant to the query. We develop two baseline algorithms and propose two new algorithms to process the *kPSK* query. Results of empirical studies on real data demonstrate the effectiveness of the *kPSK* query and the efficiency of the new algorithms.

Chapter 4

Spatial Group Keyword Query

This chapter is organized as follows: Section 4.1 formally defines the SGK query and shows the computational complexities of two instances of the problem. Section 4.2 presents an approximate algorithm and several exact algorithms for the first problem instance. Section 4.3 presents two approximate algorithms and two exact algorithms for the second problem instance. Section 4.4 describes how to process the top- k SGK query. Section 4.5 reports on the empirical studies. Finally, Section 4.6 covers the additional related work and Section 4.7 offers conclusions.

4.1 Problem Statement

Let S be a set of keywords. The keywords may capture user preferences or required project partner capabilities, depending on the application. Let D be a database consisting of m geo-textual objects. Each object o in D is associated with a location $o.\lambda$ and a set of keywords $o.\psi$, $o.\psi \subset S$, that describes this object (*e.g.*, the menu of restaurant or the skills of a possible project partner).

Definition 4.1 *Spatial Group Keyword Query:* A spatial group keyword (SGK) query q is in the form of $\langle q.\lambda, q.\psi \rangle$, where $q.\lambda$ is a location and $q.\psi$ represents a set of keywords. It finds a group of objects χ , $\chi \subseteq D$, such that $\cup_{r \in \chi} r.\psi \supseteq q.\psi$ and such that the cost $\text{Cost}(q, \chi)$ is minimized.

Definition 4.2 *Feasible Group:* Given a SGK query q , if a group of objects G can cover all the keywords in $q.\psi$ (*i.e.*, $\cup_{r \in G} r.\psi \supseteq q.\psi$), we call G a “feasible group” or a “feasible solution”.

We proceed to present cost functions. Given a set of objects χ , the cost function has two weighted components:

$$\text{Cost}(q, \chi) = \alpha C_1(q, \chi) + (1 - \alpha) C_2(\chi),$$

where $C_1(\cdot)$ is dependent on the distance of the objects in χ to the query object and $C_2(\cdot)$ characterizes the inter-object distances among the objects in χ .

This type of cost function is capable of expressing that result objects should be near the query location ($C_1(\cdot)$), that the result objects should be near each other ($C_2(\cdot)$), and that these two aspects are given different weights (α). We consider two instantiations of the cost function $\text{Cost}(q, \chi)$ that we believe match the intended applications well.

1. **SUM** cost function:

$$\text{Cost}(q, \chi) = \sum_{r \in \chi} (\text{Dist}(r, q)) \quad (4.1)$$

The cost function is the sum of the distance between each object in χ and the query location. This may fit with applications where the objects need to meet at the query location, such as incident handling or the finding of project partners.

2. **MAX+MAX** cost function:

$$\text{Cost}(q, \chi) = \alpha \max_{r \in \chi} (\text{Dist}(r, q)) + (1 - \alpha) \max_{r_1, r_2 \in \chi} (\text{Dist}(r_1, r_2)) \quad (4.2)$$

The first part of this cost function is the maximal distance between any object in χ and the query location q , and the second part is the maximum distance between two objects in χ (this can be understood as the diameter of the result). When there are multiple optimal groups of objects, we choose one group randomly. This second cost function may fit with applications such as tourists planning visits to several points of interest. For ease of presentation, we disregard parameter α in the rest of this chapter. But the proposed algorithms remain applicable when α is enabled.

We have the following theorem to demonstrate the hardness of answering the spatial group keyword query.

Theorem 4.8 *The problem of answering the spatial group keyword query using either a SUM or a MAX+MAX cost functions is NP-hard.*

Proof:

- (i) *We first consider the SUM cost function. We prove the theorem by a reduction from the weighted set cover problem. An instance of the weighted cover problem consists of a universe $U = \{1, 2, \dots, n\}$ of n elements and a family of sets $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$, where $S_i \subseteq U$ and each S_i is associated with a positive cost C_{S_i} . The problem is to find a subset $\mathcal{F}$ of $\mathcal{S}$ such that $\cup_{S_i \in \mathcal{F}} S_i = U$ and such that its cost $\sum_{S_i \in \mathcal{F}} (C_{S_i})$ is minimized.*

To reduce this problem to the problem of answering the SUM spatial group keyword, we observe that each element in U corresponds to a keyword in $q.\phi$, that each S_i corresponds to a spatial object containing a set of keywords, and that the weight of S_i is $\text{dist}(q, S_i)$. It is easy to show that there exists a solution to the weighted set cover problem if and only if there exists a solution to query q .

- (ii) *Considering next the MAX+MAX cost function, we prove the theorem by a reduction from the 3-SAT problem. An instance of the 3-SAT problem consists of $\Phi = C_1 \wedge C_2, \dots, \wedge C_l$, where each clause $C_j = x_j \vee y_j \vee z_j$, and $\{x_j, y_j, z_j\} \in \{e_1, \bar{e}_1, e_2, \bar{e}_2, \dots, e_n, \bar{e}_n\}$. The decision problem is to determine whether we can assign a truth value to each of the literals (e_1 through e_n) such that Φ is true.*

Next, we reduce this problem to an instance of the decision problem of answering the MAX+MAX spatial group keyword query, which is to decide if there exists a group with cost value at most C . The reduction is inspired by the proof for the hardness of the Multiple-Choice Cover (MCC) problem [4] (that is different from our problem).

Consider a circle with diameter d and with the query point q as its center, and let each variable e_i correspond to a point in the circle, while its negation $\bar{e}_i$ corresponds to the diametrically opposite on the circle. The distance between e_i and $\bar{e}_i$ is d . We set $d = \frac{2}{3}(C + 2\epsilon)$, where $\epsilon > 0$. We also set another value $d_1 = \frac{2}{3}(C - \epsilon)$, such that $d > d_1$ and we can set ϵ to be a very small value to make sure that d is sufficiently close to d_1 ; thus, the distance between any two points corresponding to different variables can be no larger than d_1 .

Each set S_i ($i \in [1, n]$) contains a pair of points e_i and $\bar{e}_i$, and the two points contain a distinct keyword in $q.\psi$. Each set S_j ($j \in [n+1, n+l]$) contains each triple of points corresponding to a clause C_{j-n} , and they contain a distinct keyword in $q.\psi$. Thus, to cover all keywords in $q.\psi$, a query result of q must contain one point from each S_i (i.e., e_i and $\bar{e}_i$), and it must contain at least one point from each S_j (corresponding to clause C_{j-n}).

Given this mapping, we can see that if there exists a truth assignment for Φ , we can find a group χ for the MAX+MAX SGK query. In this group, all the keywords in q are covered, and the cost can be computed as $\text{Cost}(q, \chi) = \max_{r \in \chi} \text{Dist}(r, q) + \max_{i, j \in \chi} \text{Dist}(i, j) = \frac{d}{2} + \max \text{Dist}(i, j), i, j \in \chi$, which indicates that a feasible solution χ with cost at most $\frac{d}{2} + d_1 = C$ exists. On the other hand, if there exists a subset of points on the circle whose diameter is at most d covering all the query keywords, there exists a truth assignment for the instance Φ . This completes the proof.

4.2 Processing SUM SGK Queries

An approximation algorithm is first presented in Section 4.2.1. The number of keywords of a query may be small in some applications, and this motivates us to develop an exact algorithm for processing the SUM SGK query. Without using an index, we present a dynamic programming algorithm in Section 4.2.2, and an improved dynamic programming

algorithm in Section 4.2.3. A dynamic programming algorithm that uses an index to prune the search space is described in Section 4.2.4.

4.2.1 Approximation Algorithm

We showed that the problem of answering the SUM SGK query is NP-hard by a reduction from the Weighted Set Cover (WSC) problem in Theorem 4.8. The reduction in the proof is approximation preserving. Thus, the approximation properties of the WSC problem carry over to our problem.

For the WSC problem, it is known (see [28]) that a greedy algorithm is an H_k -approximation algorithm for the weighted k -set cover, where $H_k = \sum_{i=1}^k \frac{1}{i}$ is the k -th harmonic number. In our problem, k is the number of query keywords. Thus, we can adapt the greedy algorithm to process the SUM spatial group keyword query.

A straightforward method of adapting the greedy algorithm is to decompose the given user query q dynamically into a sequence of partial queries, each containing a different set of keywords depending on the preceding partial queries, and then to evaluate these partial queries. Specifically, we start with the user query q , which can be regarded as the first partial query, and we find the object with the lowest cost that covers part or all of the keywords in q . The object is added to the result set. The uncovered keywords in q form a new partial query with the same spatial location as q . We then find an object with the lowest cost that covers part or all of the keywords in the new partial query. This process continues until all keywords are covered or some keyword cannot be covered by any object. This method needs to scan the dataset multiple times, once for each partial query.

To avoid multiple scans, we propose a greedy algorithm on top of the IR-tree. We proceed to focus on two aspects of the idea that are important to the performance: (1) how to find the object with the lowest cost for each partial query using the IR-tree, and (2) whether we can reuse the computation for the preceding partial query when computing the next partial query.

Given a partial query q_s , we adopt the best-first strategy to traverse the IR-tree. We use a min-priority queue to maintain the intermediate results. The key of the queue is the cost of each element. The cost of an object o is computed by $\frac{\text{Dist}(o, q)}{|o.\psi \cap q_s.\psi|}$; the cost of a node entry e is computed by $\frac{\text{minDist}(e, q)}{|e.\psi \cap q_s.\psi|}$, where $\text{minDist}(e, q)$ represents the minimum distance between q and e .

Lemma 4.6 *Given a partial query q_s and an IR-tree, the cost of a node is a lower bound of the cost of any of its child nodes.*

Proof: *Given a node e and any of its child nodes e' , we have $\text{minDist}(e, q) \leq \text{minDist}(e', q)$, and $|e.\psi \cap q_s.\psi| \geq |e'.\psi \cap q_s.\psi|$.*

Lemma 4.6 says that the cost of a node is a lower bound of the costs of all objects in the subtree rooted at the node. Thus, if the cost exceeds that of some object that has been visited, we can disregard all objects in the subtree for q_s . This guarantees the correctness of the best-first strategy for finding an object with the lowest cost for a partial query q_s .

We next discuss whether we can reuse the computation for preceding partial queries. An obvious method is to process each partial query from scratch. However, this incurs repeated computation when a node or an object is visited for multiple times. To avoid this, we divide the entries (corresponding to leaf and non-leaf nodes) in the priority queue into two parts: (1) the entries that have already been visited when processing previous partial queries, and (2) the entries that have not yet been visited.

Lemma 4.7 *The elements in the priority queue that have been visited when processing previous partial queries can be disregarded when processing a new partial query.*

Proof: *The keyword set of a previous partial query is a superset of the keyword set of a new partial query. For a visited node, all its entries containing keywords of the new partial query have been enqueued into the priority queue; thus, we can disregard the elements that have been visited.*

The pseudocode is outlined in Algorithm 4. The algorithm uses a min-priority queue for the best-first search with the cost as the key. Variable $mSet$ keeps the keyword set of the current partial query, and $pSet$ keeps the keyword set of the preceding partial query. For each partial query, we use the best-first search to find an object that overlaps with the query keyword $mSet$ and has the lowest cost.

Whenever the algorithm pops an object from U , it is guaranteed that the text description of the object overlaps with $mSet$ (the keyword set of the current partial query), and that the object has the lowest cost. Thus, it becomes part of the result. The algorithm proceeds with the next partial query by changing the keyword component (line 11). Based on Lemma 4.7, we do not need to scan all objects to process the new partial query. Rather, we only have to update the unvisited elements in the priority queue with the new cost based on the new partial query (lines 12–15). We then use the best-first search to process the new partial query.

Assume that there are n query keywords, and the size of the queue used in the algorithm is at most L . The cost of maintaining the queue is $O(L \log L)$. Next, at most n object are dequeued from the queue, and we need to reorganize the queue when an object is dequeued, which costs $O(nL \log L)$. For the other objects, we only need to remove them from the queue, which costs at most $O(\log L)$. Thus the total worst case complexity is $O(nL \log L)$. Note that L is at most the number of IR-tree nodes plus the number of objects in the dataset.

Algorithm 4: SUM-Appro($q, irTree$)

```

1  $U \leftarrow$  new min-priority queue;
2  $U.Enqueue(irTree.root, 0)$ ;
3  $Group \leftarrow \emptyset$ ;  $Cost \leftarrow 0$ ;
4  $mSet \leftarrow q.\psi$ ;  $pSet \leftarrow q.\psi$ ;
5 while  $mSet \neq \emptyset$  and  $U$  is not empty do
6    $e \leftarrow U.Dequeue()$ ;
7    $Cost \leftarrow Cost + e.Key$ ;
8   if  $e$  is an object then
9      $Group \leftarrow Group \cup e$ ;
10     $pSet \leftarrow mSet$ ;
11     $mSet \leftarrow mSet \setminus e.\psi$ ;
12    for each entry  $e'$  in  $U$  do
13      if  $e'.\psi \cap e.\psi \neq \emptyset$  then  $e'.key = \frac{e'.key * |e'.\psi \cap mSet|}{|e'.\psi \cap pSet|}$ ;
14      else remove  $e$  from  $U$ ;
15    reorganize the priority queue  $U$  using new key values;
16  else
17    for each entry  $e'$  in node  $e$  do
18      if  $mSet \cap e'.\psi \neq \emptyset$  then
19        if  $e$  is a non-leaf node then  $dist \leftarrow \text{minDist}(e', q)$ ;
20        else  $dist \leftarrow \text{Dist}(o, q)$ ;
21         $U.Enqueue(e', \frac{dist}{|mSet \cap o.\psi|})$ ;
22 return  $Cost$  and  $Group$ ; // results

```

4.2.2 Exact Algorithm Without an Index

An obvious exact algorithm enumerates every subset of spatial objects whose text descriptions overlap with the query keyword set in D . For each such subset, the algorithm then checks whether the subset covers all query keywords and computes its cost. This yields an exponential running time in terms of the number of objects, which is very expensive if D is large.

A better method is to perform the exhaustive search on a smaller set of objects. We proceed to introduce a lemma that lays the foundation for the algorithm to be proposed.

Lemma 4.8 Consider a query q and two objects o_i and o_j , each of which contains a subset of the query keywords. Let $ws_i = q.\psi \cap o_i.\psi$ and $ws_j = q.\psi \cap o_j.\psi$. If $\text{Dist}(o_i, q) < \text{Dist}(o_j, q)$, $\{o_i\}$ is a better group than $\{o_j\}$ for any keyword subset of $ws_i \cap ws_j$.

Proof: The proof is obvious since o_i always incurs lower cost than does o_j for any keyword subset $ws_i \cap ws_j$.

According to the lemma, given a subset of query keywords ws , among the objects covering ws , the one that is the closest to the query contributes the lowest cost to ws .

Example 4.2 Consider a query q with keywords $q.\psi = \{t_1, t_2, t_3\}$ and the four objects in Table 4.1. We know that $\text{Dist}(o_1, q) < \text{Dist}(o_2, q)$ and $o_1 \cap o_2 = \{t_2\}$. According to lemma 4.8, $\{o_1\}$ is a better result set for the query with keyword set $\{t_2\}$.

	o_1	o_2	o_3	o_4
Distance to the query	1	2	2.5	4
Keywords	t_1, t_2	t_2, t_3	t_1, t_3	t_1

Table 4.1: Example dataset

Since the set of query keywords is small, the number of its subsets is not large although it is exponential to the number of query keywords. For each subset of query keywords, we find the object that covers the subset of query keywords and has the lowest costs. We denote the set of these objects as $objSet$, and we can find the optimal group according to the following lemma:

Lemma 4.9 Given a SUM SGK query q , the optimal group must be a subset of $objSet$.

Proof: This can be proved by contradiction. Suppose that an object $o_m \notin objSet$ is contained in the optimal group. It must be true that there exists an object $o_i \in objSet$ which contains the same keywords as o_m does. However, according to Lemma 4.8, o_i is better than o_m w.r.t. $o_m.\psi$. Hence, we can replace o_m with o_i in the optimal group to obtain a better cost, which leads to a contradiction.

Based on Lemma 4.9, we can do an exhaustive search on these objects to find the best group. Unfortunately, this method is time-consuming since it runs exponentially in terms of the number of objects in $objSet$, which can be exponential in the number of query keywords. The time complexity of this method is $O(2^{|objSet|})$. Suppose there are n keywords in the keyword component of a query q , at most $(2^n - 1)$ objects need to be considered, and thus its time complexity is at most $O(2^{(2^n - 1)})$.

In this chapter, we develop a dynamic programming algorithm with exponential running time in terms of the number of query keywords. The idea of the algorithm is summarized as follows: Given a query q with n ($= |q.\psi|$) keywords, we process the subsets of $q.\psi$ in breadth-first order, *i.e.*, we process subsets in ascending order of their length. For each subset X of $q.\psi$, we find the best set of covering objects, *i.e.*, a set of objects that cover X and have the lowest cost, by utilizing the best covering sets of the subsets of X .

Existing WSC algorithms are mostly approximation algorithms. Several recent proposals [12,33] have good (*e.g.*, constant) approximation guarantees with moderately exponential running time. Björklund et al. [11] propose an exact algorithm for the unweighted set cover problem using the inclusion-exclusion principle, which is not directly applicable to the WSC problem.

Formally, let F be the set of all subsets of $q.\psi$. For each subset $X \subseteq q.\psi$, with a bit abuse of notation, we denote the set of objects that cover X and has the lowest cost by $\text{Group}(X)$ and the cost of covering set X by $\text{Cost}(X)$.

Our dynamic programming algorithm avoids enumerating all the set partitions. Equation 4.3 shows the approach to computing the lowest cost for each subset X . If X is not covered by any object o , its cost is initialized to ∞ ; otherwise, its cost is initialized to the cost of the best covering object, as shown in Equation 4.3. Then the dynamic programming idea is adopted to find the lowest cost of each subset X in ascending order of the length of X . Specifically, for each X , we check each pair of component subsets (whose optimal costs are already known) to find a pair with the lowest cost. Note that the optimal set of two subsets may share some objects whose costs are computed by oCost .

$$\text{Cost}(X) = \begin{cases} \min_{o \in D \wedge X \subseteq o.\psi} \{\text{Dist}(o, q)\}, & \exists o (X \subseteq o.\psi) \\ \infty, & \text{otherwise} \end{cases} \quad (4.3)$$

$$\text{Cost}(X) = \min_{S \in F \wedge S \subseteq X} \{\text{Cost}(X \setminus S) + \text{Cost}(S) - \text{oCost}(S, X \setminus S)\}$$

$$\text{oCost}(S, X \setminus S) = \sum_{o \in \text{Group}(X \setminus S) \cap \text{Group}(S)} \text{Dist}(o, q) \quad (4.4)$$

$$\text{Group}(X) = \begin{cases} \arg \min_{o \in D \wedge X \subseteq o.\psi} \{\text{Dist}(o, q)\}, & \exists o (X \subseteq o.\psi) \\ \emptyset, & \text{otherwise} \end{cases} \quad (4.5)$$

$$S^* = \arg \min_{S \in F \wedge S \subseteq X} \{\text{Cost}(X \setminus S) + \text{Cost}(S) - \text{oCost}(S, X \setminus S)\}$$

$$\text{Group}(X) = \text{Group}(S^*) \cup \text{Group}(X \setminus S^*)$$

To implement the algorithm efficiently, we encode the subsets. A query q with n keywords $q.\psi = \{t_1, t_2, \dots, t_n\}$ has $(2^n - 1)$ non-empty subsets of keywords. We encode each subset X by an integer i of n bits, where each bit corresponds to a keyword in q . If the j^{th} keyword is contained in X , the j^{th} bit in the binary format of i is set to 1; otherwise, it is set to 0. For example, for $q.\psi = \{t_1, t_2, t_3\}$, we can encode subset $\{t_1\}$ by 1, $\{t_2\}$ by 2, and $\{t_1, t_2\}$ by 3.

We maintain two arrays: $\text{Cost}[i]$ records the lowest cost of the subset that is encoded by integer i , and $\text{Group}[i]$ records the group of objects that contribute to the lowest cost.

Algorithm 5: SUM-ExactNoIndex1 (q, D)

```

1   $n \leftarrow |q.\psi|$ ;
2  for  $i$  from 1 to  $2^n - 1$  do  $\text{Cost}[i] \leftarrow \infty$ ,  $\text{Group}[i] \leftarrow \emptyset$ ;
3  for each object  $o_i$  in  $D$  do
4      if  $o_i.\psi \cap q.\psi \neq \emptyset$  then
5           $\text{Dist}[o_i] \leftarrow \text{Dist}(o_i, q)$ ;
6          for each subset  $s_i$  in  $o_i.\psi \cap q.\psi$  do
7               $i \leftarrow \text{MapToInteger}(s_i)$ ;
8              if  $\text{Cost}[i] > \text{Dist}[o_i]$  then
9                   $\text{Cost}[i] \leftarrow \text{Dist}[o_i]$ ;
10                  $\text{Group}[i] \leftarrow \{o_i\}$ ;
11 for  $i$  from 1 to  $2^n - 1$  do
12      $\text{minValue} \leftarrow \infty$ ,  $\text{bestSplit} \leftarrow 0$ ;
13     for  $j$  from 1 to  $i/2$  do
14         if  $j \& i = j$  then
15              $S \leftarrow \text{Group}[j] \cap \text{Group}[i - j]$ ;
16              $oDist \leftarrow 0$ ;
17             for each object  $o$  in  $S$  do
18                  $oDist \leftarrow oDist + \text{Dist}[o]$ ;
19              $\text{cost} \leftarrow \text{Cost}[j] + \text{Cost}[i - j] - oDist$ ;
20             if  $\text{cost} < \text{minValue}$  then
21                  $\text{minValue} \leftarrow \text{cost}$ ;
22                  $\text{bestSplit} \leftarrow j$ ;
23     if  $\text{Cost}[i] > \text{minValue}$  then
24          $\text{Cost}[i] \leftarrow \text{minValue}$ ;
25          $\text{Group}[i] \leftarrow \text{Group}[\text{bestSplit}] \cup \text{Group}[i - \text{bestSplit}]$ ;
26 return  $\text{Cost}[2^n - 1]$  and  $\text{Group}[2^n - 1]$ 
    
```

Equations 4.4 and 4.5 can be rewritten as the following equation:

$$\begin{aligned}
 \text{Cost}[i] &= \min_{j, i \& j = j} \{ \text{Cost}[j] + \text{Cost}[i - j] - \text{oCost}(i, j) \} \\
 \text{oCost}(i, j) &= \sum_{o \in \text{Group}[i] \cap \text{Group}[i - j]} \text{Dist}(o, q) \\
 \text{opt} &= \arg \min_{j, i \& j = j} \{ \text{Cost}[j] + \text{Cost}[i - j] - \text{oCost}(i, j) \} \\
 \text{Group}[i] &= \text{Group}[\text{opt}] \cup \text{Group}[i - \text{opt}]
 \end{aligned} \tag{4.6}$$

Here, $\&$ is the bit-wise AND operator, and $i \& j = j$ states that the set represented by j is a subset of the set represented by i .

The pseudocode is outlined in Algorithm 5. We progressively compute $\text{Cost}[i]$ and $\text{Group}[i]$ from $i = 1$ to $(2^n - 1)$. When $i = (2^n - 1)$, we get the results with the lowest cost, $\text{Cost}[2^n - 1]$, and the best group, $\text{Group}[2^n - 1]$. We scan the dataset D . For each single

object that overlaps with query keyword set $q.\psi$, we check whether the object contributes to the lowest cost for a keyword subset (lines 3–10). If a keyword subset is not contained by any single object, its cost is initialized to ∞ .

The algorithm proceeds to use Equation 4.6 to compute the lowest cost for each subset (lines 11–25). We check each subset, represented by j , of the current keyword subset, represented by i , to see whether the subset represented by j and the subset represented by $(i - j)$ contribute a lower cost (lines 13–22). The lowest cost from combining two subsets is kept in $minValue$, and the integer that represents the corresponding subset is kept in $bestSplit$. If the $minValue$ contributed by aggregating two subsets is smaller than the current lowest cost of the keyword subset represented by i , we update its cost $Cost[i]$ and update the group of objects $Group[i]$ that covers it (lines 23–25). Finally, we return the lowest cost and the best group (line 26).

The algorithm scans the whole dataset D in lines 3–10 to find the nearest object for each keyword subset. Then, it finds the best group according to Equation 4.6 in lines 11–22. Given n query keywords, there are at most n objects in each group corresponding to a keyword subset, and thus checking the intersection is of complexity $O(n)$. Hence, the complexity of finding the optimal group from $objSet$ is $O(2^n \cdot 2^{n-1} \cdot n)$, which is much better than the naive method with complexity $O(2^{2n-1})$.

Example 4.3 We proceed to illustrate Algorithm 5. Given a query q with the keywords set $q.\psi = \{t_1, t_2, t_3\}$ and three objects o_1, o_2 , and o_3 with description: $o_1.\psi = \{t_1, t_2\}$ and $\text{Dist}(q, o_1) = 1$; $o_2.\psi = \{t_2, t_3\}$ and $\text{Dist}(o_2, q) = 2$; $o_3.\psi = \{t_1, t_2, t_3\}$ and $\text{Dist}(o_3, q) = 4$. Lines 3–10 return the following results:

i	1	2	3	4	5	6	7
Cost	1	1	1	2	4	2	4
Group	o_1	o_1	o_1	o_2	o_3	o_2	o_3

We subsequently utilize these values to compute the final results. For example, when computing $Cost[5]$, we determine that $\{o_1, o_2\}(\text{Cost}[1] + \text{Cost}[4])$ is better than $\{o_3\}$, and we update its value. Finally, we have the following results:

i	1	2	3	4	5	6	7
Cost	1	1	1	2	3	2	3
Group	o_1	o_1	o_1	o_2	o_1, o_2	o_2	o_1, o_2

4.2.3 An Improved Exact Algorithm Without an Index

In Equation 4.4, given a keyword set X , we need to check the combinations for each subset S of X and its corresponding complementary subset $X \setminus S$ (there are $2^{|X|-1}$ subsets of X). In addition, it is also needed to check the intersection between the group of S and the group of $X \setminus S$. In this section, we propose a method improved on the approach presented in Section 4.2.2, which can further reduce the complexity of the dynamic programming approach.

The idea is that, we can do the dynamic programming based on the objects in *objSet* instead of on the subsets of X . Specifically, given a keyword subset X , the group with the minimum cost must contain at least one object in *objSet*. Therefore, given a keyword subset X and the candidate objects set *objSet*, we can check each object in *objSet* that contain some keywords in X with its corresponding complementary keyword subset *w.r.t.* X (whose optimal cost is already known) to find the lowest cost for X , as shown in Equation 4.7.

$$\text{Cost}(X) = \min_{o \in \text{objSet} \wedge o.\psi \cap X \neq \emptyset} \{\text{Cost}(X \setminus o.\psi) + \text{Dist}(o, q)\} \quad (4.7)$$

The detail of this algorithm is depicted in Algorithm 6. We progressively compute $\text{Cost}[i]$ and $\text{Group}[i]$ from $i = 1$ to $(2^n - 1)$, and when $i = (2^n - 1)$, the lowest cost is stored in $\text{Cost}[2^n - 1]$ and the best group is stored in $\text{Group}[2^n - 1]$.

We initialize $\text{Cost}[i]$ and $\text{Group}[i]$ by scanning the dataset D (lines 4–11) as does Algorithm 5. Next, we put all the objects that contribute to the lowest cost of some keyword subset into *objSet* (lines 12–13). Next, for each keyword subset i , we enumerate each object o in *objSet* to do the dynamic programming (lines 14–24). If i is a power of 2, which means that it represents a single keyword subset, its lowest cost has already been obtained when scanning D , and thus we ignore it (line 15). We first compute the integer of the complementary keyword subset of $o.\psi$ *w.r.t.* i , *i.e.*, bit_{cp} . If it equals to i , which means that o does not contain any keywords in i , and thus it can be ignored; if it equals to 0, which means that the keyword subset represented by i is a subset of $o.\psi$, and in the step of finding the lowest cost for i by scanning D , o has been checked whether it can contribute to the lowest cost, and thus it can be ignored as well (line 19). Line 20 is used to avoid generating duplicated groups. *maxdi* denotes the maximum object identifier in a group. If it is smaller than the identifier of the current object o , we do not combine them. Instead, this group will be generated when we considering the object whose identifier is *maxdi*. The correctness of this step is due to that each keyword subset which is the subset of i already has the lowest cost value. Otherwise, we combine $\text{Group}[\text{bit}_{cp}]$ and o to see if this can generate a lower cost for the keyword subset i (lines 21–24). Finally, we return the lowest cost and the best group (line 25).

Algorithm 6: SUM-ExactNoIndex2 (q, D)

```

1  $n \leftarrow |q.\psi|$ ,  $objSet \leftarrow \emptyset$ ;
2 initialize two arrays  $Cost[]$  and  $Group[]$ ;
3 for  $i$  from 1 to  $2^n - 1$  do  $Cost[i] \leftarrow \infty$ ,  $Group[i] \leftarrow \emptyset$ ;
4 for each object  $o$  in  $D$  do
5   if  $o.\psi \cap q.\psi \neq \emptyset$  then
6      $Dist[o] \leftarrow Dist(o, q)$ ;
7     for each subset  $s$  in  $o.\psi \cap q.\psi$  do
8        $i \leftarrow MapToInteger(s)$ ;
9       if  $Cost[i] > Dist[o]$  then
10          $Cost[i] \leftarrow Dist[o]$ ;
11          $Group[i] \leftarrow \{o\}$ ;
12 for  $i$  from 1 to  $2^n - 1$  do
13   if  $Group[i] \neq \emptyset$  then  $objSet \leftarrow objSet \cup Group[i]$ ;
14 for  $i$  from 1 to  $2^n - 1$  do
15   if  $i$  is the power of 2 then continue;
16   for each object  $o$  in  $objSet$  do
17      $bit_o \leftarrow MapToInteger(o.\psi \cap q.\psi)$ ;
18      $bit_{cp} \leftarrow i - i \& bit_o$ ;
19     if  $bit_{cp} = i$  or  $bit_{cp} = 0$  then continue;
20     if  $Group[bit_{cp}].maxdi < o.di$  then continue;
21      $oDist \leftarrow Cost[bit_{cp}] + Dist(o, q)$ ;
22     if  $oDist < Cost[i]$  then
23        $Cost[i] \leftarrow oDist$ ;
24        $Group[i] \leftarrow Group[bit_{cp}] \cup o$ ;
25 return  $Cost[2^n - 1]$  and  $Group[2^n - 1]$ 

```

The algorithm also first scans the whole dataset D to find $objSet$. It finds the best group according to Equation 4.7 in lines 14–24. Given a keyword subset X , the complexity of finding the lowest cost of X is $O(|objSet|)$, and hence the complexity of finding the optimal group from $objSet$ in this algorithm is $O(2^n \cdot |objSet|)$.

Given n query keywords, $|objSet|$ is at most $2^n - 1$. However, $|objSet|$ is always much smaller than $2^n - 1$. This is because that for many keyword subsets, we cannot find an object which contains all the keywords in it. Note that o cannot appear in the optimal group of the keyword subset $X \setminus o.\psi$, and thus we avoid checking the intersection of two subsets in the SUM-ExactNoIndex1 algorithm.

Example 4.4 Recall Example 4.3. After scanning the dataset, we can get the same table as in Example 4.3 for each keyword subset, and we can obtain $objSet = \{o_1, o_2, o_3\}$.

However, instead of enumerating the subsets to get the lowest cost, this algorithm enumerates the objects in $objSet$. Taking the $i = 7$ as an example, there are three objects

i	1	2	3	4	5	6	7
Cost	1	1	1	2	4	2	4
Group	o_1	o_1	o_1	o_2	o_3	o_2	o_3

in $objSet$, i.e., $\{o_1, o_2, o_3\}$, and we enumerate each of them to find the result for it. Initially its cost is 4 contributed by o_3 .

1) First, we consider o_1 (whose integer value is 3). The integer of its complementary keyword subset is 4, whose cost is 2 contributed by o_2 . Therefore, we can obtain a lower cost $\text{Dist}(o_1, q) + \text{Cost}[4] = 1 + 2 = 3$ for $i = 7$.

2) Second, we consider o_2 (whose integer value is 6). The integer of its complementary keyword subset is 1, whose cost is 1 contributed by o_1 . However, since o_1 has smaller identifier than does o_2 , we do not combine them. This group is already generated when we consider o_1 .

3) Third, o_3 covers all three query keywords, and thus we ignore it since it is already processed when we scan the dataset.

4.2.4 Exact Algorithm Using an Index

The dynamic programming algorithm presented above needs to scan the whole dataset. This has two drawbacks: (1) it wastes computation when checking many unnecessary objects that do not contain any query keyword, and (2) all the objects whose text descriptions overlap with the query keywords are scanned to obtain the lowest costs for the query keyword subsets.

To overcome the first drawback, we utilize the IR-tree that enables us to retrieve only the objects that contain some query keywords while avoiding checking the objects containing no query keywords. For the second drawback, we show that it is not always necessary to scan all the objects covering part of the query keywords.

We propose the following principle for our algorithm: we process objects in ascending order of their distances to a query q . By following that order, we know that the lowest cost of a subset is always contributed by a single or a group of closer objects based on Lemma 4.8.

Lemma 4.10 *Consider a query q . If we process objects in ascending order of their distances to q , when we reach an object o_i containing a query keyword subset ws , all subsets of ws will get their lowest costs.*

Proof: Obvious since all objects to be visited after o_i have larger cost for any subset of ws ; thus, its lowest cost is either contributed by o_i or by objects visited earlier.

Example 4.5 Recall the query in Example 4.2. We first process object o_1 , and we know 1 is the lowest costs of subsets $\{t_1, t_2\}$, $\{t_1\}$, and $\{t_2\}$. Then we reach o_2 , and we know 2 is the lowest costs of subsets $\{t_2, t_3\}$ and $\{t_3\}$ ($\{t_2\}$ already has lowest cost 1).

Based on Lemma 4.10, we can derive a stopping condition for our algorithm—we reach an object that contains all the query keywords. However, if no such object exists in the dataset, the algorithm is still required to scan to the furthest object containing some query keywords before it can stop. In the example in Table 4.1, we need to read all the four objects. But if the third furthest object o_3 covers $\{t_1, t_2, t_3\}$, we need not read o_4 .

We proceed to present an additional stopping condition.

Lemma 4.11 *Given two query keyword subsets ws_i and ws_j , and with union $ws_u = ws_i \cup ws_j$, we have $\text{Cost}(ws_u) \leq \text{Cost}(ws_i) + \text{Cost}(ws_j)$.*

Proof: *Obvious from Equations 4.3–4.5.*

Based on Lemma 4.11, for any two keyword subsets whose lowest costs are known, we can obtain an upper bound of the lowest cost value for the keyword subset that is the union of the two keyword subsets. We denote the upper bound by $Cost_u$.

In our algorithm, we keep track of the upper bounds for the subsets whose costs are still unknown. Whenever we reach an object from which some keyword subset gets its lowest cost (according to Lemma 4.10), the subset, together with each of the subsets that have either lowest costs or upper bounds of cost (*i.e.*, the keyword subsets that are covered by visited objects), are used to update the upper bound cost values of the corresponding union keyword subsets.

Example 4.6 *Recall Example 4.2. After the object o_2 is scanned, $\{t_3\}$ gets its lowest cost 2. We can compute an upper bound for $\{t_1, t_3\}$ using the costs of $\{t_1\}$ and $\{t_3\}$, *i.e.*, $Cost_u(\{t_1, t_3\}) = \text{Cost}(\{t_1\}) + \text{Cost}(\{t_3\}) = 3$ (covered by o_1 and o_2). Similarly, we can also compute an upper bound value 3 for $\{t_1, t_2, t_3\}$ using the costs of $\{t_1, t_2\}$ and $\{t_3\}$, and the set is also covered by o_1 and o_2 .*

When we reach o_3 , we get a lower cost of 2.5 for $\{t_1, t_3\}$ (the previous upper bound of 3 is updated). Then $\{t_1, t_3\}$ are combined with $\{t_2\}$ to form $\{t_1, t_2, t_3\}$ with a cost of 3.5. Since this value exceeds its current upper bound, no update is needed.

We are ready to introduce a lemma that provides an early stopping condition for our algorithm.

Lemma 4.12 *Suppose that we scan objects in ascending order of their distances to q . Given a keyword subset ws , when we reach object o_i , and if $\text{Dist}(o_i, q) \geq Cost_u(ws)$, then $\text{Cost}(ws) = Cost_u(ws)$, where $Cost_u(ws)$ is the current upper bound of ws .*

Proof: *We prove this by contradiction. If any object o_j further to q than o_i is a member of the best group then it must have*

$$\text{Cost}(ws) \geq \text{Dist}(o_j, q) \geq \text{Dist}(o_i, q) \geq Cost_u(ws)$$

Since $Cost_u(ws)$ cannot be smaller than $Cost(ws)$, no further object will be contained in the best group. In addition, $Cost_u(ws)$ is the current minimum cost value, and thus it becomes the lowest cost of ws .

Example 4.7 Recall again Example 4.2. By following the distances in ascending order, when the algorithm reaches o_4 ($Dist(q, o_4) = 4$), we can conclude that $Cost_u(\{t_1, t_2, t_3\}) = 3$ is the lowest cost and that the best group is (o_1, o_2) .

The pseudocode is described in Algorithm 7. All the keyword subsets whose lowest costs are already known are stored in the variable *markedSet*, and the subsets that have upper bounds are stored in the variable *valuedSet*. The IR-tree is used for retrieving the next nearest object that covers some query keywords. We use a min-priority queue U to store the IR-tree nodes and objects, where their distances to the query are the keys.

The priority queue U is initialized to the root node of the IR-tree (line 3). We dequeue an element e from U , and we compute the keyword intersection ks between e and q (lines 5–6). If the keyword subset ks is contained in *markedSet* (whose lowest costs are known), we do not need to process e according to Lemma 4.8 (line 7). Otherwise, we process e according to its type: 1) If e is an index node, we check each of its child nodes, denoted by e' , to see whether e' contains a keyword subset of q that is not contained in *markedSet*. If so, e' is inserted into U with its minimum distance to query q as its priority key (lines 8–12). 2) If e is an object, we first utilize its distance to q to move some keyword subsets from *valuedSet* to *markedSet*. The subsets whose upper bounds are smaller than $Dist(e, q)$ get their lowest costs (lines 14–20) according to Lemma 4.12. If the query keyword set $q.\psi$ is confirmed to get its lowest cost, the algorithm terminates (line 18). Then for each subset ss of $q.\psi \cap e.\psi$, if its lowest cost is unknown (line 22), the object e constitutes the best group (Lemma 4.10) for ss . Since ss may already be covered by previously visited objects and have an upper bound of its lowest cost, we remove ss from *valuedSet* (lines 25). Once $q.\psi$ gets its lowest cost, the algorithm terminates (lines 28–30). In lines 31–40, we combine the object e with the subsets that already have cost values (Lemma 4.11). In line 33, “|” is the bit-wise OR operator. If one is the subset of the other (line 34), we do not combine the two subsets; otherwise, we update the cost value for the union keyword subset (lines 35–40).

This algorithm runs faster than Algorithm 6 due to two reasons: first, using the IR-tree avoids scanning the whole dataset; second, based on Lemma 4.12, we are able to find the best group without scanning objects whose distances exceed the cost of the current best group.

Example 4.8 Recall Table 4.1 in Example 4.2. The algorithm works as follows.

1) After processing o_1 , the result is shown in Table 4.2, in which i is the integer representing a keyword subset and status “M” means that the subset is contained in

Algorithm 7: SUM-ExactWIndex($q, irTree$)

```

1   $markedSet \leftarrow \emptyset, valuedSet \leftarrow \emptyset, n \leftarrow |q.\psi|;$ 
2  for  $i$  from 1 to  $2^n - 1$  do  $Cost[i] \leftarrow \infty, Group[i] \leftarrow \emptyset;$ 
3   $U \leftarrow$  new min-priority queue;  $U.Enqueue(irTree.root, 0);$ 
4  while  $U$  is not empty do
5       $e \leftarrow U.Dequeue();$ 
6       $ks \leftarrow q.\psi \cap e.\psi;$ 
7      if  $ks \in markedSet$  then continue;
8      if  $e$  is a node then
9          foreach entry  $e'$  in node  $e$  do
10             if  $e$  is a non-leaf node then  $dist \leftarrow \minDist(e', q);$ 
11             else  $dist \leftarrow Dist(e', q);$ 
12             if  $q.\psi \cap e'.\psi \neq \emptyset$  and  $q.\psi \cap e'.\psi \notin markedSet$  then  $U.Enqueue(e', dist);$ 
13     else //  $e$  is an object
14         foreach set  $S \in valuedSet$  do
15              $i \leftarrow MapToInteger(S);$ 
16             if  $Cost[i] < Dist(q, e)$  then
17                 if  $i = 2^n - 1$  then // Lemma 4.12
18                     returnCost  $[2^n - 1]$  and  $Group[2^n - 1];$ 
19                      $valuedSet \leftarrow valuedSet \setminus S;$ 
20                      $markedSet \leftarrow markedSet \cup S;$ 
21             foreach subset  $ss \subseteq ks$  do
22                 if  $ss \notin markedSet$  then
23                      $i \leftarrow MapToInteger(ss);$ 
24                      $markedSet \leftarrow markedSet \cup ss;$ 
25                     if  $ss \in valuedSet$  then  $valuedSet \leftarrow valuedSet \setminus ss;$ 
26                      $Cost[i] \leftarrow Dist(e, q);$ 
27                      $Group[i] \leftarrow \{e\};$ 
28          $j \leftarrow MapToInteger(ks);$ 
29         if  $j = 2^n - 1$  then // Lemma 4.10
30             returnCost  $[2^n - 1]$  and  $Group[2^n - 1];$ 
31         for  $i$  from 1 to  $2^n - 1$  do
32             if  $Cost[i] = \infty$  then continue;
33              $unionKey \leftarrow i | j;$ 
34             if  $unionKey = i$  or  $unionKey = j$  then continue;
35             if  $Cost[unionKey]$  is  $\infty$  then
36                  $valuedSet \leftarrow valuedSet \cup MapToSet(unionKey);$ 
37              $D \leftarrow Cost[i] + Dist(e, q);$ 
38             if  $Cost[unionKey] > D$  then
39                  $Cost[unionKey] \leftarrow D;$ 
40                  $Group[unionKey] \leftarrow Group[i] \cup \{e\};$ 
41 returnCost  $[2^n - 1]$  and  $Group[2^n - 1];$ 

```

i	1	2	3	4	5	6	7
Cost	1	1	1	∞	∞	∞	∞
Group	o_1	o_1	o_1	null	null	null	null
Status	M	M	M	null	null	null	null

 Table 4.2: Results after processing o_1

markedSet. Table 4.2 shows that $\{t_1\}$ ($i = 1$), $\{t_2\}$ ($i = 2$), and $\{t_1, t_2\}$ ($i = 3$) get their lowest costs and best groups.

2) After processing o_2 , the result is shown in Table 4.3. Except for $\{t_1, t_3\}$ and $\{t_1, t_2, t_3\}$, all the subsets obtain their lowest costs. The cost values of the two subsets are obtained by combining other subsets with known lowest cost. The status value “V” means that the subset is stored in *valuedSet*.

i	1	2	3	4	5	6	7
Cost	1	1	1	2	3	2	3
Group	o_1	o_1	o_1	o_2	o_1, o_2	o_2	o_1, o_2
Status	M	M	M	M	V	M	V

 Table 4.3: Results after processing o_2

3) After processing o_3 , we have the result shown in Table 4.4. Here, $\{t_1, t_3\}$ gets its lowest cost since it is covered by o_3 .

i	1	2	3	4	5	6	7
Cost	1	1	1	2	2.5	2	3
Group	o_1	o_1	o_1	o_2	o_3	o_2	o_1, o_2
Status	M	M	M	M	M	M	V

 Table 4.4: Results after processing o_3

4) When we reach o_4 , since its distance to the query is already larger than the currently lowest cost of $\{t_1, t_2, t_3\}$ (the only element in *valuedSet*), we do not need to process it. Set $\{t_1, t_2, t_3\}$ gets the lowest cost value 3 and is moved to *markedSet*. We now find the best group and the lowest cost.

Assume the number of query keywords is n , and the number of relevant objects is m . We call the operation of searching for the next nearest object that contains a specified keyword the “keyword-aware next nearest neighbor search,” and we assume the cost of this operation using the IR-tree is $O(F)$. The algorithm can be viewed as contiguously performing the keyword-aware next nearest neighbor search (with complexity $O(F)$), at most m times. Once an object is retrieved, it is combined with each query keyword subset that already has a cost to update the costs of other keyword subsets. Thus, the worst case complexity of this algorithm is $O(m(F + 2^n))$.

4.3 Processing MAX+MAX SGK Queries

We present two approximation algorithms in Sections 4.3.1 and 4.3.2, and two exact algorithms in Section 4.3.3 and 4.3.4.

4.3.1 Approximation Algorithm 1

Given a query q , the idea of the algorithm, called **MAXMAX-Appro1**, is to find the nearest object for each keyword t_i in $q.\psi$. The set of all such nearest objects make up the result set. The pseudocode, which assumes that the dataset is indexed using the IR-tree, is outlined in Algorithm 8. The algorithm uses a min-priority queue U for the best-first search. In each iteration, we dequeue an element e from U . If e is an object, we push it into the result set and update the uncovered keyword subset (lines 7–10); if e is a node in the IR-tree, we insert all its child nodes that contain some uncovered keywords into U (lines 12–17). The runtime of this algorithm is linear in the number of query keywords.

Algorithm 8: MAXMAX-Appro1($q, irTree$)

```

1  $U \leftarrow$  new min-priority queue;
2  $U.Enqueue(irTree.root, 0)$ ;
3  $Group \leftarrow \emptyset$ ;  $Cost \leftarrow 0$ ;
4  $uSkiSet \leftarrow q.\psi$ ; // uncovered keywords
5 while  $U$  is not empty do
6    $e \leftarrow U.Dequeue()$ ;
7   if  $e$  is an object then
8      $Group \leftarrow Group \cup e$ ;  $Cost \leftarrow Cost + Dist(e, q)$ ; // add e to result
9      $uSkiSet \leftarrow uSkiSet \setminus e.\psi$ ;
10    if  $uSkiSet = \emptyset$  then break;
11  else
12    read the posting lists of  $e$  for keywords in  $uSkiSet$ ;
13    foreach entry  $e'$  in node  $e$  do
14      if  $uSkiSet \cap e'.\psi \neq \emptyset$  then
15        if  $e$  is a non-leaf node then
16           $U.Enqueue(e', minDist(e', q))$ ;
17        else  $U.Enqueue(o, Dist(o, q))$ ;
18 return  $Group$  and  $Cost$ ; // results

```

Example 4.9 Consider a query $q.\psi = \{t_1, t_3, t_5\}$ and the objects shown in Figure 2.6. The object o_1 covering t_1 is first added to the result. Then o_2 containing t_3 is added, and when o_4 containing t_5 is retrieved, we obtain a group. Object o_4 has the maximum distance to query, which is 3.2. The maximum diameter is 6, which is the distance between o_2 and o_4 . Hence, the cost of this group is 9.2.

We proceed to show that MAXMAX-Appr1 is within an approximation factor of 3. We denote the group returned by MAXMAX-Appr1 as APP_1 .

Theorem 4.9 *The cost of APP_1 for a given query q , is at most three times the cost of the optimal solution OPT : $\text{Cost}(q, APP_1) \leq 3 \cdot \text{Cost}(q, OPT)$.*

Proof: Let o_f denote the furthest object to q in APP_1 , and let $d = \text{Dist}(o_f, q)$. Obviously, the optimal solution OPT satisfies $\text{Cost}(q, OPT) \geq d$.

In the group APP_1 , the largest possible distance between two objects in APP_1 is $2d$. Thus, we have the following cost: $\text{Cost}(q, APP_1) \leq d + 2d \leq 3 \cdot \text{Cost}(q, OPT)$.

We assume that using IR-tree the cost of finding the next nearest neighbor containing a specified keyword is $O(F)$. This algorithm requires at most n such operations, and thus the complexity is $O(nF)$.

4.3.2 Approximation Algorithm 2

Based on MAXMAX-Appr1, we present an algorithm with a better approximation bound. The idea of this algorithm can be described as follows. We first invoke the algorithm MAXMAX-Appr1 to find a group of objects APP_1 . Let t_{inf} be the most infrequent keywords in $q.\psi$. Next, for each object o_i containing t_{inf} , we create a new query q_{o_i} using the position of o_i and the keywords of the original query q , i.e., $q_{o_i}.\lambda = o_i.\lambda$ and $q_{o_i}.\psi = q.\psi \setminus o_i.\psi$. We then invoke MAXMAX-Appr1 to find a group of objects for q_{o_i} , denoted by G_{o_i} , and we compute the cost of this group w.r.t. q . After each object containing t_{inf} is processed, we find the group with the smallest cost, and compare it with APP_1 , and finally we return the one with smaller cost as the result.

This algorithm only focuses on the objects containing the most infrequent query keyword. However, the number of such objects may still be large. We show that it is not necessary to process each object containing t_{inf} according to the following lemma.

Lemma 4.13 *Given a MAX+MAX SGK query q and the current best cost $curCost$, any object whose distance to q is larger than $curCost$ cannot be contained in the optimal group of q .*

Proof: If an object o with distance to q larger than $curCost$ is contained in a group G , we have $\text{Cost}(q, G) \geq \max_{r_1, r_2 \in G} \text{Dist}(r_1, r_2) \geq \text{Dist}(o, q) > curCost$, and thus G cannot be the optimal group.

Based on Lemma 4.13, we can process the objects containing t_{inf} in ascending order of their distances to the query q . When we obtain a group, we update the current best cost if it is larger than the cost of the new found group. When we reach an object whose distance is even larger than the current best cost, we can stop and return.

Algorithm 9: MAXMAX-Appro2($q, irTree$)

```

1  $U \leftarrow$  new min-priority queue;
2  $U.Enqueue(irTree.root, 0)$ ;
3  $(curGroup, curCost) \leftarrow$  MAXMAX-Appro1( $q, irTree$ );
4  $t_{inf} \leftarrow$  the most infrequent keyword in  $q.\psi$ ;
5 while  $U$  is not empty do
6    $e \leftarrow U.Dequeue()$ ;
7   if  $e$  is not an object then
8     if  $\text{minDist}(e, q) \geq curCost$  then break; // Lemma 4.13
9     foreach entry  $e'$  in node  $e$  do
10      if  $t_{inf} \in e'.\psi$  then
11        if  $e$  is a leaf node then
12           $U.Enqueue(e', \text{Dist}(e', q))$ ;
13        else  $U.Enqueue(e', \text{minDist}(e', q))$ ;
14   else
15     if  $\text{Dist}(e, q) \geq curCost$  then break; // Lemma 4.13
16      $q_e.\lambda \leftarrow e.\lambda$ ;  $q_e.\psi \leftarrow q.\psi \setminus e.\psi$ ;
17      $(group, cost) \leftarrow$  MAXMAX-Appro1( $q_e, irTree$ );
18     if  $cost < curCost$  then
19        $curCost \leftarrow cost$ ;
20        $curGroup \leftarrow group$ ;
21 return  $curGroup$  and  $curCost$ ;

```

The pseudocode is given in Algorithm 9. We first find a group using MAXMAX-Appro1, and it serves as the current best group (line 3). Then we find the word t_{inf} that is the most infrequent query keyword (line 4). In lines 5–20, we incrementally search for the next nearest objects containing t_{inf} within the range of $curCost$. We dequeue an element e from U in each step. If it is an IR-tree node, we check if its minimum distance exceeds $curCost$ (line 9). If so, the algorithm terminates according to Lemma 4.13. Otherwise, we read all its child nodes and insert the nodes that contain t_{inf} into U according to their minimum distances to q (lines 9–13). If e is an object, we also compare its distance to q with $curCost$ to determine whether the algorithm terminates (line 15). We create a new query q_e with the position of e and the texts of q , and we then find a group using q_e as the query and compute its cost (lines 16–17). If this new cost is smaller than $curCost$, we update $curCost$ and the current best group $curGroup$ (lines 18–20). Finally, we return $curGroup$ and $curCost$.

Example 4.10 Recall query q and the dataset in Example 4.9. Algorithm MAXMAX-Appro1 is first invoked to return a group $\{o_1, o_2, o_4\}$ with cost 9.2. In the query, t_3 is the most infrequent keywords, and it is only contained in o_2 and o_3 . We search for a group near o_2 , that is $\{o_2, o_7, o_8\}$ with cost 10.2 ($\text{Dist}(o_8, q) + \text{Dist}(o_2, o_7) = 8 + 2.2$), which is

worse than that of the previous one. Then, we search for a group near o_3 and we find $\{o_3, o_4\}$ with cost 8.2 ($\text{Dist}(o_4, q) + \text{Dist}(o_3, o_4) = 3.2 + 5$). Therefore, $\{o_3, o_4\}$ becomes the current best group, and we return it as the result.

We proceed to study the approximation ratio of the algorithm. We denote the optimal group for a given query q as OPT , and denote the group returned by MAXMAX-Appro2 as APP_2 .

Lemma 4.14 *Given a query q and an object o_i containing t_{inf} , the cost of the group found at the position of o_i , i.e., $\text{Cost}(q, G_{o_i})$, is in the following range: $\text{Dist}(o_i, q) + \text{Dist}(o_i, o_{max}) \leq \text{Cost}(q, G_{o_i}) \leq \text{Dist}(o_i, q) + 3\text{Dist}(o_i, o_{max})$, where o_{max} is the furthest object to o_i in G_{o_i} .*

Proof: 1) $\text{Dist}(o_i, q)$ is a lower bound on the distance of this group to q , and $\text{Dist}(o_i, o_{max})$ is a lower bound on the diameter of this group. As a result, the minimum cost of G_{o_i} is $\text{Dist}(o_i, q) + \text{Dist}(o_i, o_{max})$.

2) $\text{Dist}(o_i, q) + \text{Dist}(o_i, o_{max})$ is the maximum possible distance to q of G_{o_i} . Further, the diameter will not exceed $2\text{Dist}(o_i, o_{max})$ since every object of G_{o_i} is in the circle with center o_i and with radius $\text{Dist}(o_i, o_{max})$. Therefore, the cost of this group is upper bounded by $\text{Dist}(o_i, q) + 3\text{Dist}(o_i, o_{max})$.

Theorem 4.10 *The approximation ratio of algorithm MAXMAX-Appro2 is not larger than 1.8, i.e., $\text{Cost}(q, APP_2) \leq 1.8 \cdot \text{Cost}(q, OPT)$.*

Proof: Let o_f denote the furthest object to q in APP_1 , let o_j denote the object containing the word t_{inf} in the optimal group OPT , and let o_{max} be the object that is the furthest to o_j in G_{o_j} .

1) Consider the case when $\text{Dist}(o_j, q) \geq \text{Dist}(o_f, q)$.

Object o_{max} must contain some keyword, denoted by t , which is not covered by the other objects in G_{o_j} . Among objects containing t , o_{max} is the closest to o_j . Therefore, in OPT , the object covering t cannot be closer to o_j than o_{max} . As a result, $\text{Dist}(o_j, o_{max})$ is a lower bound on the diameter of OPT . $\text{Dist}(o_j, q)$ is a lower bound on the distance between q and OPT . Hence, we get $\text{Cost}(q, OPT) \geq \text{Dist}(o_j, q) + \text{Dist}(o_j, o_{max})$.

Because APP_2 is either the group with the smallest cost among groups found on each object containing t_{inf} or APP_1 , we know that $\text{Cost}(q, APP_2) \leq \text{Cost}(q, G_{o_j})$ and $\text{Cost}(q, APP_2) \leq \text{Cost}(q, APP_1)$. According to Lemma 4.14, we get $\text{Cost}(q, APP_2) \leq \text{Cost}(q, G_{o_j}) \leq \text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max})$, and according to Theorem 4.9, we get $\text{Cost}(q, APP_2) \leq 3\text{Dist}(o_f, q) \leq 3\text{Dist}(o_j, q)$. Thus, it holds true that:

$$\frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{\text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max})}{\text{Dist}(o_j, q) + \text{Dist}(o_j, o_{max})}, \frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{3\text{Dist}(o_j, q)}{\text{Dist}(o_j, q) + \text{Dist}(o_j, o_{max})}$$

If $\text{Dist}(o_j, o_{max}) \leq \frac{2}{3}\text{Dist}(o_j, q)$, then $\text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max}) \leq 3\text{Dist}(o_j, q)$. We have $\frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{\text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max})}{\text{Dist}(o_j, q) + \text{Dist}(o_j, o_{max})} \leq 1.8$. Otherwise, $\text{Dist}(o_j, o_{max}) > \frac{2}{3}\text{Dist}(o_j, q)$, then $\text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max}) > 3\text{Dist}(o_j, q)$, and we have $\frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{3\text{Dist}(o_j, q)}{\text{Dist}(o_j, q) + \text{Dist}(o_j, o_{max})} \leq 1.8$.

2) Now consider the case when $\text{Dist}(o_j, q) < \text{Dist}(o_f, q)$.

o_f must contain some keyword t that is not covered by any other objects in APP_1 . Since o_f is the closest object to q containing t , the object containing t in OPT must be further to q than o_f , and thus $\text{Dist}(o_f, q)$ is the lower bound of the maximum distance of an object in OPT to q . $\text{Dist}(o_j, o_{max})$ is the lower bound of the maximum distance between any pair. Therefore, we can get $\text{Cost}(q, OPT) \geq \text{Dist}(o_f, q) + \text{Dist}(o_j, o_{max})$.

Because $\text{Cost}(q, APP_2) \leq \text{Cost}(q, G_{o_j})$ and $\text{Cost}(q, APP_2) \leq \text{Cost}(q, APP_1)$, we get $\text{Cost}(q, APP_2) \leq \text{Cost}(q, G_{o_j}) \leq \text{Dist}(o_j, q) + 3\text{Dist}(o_j, o_{max}) < \text{Dist}(o_f, q) + 3\text{Dist}(o_j, o_{max})$ and $\text{Cost}(q, APP_2) \leq 3\text{Dist}(o_f, q)$. Thus, it holds true that:

$$\frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{\text{Dist}(o_f, q) + 3\text{Dist}(o_j, o_{max})}{\text{Dist}(o_f, q) + \text{Dist}(o_j, o_{max})}, \quad \frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)} \leq \frac{3\text{Dist}(o_f, q)}{\text{Dist}(o_f, q) + \text{Dist}(o_j, o_{max})}$$

It can be proved that $\frac{\text{Cost}(q, APP_2)}{\text{Cost}(q, OPT)}$ is no larger than 1.8 as well. Thus, we complete the proof.

Assume that there are n query keywords, the number of objects that contain the most infrequent keyword is N_{inf} . Since we only invoke **MAXMAX-Appro1** around objects containing t_{inf} , we need at most $O(N_{inf})$ IR-tree keyword-aware next nearest neighbor search operations (each has complexity $O(F)$), and the cost is $O(N_{inf}F)$. On each object containing t_{inf} , we invoke **MAXMAX-Appro1** whose complexity is $O(nF)$, and thus the final complexity is $O(N_{inf}(F + nF)) = O(nFN_{inf})$.

4.3.3 Exact Algorithm 1

It is challenging to develop an exact algorithm for the **MAX+MAX** spatial group keyword queries, as it appears that an exact algorithm cannot avoid an exhaustive search of the object space.

We utilize the **MAXMAX – Appro2** algorithm to first derive an upper bound cost for the best group and then use this cost to bound the exhaustive search in the object space. Specifically, we develop several pruning strategies to prune the enumeration space. With these efforts, we expect the exact algorithm to be reasonably efficient when the dataset contains at most tens of thousands of objects and the number of query keywords is small.

Before presenting the idea underlying the algorithm, we define the concept of *covering node set* and the lower bound cost of such a set.

Definition 4.3 Covering Node Set. *Given a query q , a covering node set is a set of nodes that cover the query keywords, with each node contributing at least one object to the final result.*

Lemma 4.15 *Given a query q and a covering node set N ,*

- 1) *if N contains only one node E , its lower bound cost is: $\minCost(N) = \minDist(q, E)$;*
- 2) *if N contains multiple nodes, its lower bound cost is:*

$$\minCost(N) = \max_{E_i \in N} \minDist(q, E_i) + \max_{E_j, E_k \in N} \minDist(E_j, E_k),$$

where $\max_{E_i \in N} \minDist(q, E_i)$ is the minimum distance from q to a group from N , and $\max_{E_j, E_k \in N} \minDist(E_j, E_k)$ is the minimum diameter of a group from N .

Proof: If the group χ is only contributed by E , $\max_{o_i \in \chi} (\text{Dist}(o_i, q)) \geq \minDist(q, E)$, and thus $\text{Cost}(q, \chi) \geq \minDist(q, E)$. If χ is contributed by multiple nodes in N , $\max_{o_i \in \chi} (\text{Dist}(o_i, q)) \geq \max_{E_i \in N} \minDist(q, E_i)$, and we also know $\max_{o_j, o_k \in \chi} (\text{Dist}(o_j, o_k)) \geq \max_{E_j, E_k \in N} \minDist(E_j, E_k)$. Hence, $\text{Cost}(q, \chi) \geq \minCost(N)$.

Thus, $\minCost(N)$ is the lower bound of the cost of the best group from N .

The algorithm's idea is to perform a best-first search on the IR-tree to find the covering node sets, with some objects from these nodes constituting a group satisfying the keywords requirement of a query. We process the covering node set with the lowest cost to find covering node sets from their child nodes. When we reach a covering node set consisting of leaf nodes, we find a group of objects with the lowest cost by performing an exhaustive search.

The pseudocode is given in Algorithm 10. A priority queue U stores the covering node sets. Algorithm 8 (MAXMAX-Appro1) is invoked to find a group, and its cost serves as the current lowest cost (lines 3–4). We next search from the root node, enumerating all its child node sets to find covering node sets. If a node set covers the query keywords, we estimate its lower bound cost by Lemma 4.15 and insert it into U with the estimated cost as the key. After we finish enumerating the covering node sets of a covering node set, we dequeue a covering node set N from U , and we find its lower level covering node sets that cover the query keywords. The covering node sets whose lower bounds are smaller than the current lowest cost are inserted into U (lines 14–17). Once we reach a leaf node, we do an exhaustive search to get the best group in the covering node set, and we update the lowest cost stored in $Cost_V$ with the cost of this group (lines 10–12).

The algorithm terminates when the lower bound of the covering node set at the top position of U is larger than the current lowest cost (line 7) because the remaining covering node sets in U have larger costs than the current lowest cost, and because covering node sets at their lower levels do not contain better groups according to Lemma 4.16.

Algorithm 10: MAXMAX-Exact1 ($q, irTree$)

```

1  $U \leftarrow$  new min-priority queue;
2  $U.Enqueue(irTree.root, 0)$ ;
3  $V \leftarrow Type2App2(q, irTree)$ ;
4  $Cost_V \leftarrow$  the cost of  $V$ ;
5 while  $U$  is not empty do
6    $N \leftarrow U.Dequeue()$ ;
7   if  $minCost(N) \geq Cost_V$  then break;
8   if  $N$  contains leaf nodes then
9      $V' \leftarrow ExhaustiveSearch(N)$ ;
10     $Cost_{V'} \leftarrow$  the cost of  $V'$ ;
11    if  $Cost_{V'} < Cost_V$  then
12       $Cost_V \leftarrow Cost_{V'}$ ;  $V \leftarrow V'$ ;
13  else
14     $S \leftarrow EnumerateNodeSets(N, Cost_V, q)$ ;
15    foreach node set  $ns$  in  $S$  do
16      if  $q.\psi \subseteq ns.\psi$  then
17         $U.Enqueue(ns, minCost(ns))$ ;
18 return  $V$  and  $Cost_V$ 

```

Lemma 4.16 For any lower level covering node set L enumerated from covering node set N , we have $minCost(L) \geq minCost(N)$.

Proof: a) Denote by l_a the child node of the node n_i that is the furthest from Q in N . We have: $minDist(q, l_a) \geq minDist(q, n_i)$.

b) Denote by l_b and l_c the child nodes of n_j and n_k that have the largest distance among all pairs of child nodes from n_j and n_k . We then have: $minDist(l_b, l_c) \geq minDist(n_j, n_k)$. In addition, as l_a, l_b , and $l_c \in L$, it is true that $max_{l_i \in L} minDist(q, l_i) \geq minDist(q, l_a)$, and $max_{l_j, l_k \in L} minDist(l_j, l_k) \geq minDist(l_b, l_c)$. We then obtain $minCost(L) \geq minDist(q, l_a) + minDist(l_b, l_c) \geq minCost(N)$.

We proceed to describe `EnumerateNodeSets` called in line 14, which enumerates all possible lower level covering node sets from an upper level covering node set. First, we consider the simple case where the covering node set N only contains one node. We use a bottom-up method to find all the covering node sets. Let $n = |q.\psi|$ be the number of keywords in query q . The size of a covering node set is at most n based on the pigeon-hole principle. We first enumerate the covering node sets that consist of a single node. Then these covering node sets are combined to form covering node sets with two members. In general, all the covering node sets of size m can be combined by two covering node sets of size $(m - 1)$. If two covering node sets with size $(m - 1)$ share the first $(m - 2)$ nodes and the lower bound cost of the new combined node set is smaller than that of the current lowest cost, this new set is a candidate covering node set.

Algorithm 11: EnumerateNodeSets ($N, Cost, q$)

```

1  $setList \leftarrow \emptyset$ ;
2 foreach node  $n_i$  in  $N$  do
3    $cList_i \leftarrow \emptyset$ ;
4    $L_i^1 \leftarrow \emptyset$ ;
5   foreach child node  $c_i$  of  $n_i$  do
6     if  $minCost(c_i) \geq Cost$  then  $L_i^1 \leftarrow L_i^1 \cup c_i$ ;
7   for  $m$  from 2 to  $(|q.\psi| - |N| + 1)$  do
8      $L_i^m \leftarrow \emptyset$ ;
9     foreach node set  $NS_1 \in L_i^{m-1}$  do
10      foreach node set  $NS_2 \in L_i^{m-1}$  do
11        if  $NS_1$  and  $NS_2$  share the first  $(m-1)$  nodes then
12           $NS \leftarrow Merge(NS_1, NS_2)$ ;
13          if  $minCost(NS) < Cost$  then
14             $L_i^m \leftarrow L_i^m \cup NS$ ;
15       $cList_i \leftarrow cList_i \cup L_i^m$ ;
16 foreach node set  $ns$  formed by node sets selected from each of  $cList_1 \dots cList_n$  do
17   if  $minCost(ns) < Cost$  then
18      $setList \leftarrow setList \cup ns$ ;
19 return  $setList$ ;

```

Next, we move to the case where a covering node set N contains more than one node. For each node, we follow the previous method to get a list of candidate node sets. However, we do not need to enumerate all combinations. Rather, we only need to enumerate its child nodes up to size $(|q.\psi| - |N| + 1)$ according to the pigeon-hole principle (each node in N must contribute at least one child node). After we get a list of child node sets from each node, we select a node set from each list and merge them to determine whether they are a covering node set and to learn whether the merged node set has a lower bound cost that is smaller than the current lowest cost. The details of EnumerateNodeSets are covered in Algorithm 11.

Assume that there are n query keywords, and the number of relevant objects is m . This algorithm perform the exhaustive search first on the IR-tree node space and then on the object space, and thus the worst case time complexity is $O(m^n)$.

4.3.4 Exact Algorithm 2

MAXMAX-Exact1 is a best-first search method based on the feasible set space. Though equipped with several pruning techniques, it is prohibitively expensive when the number of query keywords increases. We extend the idea of the MAXMAX-Appro2 algorithm to devise a better exact algorithm.

Because the optimal group must contain an object containing the most infrequent query keyword t_{inf} , we can do an exhaustive search around each object containing t_{inf} to find the best group containing this object. We repeat this until all objects containing t_{inf} are processed, and the group with the smallest cost must be the optimal answer to the query.

We first utilize MAXMAX-Appro2 to derive an upper bound cost for the optimal group. It is initially used to bound the exhaustive search space on an object containing t_{inf} . We also develop several pruning strategies to reduce the enumeration of groups during the exhaustive search around an object. The upper bound cost keeps decreasing as more groups are enumerated, and it is used to prune the enumeration in further search. With these efforts, we expect the exact algorithm to be reasonably efficient when the dataset contains at most tens of thousands of objects and the number of query keywords is small.

I. Bound the exhaustive space around an object containing t_{inf} .

Since enumerating the groups runs exponentially with the number of objects in the search space, how to reduce the exhaustive search space around an object containing t_{inf} is crucial. We proceed to introduce how we can do this with the help of the current best cost $curCost$.

Lemma 4.17 *Given a MAX+MAX SGK query q and two objects o_i and o_j , the lower bound value of the cost of a group G containing o_i and o_j can be computed by $\max\{\text{Dist}(o_i, q), \text{Dist}(o_j, q)\} + \text{Dist}(o_i, o_j)$. We denote this value as $LOW_q(o_i, o_j)$.*

Proof: It is obvious that $\max\{\text{Dist}(o_i, q), \text{Dist}(o_j, q)\} \leq \max_{r \in G} \text{Dist}(r, q)$ and $\text{Dist}(o_i, o_j) \leq \max_{r_1, r_2 \in G} \text{Dist}(r_1, r_2)$, and thus $LOW_q(o_i, o_j) \leq \text{Cost}(q, G)$.

Hence, if $LOW_q(o_i, o_j)$ is larger than $curCost$, we know that o_i and o_j cannot contribute to the optimal group together. We denote C_o^r as the circle area with o as the center and with r as the radius, and we denote E_{o_1, o_2}^r as the ellipse area with o_1 and o_2 as the foci and with r as the transverse diameter. We have the following lemma to find the exhaustive search space around an object containing t_{inf} :

Lemma 4.18 *Given a MAX+MAX SGK query q and an object o_i containing t_{inf} , the most infrequent keyword in q , the search space around o_i is $C_{o_i}^{diam} \cap E_{q, o_i}^{curCost}$, where $diam = curCost - \text{Dist}(q, o_i)$. We denote this area by $S_{o_i}^q$.*

Proof: 1) Given q and o_i , if an object o_m whose distance to o_i is larger than $diam$, then $LOW_q(o_i, o_m) \geq \text{Dist}(q, o_i) + diam = curCost$. Thus, any object that is possible to combine with o_i to form the optimal group must be in $C_{o_i}^{diam}$.

2) Within $C_{o_i}^{diam}$, if an object o_j whose distance to q is further than that of o_i , then $LOW_q(o_i, o_j) = \text{Dist}(q, o_j) + \text{Dist}(o_i, o_j)$. Hence, if it is possible that o_i and o_j can contribute to the optimal group together, it must be true that $LOW_q(o_i, o_j) \leq curCost$, which means that o_j is in $E_{q, o_i}^{curCost}$.

Therefore, we can conclude that $C_{o_i}^{diam} \cap E_{q, o_i}^{curCost}$ is the search space around o_i . Figure 4.1 demonstrates the proof.

Figure 4.1: Exhaustive search space around o_i containing t_{inf} .

Based on this lemma, we can prune objects according to the following lemma.

Lemma 4.19 *Pruning Strategy*. *Given a MAX+MAX SGK query q and an object o , if for each object o_i containing t_{inf} , o is not in $\mathbb{S}_o^q$, o can be pruned.*

Proof: *The optimal group must contain one object containing t_{inf} . If o cannot contribute to the optimal group with any object containing t_{inf} together, we know it cannot be contained in the optimal group.*

This strategy can be extended to prune an MBR in the IR-tree index.

Lemma 4.20 *Pruning Strategy*. *Given an MBR R , if for each object o in O_{inf} , R does not intersect with $\mathbb{S}_o^q$, R can be pruned.*

Proof: *It is obviously correct based on Lemma 4.19.*

II. Enumerate the best group in the search space.

Next, we introduce how we do the exhaustive search within the search space around an object containing t_{inf} to find the group with the smallest cost containing this object. We call this object the “pivot” in the enumeration.

We adopt the depth-first-search strategy to do the enumeration. We use *selectedSet* to store the objects that are already selected in the current enumeration. The pivot is always in *selectedSet* since it must be contained in the generated group. We use *candidateSet* to store the objects within $\mathbb{S}_{pivot}^q$ (the search space of the pivot *w.r.t.* q) that are possible to combine with objects in *selectedSet* to form a group whose cost is smaller than the current best group.

The first part of the cost function of the MAX+MAX SGK query is determined by the furthest object to q in a group. Based on this fact, we first choose an object o_m to be the furthest object and put it in *selectedSet*. Then, in the following search, an object o_j should not be taken into consideration if $\text{Dist}(o_j, q) > \text{Dist}(o_m, q)$. This means that *candidateSet* only contains objects that are closer to q than is o_m . This greatly reduces the search space.

Then, we append new objects from *candidateSet* to *selectedSet* and this step is performed iteratively. This new object must cover a query keyword which has not been covered by the objects in *selectedSet* yet. The level of this depth-first-search is equal to the number of query keywords, because each object contains at least one new query keyword. The current best cost *curCost* is updated when a group covering all the query keywords with smaller cost is found.

Function enumerateBestGroup($\mathbb{S}, \text{pivot}, \text{curCost}, q$)

```

1  objList  $\leftarrow$  objects in  $\mathbb{S}$  in ascending order of the distances to  $q$ ;
2  candidateSet  $\leftarrow \emptyset$ ;
3  for each object  $o$  in objList do
4      candidateSet  $\leftarrow$  candidateSet  $\cup o$ ;
5      selectedSet  $\leftarrow \{\text{pivot}, o\}$ ;
6      pairDist  $\leftarrow \text{Dist}(\text{pivot}, o)$ ;
7      furDist  $\leftarrow \max(\text{Dist}(\text{pivot}, q), \text{Dist}(o, q))$ ;
8      if candidateSet. $\psi = q.\psi$  then
9          (cost, group)  $\leftarrow$ 
              search( $q, \text{curCost}, \text{selectedSet}, \text{candidateSet}, \text{pairDist}, \text{furDist}, o.di$ );
10         if cost < curCost then
11             curCost  $\leftarrow$  cost; curGroup  $\leftarrow$  group;
12 return curCost and curGroup
    
```

The pseudocode is described in Function enumerateBestGroup. The first step is to determine the furthest object. We process the objects in ascending order of their distances to the query q (line 1). In line 4, each object is added to *candidateSet* one by one to make sure that the last added object must be the furthest to the query location. *selectedSet* initially contains only *pivot* and the furthest object we chose, and hence *pairDist* and *furDist* are initialized correspondingly (lines 5–7). When *candidateSet* could cover all the query keywords we call the function *search*() to search the best solution containing *pivot* and o iteratively. If the new found group is better than the current group, it becomes the current best one (lines 8–11).

Although we can bound the search space according to Lemma 4.18 given a pivot, unfortunately, if the number of candidate objects in the search space is still large, the time cost of enumerating the best group increases prohibitively. We develop several pruning

strategies based on both textual and geometric properties to reduce the enumeration in the function *search()*.

Textual Pruning A new object is added to the selected object set in each search step, and *selectedSet. ψ* increases until it covers all the query keywords. Therefore, if an object *o* cannot contribute any new keyword to the selected objects set, this object can be ignored in the current search process. It means that it should not be inserted into *candidateSet* for the current *selectedSet*.

Distance Pruning Since the maximum distance to the query is already fixed (represented by *furDist*), the cost of any feasible solution is only determined by the second part in the cost function. Hence, if after an object is added to *selectedSet*, the cost of *selectedSet* is larger than *curCost*, the current search process can be terminated, because further search can obtain no better groups.

Termination Condition We can utilize the textual information to decide whether the current search can be terminated. If the objects in *candidateSet* cannot cover the keywords that have not been covered by *selectedSet*, we can stop since even if we select all the objects in *candidateSet* a group covering all the query keywords cannot be generated. Formally, if $selectedSet.\psi \cup candidateSet.\psi \not\supseteq query.\psi$, we stop the current search step.

The pseudocode is described in Function *search*. First, if *selectedSet* already covers all the query keywords, we compare this group with the current best group, and return the better one as the result (lines 1–5). Then, we begin the depth-first-search and append each object from *candidateSet* to *selectedSet* (lines 8–16). The textual pruning strategy is shown in line 9. Line 10 is used to avoid duplicated enumeration of the same group. The distance pruning strategy is shown in lines 11–14. The termination condition is checked in lines 17–18. Next, after we have appended a new object to *selectedSet*, we call the function recursively to find the groups with the new candidate objects set *newcandSet* and update *curCost* and *curGroup* correspondingly (lines 19–25).

III. The final exact algorithm for the MAX+MAX SGK query.

The exact algorithm is presented in Algorithm 12. We use the IR-tree index to find the objects containing the most infrequent keyword t_{inf} , and when the distance to *q* reaches beyond *curCost*, we terminate the algorithm. After we get an object containing t_{inf} , we first obtain its search range according to Lemma 4.18 utilizing the IR-tree (lines 16–25). Next, we call Function *enumerateBestGroup* to find the best group containing this object (line 26), and it is used to update the current best group (lines 27–29). Finally, we return *curCost* and *curGroup* (line 30).

Assume that there are *n* query keywords, and the number of objects that contain the most infrequent keyword is N_{inf} . This algorithm performs the exhaustive search around N_{inf} objects. Assume that the number of objects in the search range around an object containing the most infrequent keyword is *r* in the worst case. The cost of the exhaustive search around the object is $O(r^n)$. Thus, the complexity of this algorithm is $O(N_{inf}(F + r^n))$.

Function $\text{search}(q, \text{curCost}, \text{selectedSet}, \text{candidateSet}, \text{pairDist}, \text{furDist}, \text{stId})$

```

1 if  $\text{selectedSet}.\psi = q.\psi$  then
2   if  $\text{furDist} + \text{pairDist} < \text{curCost}$  then
3      $\text{curCost} \leftarrow \text{furDist} + \text{pairDist};$ 
4      $\text{curGroup} \leftarrow \text{selectedSet};$ 
5   return  $\text{curCost}$  and  $\text{curGroup};$ 
6  $\text{nextcandSet} \leftarrow \emptyset;$ 
7  $\text{leftKeywords} \leftarrow \emptyset;$ 
8 for each candidate object  $o_c$  in  $\text{candidateSet}$  do
9   if  $o_c.\psi \subseteq \text{selectedSet}.\psi$  then continue;
10  if  $o_c.di < \text{stId}$  then continue;
11   $\text{selectedDiam} \leftarrow 0;$ 
12  for each selected object  $o_s$  in  $\text{selectedSet}$  do
13     $\text{selectedDiam} = \max(\text{selectedDiam}, \text{Dist}(o_s, o_c));$ 
14  if  $\text{selectedDiam} + \text{furDist} > \text{curCost}$  then continue;
15   $\text{nextcandSet} \leftarrow \text{nextcandSet} \cup o_c;$ 
16   $\text{leftKeywords} \leftarrow \text{leftKeywords} \cup o_c.\psi;$ 
17 if  $\text{leftKeywords} \cup \text{selectedSet}.\psi \neq \text{query}.\psi$  then
18   return  $\text{curCost}$  and  $\text{curGroup};$ 
19 for each object  $o_n$  in  $\text{nextcandSet}$  do
20    $\text{selectedSet} \leftarrow \text{selectedSet} \cup o_n;$ 
21    $\text{pairDist} \leftarrow \max(\text{pairDist}, \text{selectedDiam});$ 
22    $(\text{cost}, \text{group}) \leftarrow$ 
23      $\text{search}(q, \text{curCost}, \text{selectedSet}, \text{nextcandSet}, \text{pairDist}, \text{furDist}, o_n.di);$ 
24   if  $\text{cost} < \text{curCost}$  then
25      $\text{curCost} \leftarrow \text{cost}; \text{curGroup} \leftarrow \text{group};$ 
26    $\text{selectedSet} \leftarrow \text{selectedSet} \setminus o_n;$ 

```

4.4 Processing Top- k Spatial Group Keyword Queries

It is attractive to be able to return several groups, thus providing users with more options. Based on Definition 4.1, we define the top- k spatial group keyword query:

Definition 4.4 Top- k Spatial Group Keyword Query: A top- k spatial group keyword (k SGK) query q is in the form of $\langle q.\lambda, q.\psi \rangle$, where $q.\lambda$ is a location and $q.\psi$ represents a set of keywords. It finds k groups of objects $\mathcal{X}_k = \langle \chi_1, \dots, \chi_k \rangle$, where χ_i ($1 \leq i \leq k$) is a feasible group of q , such that any feasible group $\chi_m \notin \mathcal{X}_k$ has cost value larger than the cost of any group $\chi_i \in \mathcal{X}_k$.

The proposed algorithms for answering the SGK queries can easily be extended to answer the corresponding top- k SGK (k SGK) queries. We cover the exact top- k algorithms for the SUM and MAX+MAX SGK queries in Section 4.4.1 and Section 4.4.2, respectively.

Algorithm 12: MAXMAX-Exact2($q, irTree$)

```

1   $U \leftarrow$  new min-priority queue;
2   $U.Enqueue(irTree.root, 0)$ ;
3   $(curGroup, curCost) \leftarrow$  MAXMAX-Appro2( $q, irTree$ );
4   $t_{inf} \leftarrow$  the most infrequent keyword in  $q.\psi$ ;
5  while  $U$  is not empty do
6     $e \leftarrow U.Dequeue()$ ;
7    if  $e$  is not an object then
8      if  $\text{minDist}(e, q) \geq curCost$  then break
9      foreach entry  $e'$  in node  $e$  do
10         if  $t_{inf} \in e'.\psi$  then
11           if  $e$  is a leaf node then
12              $U.Enqueue(e', \text{Dist}(e', q))$ ;
13           else  $U.Enqueue(e', \text{minDist}(e', q))$ ;
14   else
15     if  $\text{Dist}(e, q) \geq curCost$  then break
16      $\mathbb{S} \leftarrow \emptyset$ ;
17      $W \leftarrow$  new min-priority queue;
18      $W.Enqueue(irTree.root, 0)$ ;
19     while  $W$  is not empty do
20        $e' \leftarrow W.Dequeue()$ ;
21       if  $e'.Key > curCost$  then break
22       if  $e'$  is a node then
23         foreach node  $n$  in  $e'$  do
24           if  $n.\psi \cap q.\psi \neq \emptyset$  then  $W.Enqueue(n, LOW_q(e, e'))$ ;
25       else  $\mathbb{S} \leftarrow \mathbb{S} \cup e'$ ;
26      $(group, cost) \leftarrow \text{enumerateBestGroup}(\mathbb{S}, e, curCost, q)$ ;
27     if  $cost < curCost$  then
28        $curCost \leftarrow cost$ ;
29        $curGroup \leftarrow group$ ;
30 return  $curGroup$  and  $curCost$ ;

```

4.4.1 Top- k SUM SGK Query

We extend the exact algorithm presented in Section 4.2.4 to answer the SUM k SGK query efficiently. We still utilize the IR-tree, and we process the objects in ascending order of their distances to a given query q . To find the top- k results, the stopping condition is changed to these two: 1) we reach the k^{th} object covering all the query keywords; and 2) we reach an object such that its distance to q is larger than that of the current k^{th} group covering all the query keywords.

The idea of this algorithm is as follows. We use an array of priority queues to store the top- k groups for each query keyword subset. Each group is represented by a 3-tuple

(*cost, objects, status*). We utilize the IR-tree to process objects that contain some query keywords according to the ascending order of their distances to the query, as we do in Algorithm 7.

In the SUM-ExactWIndex algorithm, each query keyword subset has a status telling whether its lowest cost has been found. This can be used to prune objects and MBRs that cover a subset that already has the lowest cost. We need to do similarly for finding the top- k results. Since each keyword subset has k groups, we also need to keep track of the status for each of them, i.e., whether they are really in the top- k list of this keyword subset. When we reach an object e , the groups with cost values smaller than the distance of e to q change their status to “marked.” When all the top- k groups of a keyword subset have status “marked,” the status of this subset is changed to “marked,” and it is further used to prune objects and MBRs.

The object e is then used to update the top- k groups of each keyword subset that is a subset of $e.\psi \cap q.\psi$. If this object is the k^{th} group of a keyword subset, we know that all the top- k groups have been found for this subset, and we move it to *markedSet*. Finally, this object e is used to generate new groups with existing groups, as is done in lines 35–44 in SUM-ExactWIndex. We enumerate each query keyword subset, and for each group of that subset, it is combined with e to generate a new group for the keyword subset that is the union of $e.\psi$ and the current keyword subset. When the termination condition is satisfied, we stop the algorithm and return the k groups obtained.

In this algorithm, we maintain k groups for each keyword subset, and when we generate a new group for a keyword subset, we need to compare it with the existing groups of that subset, which has time complexity $O(\log k)$. Thus, the complexity of this algorithm is $O(k \log k)$ times that of algorithm SUM-ExactWIndex.

4.4.2 Top- k MAX+MAX SGK Query

It is straightforward to extend the MAXMAX-Exact algorithm to answer the MAX+MAX k SGK query. Recall that in MAXMAX-Exact, we first invoke MAXMAX-Appro2 to obtain a feasible group, the cost of which serves as an upper bound on the optimal cost. This upper bound cost can be utilized to obtain an exhaustive search range that is updated during the search. Thus, we just need to first obtain k groups approximately, and then we can do the same exhaustive search as in MAXMAX-Exact to find the k best groups.

We extend the MAXMAX-Appro1 algorithm to approximately find k feasible groups. First, we invoke MAXMAX-Appro1 to find a feasible group G_1 . Then, all the objects in G_1 are inserted into a queue in ascending order of their distances to q . We find the nearest object o_n to q from the queue, and for each keyword in $o_n.\psi$, we find the next nearest object containing this word, and we replace o_n with these objects to obtain a new group G_2 . The objects in G_2 not accessed yet are also inserted into the queue according to

their distances to q . We process all the remaining objects in the queue similarly. After selecting an object from the queue, we find all the discovered groups containing it, and we replace this object with further objects covering the same keyword subset in those groups to generate new groups. We stop this procedure when k groups are found.

After we find k feasible groups, the largest cost of these groups is the upper bound of the cost of the exact k^{th} group, and we denote this value by $curCost_k$. Then we find the most infrequent query keyword t_{inf} , and we process the objects containing t_{inf} (the pivot) in the ascending order of their distances to q . For each pivot, we obtain a search space utilizing Lemma 4.18 by replacing $curCost$ with $curCost_k$, and we find the best group containing the pivot by invoking Function `enumerateBestGroup`. If we find a group whose cost is smaller than $curCost_k$, we update the top- k list of groups and the value of $curCost_k$. When we reach an object whose distance is even larger than $curCost_k$, we can stop and return the k groups found already.

4.5 Experimental Study

4.5.1 Experimental Settings

Algorithms. For the SUM spatial group keyword query, we consider the approximation algorithm from Section 4.2.1 (denoted by SUM-A for short), the exact algorithms without index from Section 4.2.2 (denoted by SUM-EN1) and 4.2.3 (denoted by SUM-EN2), and the exact algorithm utilizing the IR-tree from Section 4.2.4 (denoted by SUM-EW). For the MAX+MAX spatial group keyword query, we evaluate the two approximation algorithms in Sections 4.3.1 and 4.3.2 (denoted by MAXM-A1 and MAXM-A2, respectively) and the two exact algorithms from Section 4.3.3 and 4.3.4 (denoted by MAXM-E1 and MAXM-A2, respectively). We also evaluate the approximation algorithm and the exact algorithm proposed by Long et al. [74], denoted by LONG-A and LONG-E, respectively.

We also conduct experiments for the algorithms answering k SGK queries, i.e., the exact algorithm of the top- k SUM SGK query in Section 4.4.1 (denoted by K-SUM-E for short), the exact algorithm (denoted by K-MAXM-E) of the top- k MAX+MAX SGK query in Section 4.4.2.

We evaluate the efficiency of algorithms in terms of both the query processing time and the I/O cost. Note that multiple layers of cache (e.g., disk driver cache, operating system cache, application cache) exist between the program and the disk. It is hard to measuring real physical I/Os. Instead, we report the simulated I/O cost. If a node for the R*-tree or B+-tree is visited, the number of simulated I/Os is increased by 1, and if a file is loaded into memory, the number of simulated I/Os is increased by the number of blocks (4K per block) for storing the file.

Dataset and queries. We use three datasets as described in Section 3.3.1 and in Table 3.1, i.e., **Hotel**, **Web**, and **GN**.

Hotel is small and is used to evaluate the performance of our algorithms when the dataset and index are memory resident, and the other two large datasets are used to evaluate our algorithms when the dataset and index are disk-based.

We generate 5 query sets in the space of **GN**, in which the number of keywords is 2, 4, 6, 8, and 10, respectively. We also generate 5 similar query sets in the space of both **Web** and **Hotel**. Each set comprises 50 queries. When generating a query set given the number of keywords n , we randomly select n objects from the dataset, and then we select a keyword from each object as the query keyword, and the center point of these objects is used as the query location. Such queries would need similar processing time, and we report the average cost of queries in each query set. We also conduct experiments on queries generated in other ways, and a summary of experimental results is presented in Section 4.5.2.3.

Setup. The IR-tree index structure is disk resident, and the page size is 4KB. The number of children of a node in the IR-tree is computed given the fact that each node occupies a page. This translates to 100 children per node in our implementation. All algorithms were implemented in C++ and run on an Intel(R) Xeon(R) CPU X5650 @2.66GHz with 4GB RAM.

4.5.2 Experimental Results on SGK Queries

4.5.2.1 SUM SGK Query

I. Results on Hotel. This experiment studies the performance of our algorithms when the dataset and index are in memory. Specifically, we study the efficiency and the accuracy of the four algorithms when we vary the number of query keywords on **Hotel**.

Figure 4.2.a, where the y-axis is in the logarithmic scale, shows the runtime of the five algorithms on the dataset **Hotel**. As expected, the approximation algorithm SUM-A runs faster than all the exact algorithms, *i.e.*, SUM-EN1, SUM-EN2, SUM-EW, and SUM-EP. The runtime of the approximation algorithm SUM-A increases almost linearly with the number of query keywords. It is understandable that its running time is in proportion to the number of query keywords: SUM-A keeps searching for the object with the lowest cost that covers part or all of the query keywords, and it terminates when a group of objects that covers the query keywords has been found.

For the exact algorithms, SUM-EN1 and SUM-EN2 need to scan the whole dataset and process all the objects that contain some query keywords, and thus they are slower than SUM-EW. SUM-EN2 runs faster than SUM-EN1 because the time complexity of the dynamic programming in SUM-EN2 is better than that in SUM-EN1.

Figure 4.2: Results of SUM SGK queries on **Hotel**

SUM-EW avoids scanning objects that do not contain query keywords by utilizing the IR-tree and can avoid accessing the objects whose distances to the query are larger than the cost of the discovered group, thus pruning the search space significantly. The experimental results demonstrate the usefulness of the IR-tree based pruning strategies. SUM-EP also utilizes the IR-tree. However, as analyzed in Section 4.2.4, when we reach an object e , all the keyword subsets of $e.\psi$ are considered in SUM-EP, which costs more time. Hence, it runs a bit slower than SUM-EW.

It can also be observed that the runtime of all the three exact algorithms increases with the number of query keywords; however, the increase is not exponential. The reason is that computing the costs of objects dominates the running time over the dynamic programming component.

Figure 4.2.b shows the accuracy of SUM-A on **Hotel**. It shows that the approximation algorithm is capable of achieving very accurate results.

We also conduct experiments on **Hotel** when the data and index are disk-based, and we observe qualitatively similar results.

II. Results on GN.

The objective of this set of experiments is to study the efficiency and the accuracy of the four algorithms when we vary the number of query keywords on **GN**. Figure 4.3.a shows the runtime of the algorithms on the dataset **GN**, and Figure 4.3.b shows the accuracy of SUM-A on **GN**.

Since this dataset contains a large amount of objects, the gap between the running time of the approximation algorithm and the exact algorithms is quite obvious. We can see the approximation algorithm SUM-A runs much faster than the exact algorithms. The runtime of the approximation algorithm SUM-A increases almost linearly with the number of query keywords.

SUM-EN2 is only slightly better than SUM-EN1. They almost have the same performance, because scanning the whole dataset costs too much time and it dominates the total query processing time. We can also see that SUM-EW still outperforms SUM-EP over all five query sets. The I/O cost is consistent with the runtime and thus is not reported.

III. Results on Web.

This experiment studies the efficiency and accuracy on the dataset **Web** in which each object is associated with a large set of keywords. Figure 4.3.a shows the runtime of the algorithms SUM-A and SUM-EW, and Figure 4.3.b shows the accuracy of SUM-A. To ensure readability of the figures, we omit SUM-EN1 and SUM-EN2 since they are inferior to SUM-EW (in orders of magnitude). We observe qualitatively similar results on **Web** as we do on **GN**. The I/O cost is consistent with the runtime.

4.5.2.2 MAX+MAX SGK Query

We first evaluate the performance of our algorithms on the three datasets, and then we compare our approximation and exact algorithms with the algorithms proposed by Long et al. [74].

I. Results on Hotel.

This experiment studies the performance of our algorithms when the dataset and index are in memory. Specifically, we study the efficiency and the accuracy of our two approximation (*i.e.*, MAXM-A1 and MAXM-A2) and two exact algorithms (*i.e.*, MAXM-E1 and MAXM-E2) when we vary the number of query keywords on **Hotel**. The algorithms LONG-A and LONG-E are memory-based, and we also evaluate their performance on **Hotel**.

Figure 4.3: Results of MAX+MAX SGK queries on **Hotel**

Figure 4.3.a shows the runtime of the five algorithms, and Figure 4.3.b shows the accuracy of the three approximation algorithms. Note that due to the hardness of answering the MAX+MAX SGK query, the exact algorithm may spend too much time on finding the optimal group (for some queries the two algorithms cannot return an answer even within one day). Therefore, to ensure the readability of the figures, we set a timeout threshold to 5 minutes. If the exact algorithm fails in finding a group within this threshold for a query, we terminate the algorithm. Figure 4.3.c shows the success rate of the two exact algorithms. The success rate of an algorithm is computed as the number of queries where the algorithm successfully finds a group within the pre-defined timeout threshold divided by the number of all testing queries.

As can be seen, MAXM-A1 outperforms MAXM-A2 in terms of runtime, and the accuracy of MAXM-A1 is worse than that of MAXM-A2. This is because MAXM-A1 terminates once a group of nearest objects covering query keywords is found. In contrast, MAXM-A2 may invoke MAXM-A1 multiple times. MAXM-A2 achieves good accuracy

compared with the optimal group returned by MAXM-E2. We also investigate the result of each query returned by MAXM-A1 and MAXM-A2, and it is shown that MAXM-A2 achieves statistically significant improvement over MAXM-A1 *w.r.t.* the ratio (p-value is smaller than 0.05 using t-test).

MAXM-E1 and MAXM-E2 are able to find the optimal group. However, they are much slower than the approximation algorithms. As expected, with the increase in the number of keywords, the runtime of MAXM-E2 increases exponentially due to its performing exhaustive search around the pivot objects. MAXM-E1 performs much worse than MAXM-E2, because it spends too much time on enumerating the possible combinations of IR-tree nodes. As shown in Figure 4.3.c, the success rate of MAXM-E1 drops dramatically as the number of query keywords increases. It is able to give an answer only on 10% of the queries (5 queries) when we set the number of query keywords to 10. MAXM-E2 can answer almost all the queries, and it only fails on 3 queries that contain 10 keywords.

When the number of keywords is small (*e.g.*, no larger than 10), the runtime of exact algorithms would be reasonable for applications without a high demand on query time, *e.g.*, finding research partners. However, approximation algorithms represent a better option when the query time is essential.

II. Results on GN.

Figure 4.4: Results of MAX+MAX SGK queries on GN

Figures 4.4.a shows the runtime of MAXM-A1, MAXM-A2, MAXM-E1, and MAXM-E2, Figure 4.4.b shows the accuracy of MAXM-A1 and MAXM-A2, and Figure 4.4.c shows the success rate of MAXM-E1 and MAXM-E2. Because **GN** contains a large number of objects, the number of candidate objects is usually large, and thus the query time is much longer than that on **Hotel**. MAXM-A1 still runs much faster than MAXM-A2, but its accuracy is much worse than that MAXM-A2, which is consistent with the observation on **Hotel**. The trend of the page access number is consistent with the

runtime. The success rate of both MAXM-E1 and MAXM-E2 drops quickly as the number of query keywords increases. MAXM-E1 can only find the answer for 1 query on the query set containing 8 keywords, and fails on all the queries when the number of query keywords is 10.

III. Results on Web.

Figure 4.5: Results of MAX+MAX SGK queries on **Web**

Figure 4.5.a shows the runtime of MAXM-A1, MAXM-A2, MAXM-E1, and MAXM-E2, Figure 4.5.b shows the accuracy of MAXM-A1 and MAXM-A2, and Figure 4.5.c shows the success rate of MAXM-E1 and MAXM-E2. We observe qualitatively similar results on **Web** as we do on **Hotel** and **GN**.

IV. Comparison with Existing Work.

Long et al. [74] study the problem of answering MAX+MAX SGK queries. We first briefly review their approximation algorithm LONG-A and their exact algorithm LONG-E, and next we compare our algorithms with LONG-A and LONG-E.

LONG-A processes the objects containing at least one query keyword in ascending order of their distances to the query q . On each such object o , it invokes MAXM-A1 to find a group for the query $\langle o.\lambda, q.\psi \setminus o.\psi \rangle$, and then it computes the cost of this group *w.r.t.* the original query q . If the group is better than the current best group, the current best cost is updated. It terminates when reaching an object whose distance is larger than the current best cost. They prove that LONG-A has an approximation ratio 1.375.

In LONG-E, they define the “query distance owner” of a group which is the furthest object to the query in the group, and the “pairwise distance owners” of a group which is the pair of two objects that have the largest distance. Then, they enumerate the query distance owner and the pairwise distance owners to find the optimal group. They process the objects in ascending order of their distances to the query. Each object is selected as the query distance owner, and then each pair of objects that are closer to the query is selected as the pairwise distance owners. If a better group is found, the current best cost

is updated. This process is repeated until reaching an object whose distance is larger than the current best cost.

We first compare our algorithms with LONG-A and LONG-E on **Hotel**, where the dataset and index are kept in memory¹.

Figure 4.6: Comparison of approximation algorithms on **Hotel**

Figure 4.7: Comparison of exact algorithms on **Hotel**

Figures 4.6.a and 4.6.b show the comparison of our approximation algorithm MAXM-A2 with LONG-A in terms of both efficiency and accuracy. Because both MAXM-A2 and LONG-A outperforms MAXM-A1 significantly *w.r.t.* the accuracy, we do not show the results of MAXM-A1 here. It can be observed that LONG-A always achieves better accuracy. It is able to return nearly optimal results. However, since it needs to invoke MAXM-A1 on all the candidate objects, it runs much slower than does MAXM-A2 which only invoke MAXM-A1 on objects containing the most infrequent keyword.

Figure 4.7.a shows the success rate of the exact algorithms MAXM-E1, MAXM-E2 and LONG-E. The algorithm MAXM-E2 is able to successfully return results for most

¹The codes of LONG-A and LONG-E are provided by Long et al.

queries, and has a much better success rate than do MAXM-E1 and LONG-E. LONG-E can only answer all 50 queries containing 2 keywords, but it even fails on 3 queries for those containing 4 keywords.

The queries that LONG-E and MAXM-E2 can process within the time threshold is always a subset of those which can be answered by MAXM-E2. To compare the three exact algorithms, we first run MAXM-E2 and LONG-E on the queries where LONG-E can successfully return results, and the runtime is shown in Figure 4.7.b. Then, we run the three algorithm on the queries that all of them can answer in the time limit, and the runtime is shown in Figure 4.7.c. It can be observed that MAXM-E2 outperforms both MAXM-E1 and LONG-E, and LONG-E performs better than MAXM-E1 except on the query set containing 4 keywords. The reason is that, MAXM-E1 first does the enumeration in the IR-tree nodes space and then in the object space, but LONG-E and MAXM-E2 do the enumeration on the object space directly. In addition, MAXM-E2 does the enumeration only around the objects containing the most infrequent query keyword, while LONG-E does not consider the term frequency.

We then compare our algorithms with LONG-A and LONG-E on **GN**, which contains large amount of objects and the index is disk-resident.

Figure 4.8: Comparison of approximation algorithms on **GN**

Figure 4.8.a and 4.8.b show the comparison of our approximation algorithm MAXM-A2 with LONG-A. The ratio of LONG-A is very close to 1. It can always obtain nearly optimal groups. However, since it needs to invoke MAXM-A1 on all the candidate objects containing at least one query keyword and the index is disk-resident, it runs much slower than does MAXM-A2 due to the frequent I/O. MAXM-A2 runs much faster because it only invokes MAXM-A1 on objects containing the most infrequent keyword, and its ratio is only slightly worse than that of LONG-A.

Figure 4.9.a shows the success rate of the exact algorithms MAXM-E1, MAXM-E2 and LONG-E. LONG-E even fails on 10 queries when there are 2 query keywords. It can

Figure 4.9: Comparison of exact algorithms on **GN**

answer no queries containing 8 keywords, and only 1 query containing 10 keywords can be processed by it within the time limit.

We first run MAXM-E2 and LONG-E on queries that can be answered by LONG-E, and the runtime is shown in Figure 4.9.b. We then run the three algorithms on queries that can be answered by all of them, and the runtime is shown in Figure 4.9.c. Not all of them can successfully return results on queries containing 8 and 10 keywords, and we only show the results on queries containing 2 to 6 keywords. It can be observed that MAXM-E2 outperforms LONG-E consistently. LONG-E performs much worse than does MAXM-E1 on queries containing 2 and 4 keywords. The reason may be that the pruning strategy in MAXM-E2 is efficient on queries with a few keywords, and thus the enumeration of many groups is avoided.

In conclusion, LONG-A has better accuracy but costs much longer time comparing with MAXM-A2, especially when the index is disk-resident. We notice that some queries containing 10 keywords may cost more than half an hour on **GN**, because too many objects need to be checked. We also note that the largest difference on accuracy between LONG-A and MAXM-A2 is 8% (when the number of keywords is 8) is not great. LONG-E performs better than does MAXM-E1 when there are more than 4 query keywords, but always performs worse than does MAXM-E2.

4.5.2.3 Effects of Query Keywords Frequency

On the query set generated by randomly selecting the objects and then selecting the keywords from each object, both MAXM-E1 and MAXM-E2 may fail if the frequency of the query keywords is too high. Since in this case, there may exist too many objects containing the query keywords, which need to be checked during the exhaustive search. We found that on **Web**, the number of the candidate objects may reach to around 30 million on some queries, which is over 50% of the whole dataset! Thus, we generate another 5 query sets containing 10 keywords. In each query set, we still randomly selected

the objects, but when we select the keywords for the query, we omit the keywords with very high frequency. We rank the keywords according to their frequency, and for the 5 query sets, we omit the top 5%, 6%, 7%, 8%, and 9% frequent keywords, respectively. The set of this experiments is to evaluate the performance of MAXM-E2 on queries without very high frequent keywords. MAXM-E1 still fails on most queries, and thus its runtime is not reported. We also report the performance of the approximation algorithms for the MAX+MAX SGK query.

Figure 4.10: Results on queries without frequent keywords

Figure 4.10.a shows the runtime of the two approximation algorithms and the two exact algorithms for the MAX+MAX SGK query. MAXM-E2 is able to find the answers for all queries within 30 seconds. MAXM-A1 and MAX-A2 also run faster than running on the randomly generated query sets. The accuracy of all the approximation algorithms becomes better as well.

This set of experiment demonstrates that our algorithm is able to perform quite well when the very frequent keywords are not contained in the query. We check **Web** dataset, and most of the frequent keywords are the so-called “stop words,” such as “you” and “very” etc, which do not often appear in queries. Note that we use 10 keywords in this set of experiment. The number of query keywords usually does not exceed 5 as reported in the work [106], and MAXM-E2 can answer all queries with no more than 6 keywords within the timeout threshold as shown in the experimental study. Hence, our proposed exact algorithm is applicable in most cases. In addition, if the exact algorithm runs too long, we can invoke the approximation algorithm MAXM-A2 to get nearly optimal results.

It is interesting to see that the exact algorithm MAXM-E2 performs better on queries containing only frequent keywords than on queries containing both frequent and infrequent keywords. After analyzing the results, we find the reason is that, if all query keywords are frequent, the result group usually has a small diameter, and the pruning technique successfully improves the efficiency. However, if a query contains both frequent

and infrequent keywords, the result group has a large diameter, and the exhaustive search space is large, which contains many relevant objects. As a result, there are a large number of candidate groups to be checked.

4.5.2.4 Scalability

Figure 4.11: Scalability of algorithms

To evaluate scalability, we generate 5 datasets containing from 2 to 10 million objects: we generate new locations by copying the locations in **GN** to nearby locations while maintaining the real distribution of the objects; for each new location, a document is selected randomly from the text descriptions of the objects in **GN**. Figure 4.11 shows the runtime of SUM-A and SUM-EW for the **SUM SGK** query, the runtime of MAXM-A1 and MAXM-A2 for the **MAX+MAX SGK** query (the number of query keywords is 10). All the algorithms scale well with the size of the dataset. The accuracy changes only slightly and is not affected by the dataset size, and thus is ignored.

SUM-EN1 and SUM-EN2 runs much slower than SUM-EW and are omitted. The runtime of MAXM-E1 and MAXM-E2 increases exponentially with the dataset size. Thus, to ensure readability of the figure, we omit them, which are orders of magnitudes slower than the approximation algorithms.

4.5.2.5 Experimental Results on Top- k SGK Queries

Figure 4.12: Runtime (top- k)

We study the performance of the modified versions of the algorithms for processing the top- k SGK queries, i.e., the exact algorithm of the top- k SUM SGK K-SUM-E and the exact algorithm K-MAXM-E of the top- k MAX+MAX SGK query. On the randomly generated query sets, MAXM-E2 may fail to give an answer within the 5 minutes time limit. Since K-MAXM-E is extended from MAXM-E2, it also cannot return answers for the queries where MAXM-E2 fails. Thus, we conduct this set of experiment on the query set generated as described in Section 4.5.2.3 by avoiding the top 9% frequent keywords.

The results are shown in Figure 4.12. We can see that the runtime of K-MAXM-E increases linearly with the value of k , and the runtime of K-SUM-E increases a bit more faster. This is consistent with the complexity analysis in Section 4.4.1. The complexity of K-SUM-E is $O(k \log k)$ times that of SUM-EW.

4.5.2.6 Real Examples

4.13.a: Result of SUM SGK Query

4.13.b: Result of MAX+MAX SGK Query

Figure 4.13: Real Examples

In order to show a real-world example, we crawled a dataset from Foursquare² in the region of Singapore. It consists of 23,720 points of interest, each of which contains a latitude and longitude, its categories (restaurant, mall, etc.), and a description provided by users who checked into the point of interest. We set the query location (assuming to be a user’s working place) at a building near the Raffles Place MRT station, and the query keywords are “supermarket,” “pub,” “sushi,” and “movie.” Figure 4.13 shows the results returned by exact algorithms of the SUM and MAX+MAX SGK query. It can be observed that the objects in the group of the SUM query are around the query location, and this is convenient if the user would like to go back to the building for a rest. The objects in the group of MAX+MAX query are quite close to each other, and this facilitates the user to visit them one by one.

²<http://foursquare.com>

4.5.2.7 Summarization of Empirical Study

Based on the experiments conducted on the three datasets, i.e., **GN**, **Web**, and **Hotel**, we can conclude as follows: For answering the **SUM SGK** query, the approximation algorithm has the best efficiency at the cost of accuracy. The exact algorithm utilizing the IR-tree always outperforms the algorithms without an index. For the **MAX+MAX SGK** query, the 3-approximation algorithm runs the fastest, but its accuracy is the worst. The 1.8-approximation algorithm runs much faster than the exact algorithms, and can always return results nearly optimal. The exact algorithm **MAXMAX-Exact1** runs much slower than the algorithm **MAXMAX-Exact2**, because **MAXMAX-Exact2** performs the exhaustive search directly on the object space. **MAXMAX-Exact2** also outperforms a state-of-the-art method for the **MAX+MAX SGK** query. All the algorithms run slower as the number of query keywords increases.

4.6 Additional Related Work

The work [113,114] retrieves groups of spatial keyword objects relates to the *mCK* query that takes a set of m keyword as argument. It returns m objects of minimum diameter that match the m keywords. It is assumed that each object in the result corresponds to a unique query keyword. In contrast, our query takes both a spatial location and a set of keywords as arguments, and its semantics are quite different from those of the *mCK* query. Long et al. [74] also propose algorithms to answer our **MAX+MAX SGK** query. As analyzed in Section 4.5.2.2, although their approximation algorithm has better accuracy, it costs too much time when the number of query keywords increases. Their exact algorithm performs better than our first exact algorithm which first does the enumeration in the index space, but worse than our second exact algorithm which does the enumeration directly in the object space.

The **SGK** query is also related to the team formation problem. The work [67] studies the problem of finding a team of experts in social networks. The objective is to find a group of persons each of which has some skills, from the social network, such that they can collaboratively finish a task and their communication cost is minimized. This work does not consider the spatial information. Yang et al. [107] propose the Social-Spatial Group Query, which is to select a group of nearby attendees with tight social relation. It is required that the total spatial distance of the attendees to the rally point is minimized while a certain social constraint of these attendees must be satisfied. The cost function is similar to the **SUM SGK** query. However, they do not consider the textual parts.

4.7 Conclusion

We present the new problem of retrieving a group of spatial objects, each associated with a set of keywords, such that the group covers the query's keywords and has the lowest cost measured by their distance to the query point, and the distances between the objects in the group. We study two particular instances of the problem, both of which are NP-hard. We develop approximation algorithms with provable approximation bounds and exact algorithms to solve the two problems. Results of experimental evaluation offer insight into the efficiency and the accuracy of the approximation algorithms, and the efficiency of the exact algorithms.

Chapter 5

Keyword-aware Optimal Route Search

This chapter is organized as follows: Section 5.1 formally defines the problem and shows the computational complexities of the problem. Section 5.2 presents the proposed algorithms. The empirical studies are reported in Section 5.3. Finally, Section 5.4 covers the additional related work and Section 5.5 offer conclusions.

5.1 Problem Statement

We define the problem of the keyword-aware optimal route (KOR) query, and show the hardness of the problem.

This problem is defined over a general graph $\mathcal{G}$. Specifically, $\mathcal{G} = (V, E)$ consists of a set of nodes V and a set of edges $E \subseteq V \times V$. Each node $v \in V$ represents a location associated with a set of keywords denoted by $v.\psi$; each edge in E represents a directed route between two locations in V , and the edge from v_i to v_j is represented by (v_i, v_j) . $\mathcal{G}$ can be a road network graph, or a graph extracted from users' historical trajectories. Depending on the source of $\mathcal{G}$, each edge in $\mathcal{G}$ is associated with different types of attributes. For example, if $\mathcal{G}$ is a traffic network, the attributes can be travel duration, travel distance, popularity, and travel cost. To keep our discussion simple, we consider directed graphs only in this chapter. However, our discussion can be extended to undirected graphs straightforwardly.

We define the optimal route based on two attributes on each edge (v_i, v_j) : 1) one attribute is used as the *objective value* of this edge, and it is denoted by $o(v_i, v_j)$ (e.g., the popularity), and 2) the other attribute is used as the *budget value* of this edge, which is denoted by $b(v_i, v_j)$ (e.g., the travel time). Note that we can pick up any two attributes to define the optimal route depending on different applications.

Definition 5.1 Objective Score and Budget Score. Given a route $R = \langle v_0, v_1, \dots, v_n \rangle$, the objective score of R is defined as the sum of the objective values of all the edges in R , i.e.,

$$\text{OS}(R) = \sum_{i=1}^n o(v_{i-1}, v_i),$$

and the budget score is defined as the sum of the budget values of all the edges in R , i.e.,

$$\text{BS}(R) = \sum_{i=1}^n b(v_{i-1}, v_i).$$

Figure 5.1: An example of $\mathcal{G}$

Figure 5.1 shows an example of the graph $\mathcal{G}$. We consider only five keywords (t_1 – t_5), and each keyword is represented by a distinct shape. For simplicity, each node contains a single keyword in the example. On each edge, the score inside a bracket is the budget value, and the other number is the objective value. For example, given the route $R = \langle v_0, v_3, v_5, v_7 \rangle$, we have $\text{OS}(R) = 2 + 3 + 4 = 9$ and $\text{BS}(R) = 2 + 2 + 1 = 5$.

Intuitively, a *keyword-aware optimal route* (KOR) query is to find an optimal route from a source to a target in a graph such that the route covers all the query keywords, its budget score satisfies a given constraint, and its objective score is optimized. Formally, we define the KOR query as follows:

Definition 5.2 Keyword-aware Optimal Route (KOR) Query.

Given $\mathcal{G}$, the keyword-aware optimal route query $\mathcal{Q} = \langle v_s, v_t, \psi, \Delta \rangle$, where v_s is the source

location, v_t is the target location, ψ is a set of keywords, and Δ specifies the budget limit, aims to find the route R starting from v_s and ending at v_t (i.e., $\langle v_s, \dots, v_t \rangle$) such that

$$R = \arg \min_R \text{OS}(R), \text{ subject to } \psi \subseteq \bigcup_{v \in R} (v.\psi), \text{BS}(R) \leq \Delta$$

In the example graph in Figure 5.1, given a query $\mathcal{Q} = \langle v_0, v_7, \{t_1, t_2, t_3\}, 8 \rangle$, the optimal route is $R_{\text{opt}} = \langle v_0, v_3, v_4, v_7 \rangle$ with objective score $\text{OS}(R_{\text{opt}}) = 4$ and budget score $\text{BS}(R_{\text{opt}}) = 7$. If we set Δ to 6, the optimal route becomes $R_{\text{opt}} = \langle v_0, v_3, v_5, v_7 \rangle$ with $\text{OS}(R_{\text{opt}}) = 9$ and $\text{BS}(R_{\text{opt}}) = 5$.

Theorem 5.11 *The problem of solving KOR queries is NP-hard.*

Proof: This problem can be reduced from the NP-hard weight-constrained shortest path problem (WCSP) [46]. Given a graph in which each edge has a length and a weight, WCSP finds a path that has the shortest length with the total weight not exceeding a specified value. The problem of answering KOR queries is a generalization of WCSP. If each node already covers all the query keywords, the problem of solving KOR becomes equivalent to the WCSP.

Obviously, if we disregard the query keyword constraint, the problem of solving KOR becomes WCSP. In addition, if we remove the budget constraint, the problem becomes similar to the generalized traveling salesman problem (GTSP) [52], which is also NP-hard. In GTSP, the nodes of a graph are clustered into groups, and GTSP finds a path starting and ending at two specified nodes such that it goes through each group exactly once and has the smallest length. In the problem of solving KOR, we can extract the locations whose keywords overlap with ψ , and the locations that cover the same keyword form a group. Thus, the problem of solving KOR without the budget constraint is equivalent to the GTSP. Furthermore, if we disregard the objective score, the problem of finding a route that covers all the query keywords and satisfies the budget constraint is still intractable. It is obvious that the simplified problem is also equivalent to GTSP, and thus cannot be solved by polynomial-time algorithms. Many approaches have been proposed for solving GTSP and WCSP (e.g., [26, 35, 38, 97]). However, they cannot be applied to answer the KOR queries since one more constraint or objective must be satisfied in KOR compared with GTSP and WCSP.

In the KOR problem, we consider two hard constraints, namely, the keyword coverage and the budget limit, and aim at minimizing the objective score. The simplified versions that consider any two aspects are also NP-hard as we analyzed. Hence, it is challenging to find an efficient solution to answering KOR queries. If a route satisfies the two hard constraints, the route is called a *feasible solution* or a *feasible route*.

Furthermore, we can extend the KOR query to the *keyword-aware top-k route* (KkR) query. Instead of finding the optimal route defined in KOR, the KkR query is to return k routes starting and ending at the given locations such that they have the smallest objective scores, cover the query keywords, and satisfy the given budget constraint.

5.2 Algorithms

We present the pre-processing method in Section 5.2.1, the proposed approximation algorithm **OSScaling** with provable approximation bound in Section 5.2.2, the more efficient approximation algorithm **BucketBound** also with performance guarantee in Section 5.2.3, and the greedy algorithm **Greedy** in Section 5.2.4.

5.2.1 Pre-processing

We introduce the pre-processing method. We utilize the pre-processing results to accelerate the algorithms to be proposed.

We use the Floyd-Warshall algorithm [44], which is a well-known algorithm for finding all pairs shortest path, to find the following two paths for each pair of nodes (v_i, v_j) :

- $\tau_{i,j}$: the path with the smallest objective score. The objective score of this path is denoted by $\text{OS}(\tau_{i,j})$ and the budget score is denoted by $\text{BS}(\tau_{i,j})$.
- $\sigma_{i,j}$: the path with the smallest budget score. The objective score of $\sigma_{i,j}$ is denoted by $\text{OS}(\sigma_{i,j})$ and the budget score is denoted by $\text{BS}(\sigma_{i,j})$.

For example, after the pre-processing, for the pair of nodes (v_0, v_7) in Figure 5.1, we have $\tau_{0,7} = \langle v_0, v_3, v_4, v_7 \rangle$ with $\text{OS}(\tau_{0,7}) = 4$ and $\text{BS}(\tau_{0,7}) = 7$ and $\sigma_{0,7} = \langle v_0, v_3, v_5, v_7 \rangle$ with $\text{OS}(\sigma_{0,7}) = 9$ and $\text{BS}(\sigma_{0,7}) = 5$.

Only the objective and budget scores of $\tau_{i,j}$ and $\sigma_{i,j}$ are used in the proposed algorithms, while the two paths themselves are not. The space cost is $O(|V|^2)$, where $|V|$ represents the number of nodes in the graph. In general, the number of points of interests $|V|$ within a city is not large [65, 77].

We use an inverted file to organize the word information of nodes. An inverted file index has two main components: 1) A vocabulary of all distinct words appearing in the descriptions of nodes (locations), and 2) A posting list for each word t that is a sequence of identifiers of the nodes whose descriptions contain t . We use B^+ -tree for the inverted file index, which is disk resident.

5.2.2 Approximation Algorithm OSScaling

A brute-force approach to solving KOR is to do an exhaustive search: We enumerate all candidate paths from the source node. We can use a queue to store the partial paths. In each step, we select one partial path from the queue. Then it is extended to generate more candidate partial paths and those paths whose budget scores are smaller than the specified limit are enqueued. When a path is extended to the target node, we check

whether it covers all the query keywords and satisfies the budget constraint. We record all the feasible routes, and after all the candidate routes from the source node to the target node have been checked, we select the best one of all the feasible routes as the answer to the query.

However, the exhaustive search is computationally prohibitive. Given a query with a specified budget limit Δ , we know that the number of edges in a route exploited in the search is at most $\lfloor \frac{\Delta}{b_{min}} \rfloor$, where b_{min} is the smallest budget value of all edges in $\mathcal{G}$. Thus, the complexity of an exhaustive search is $O(d^{\lfloor \frac{\Delta}{b_{min}} \rfloor})$, where d is the maximum outdegree in $\mathcal{G}$ (notice that enumerating all the simple paths is not enough for answering KOR queries). To avoid the expensive exhaustive search, we devise a novel approximation algorithm **OSScaling**. It is challenging to develop such an algorithm.

The main problem of the brute-force approach is that too many partial paths need to be stored on each node. In order to reduce the cost of enumerating the partial paths, in **OSScaling**, we scale the objective values of edges (positive values) in $\mathcal{G}$ into integers utilizing a parameter ϵ . The scaling enables us to bound the number of partial paths explored, and further to design a novel algorithm that runs polynomially in the budget constraint Δ , $\frac{1}{\epsilon}$, the number of nodes and edges in $\mathcal{G}$, and is exponential in the number of query keywords (which is typically small). Furthermore, the objective score scaling guarantees that the algorithm always returns a route whose objective score is no more than $\frac{1}{1-\epsilon}$ times of that of the optimal route, if there exists one. This is inspired by the FPTAS (fully polynomial-time approximation scheme) for solving the well-known knapsack problem [98]. Note that the problem of answering KOR queries is different from the NP-hard problem knapsack and its solutions cannot be used.

We define a scaling factor $\theta = \frac{\epsilon o_{min} b_{min}}{\Delta}$, where o_{min} and b_{min} represent the smallest objective value and the smallest budget value of all edges in $\mathcal{G}$, respectively, and ϵ is a parameter in the range $(0, 1)$. Next, for each edge (v_i, v_j) , we scale its objective value $o(v_i, v_j)$ to $\hat{o}(v_i, v_j) = \lfloor \frac{o(v_i, v_j)}{\theta} \rfloor$. We call the graph with scaled objective values as the **scaled graph**, denoted by $\mathcal{G}_S$. Given a route $R = \langle v_0, v_1, \dots, v_n \rangle$ in $\mathcal{G}_S$, we denote its scaled objective score by $\hat{OS}(R) = \sum_{i=1}^n \hat{o}(v_{i-1}, v_i)$.

On the scaled graph, we still extend from the source node to create new partial paths until we reach the target node. However, if a partial path has both smaller scaled objective score and budget score than another one on the same node, the **OSScaling** algorithm ignores it. Before detailing the algorithm, we introduce the following important definitions.

Definition 5.3 Node Label. For each node v_i , we maintain a list of labels, in which each label corresponds to a path P_i^k from the source node v_s to node v_i . The label is denoted by L_i^k and is in format of $(\lambda, \hat{OS}, OS, BS)$, where $L_i^k.\lambda$ is the keywords covered by P_i^k , $L_i^k.\hat{OS}$, $L_i^k.OS$, and $L_i^k.BS$ represent the scaled objective score, the original objective score, and the budget score of P_i^k , respectively.

Example 5.11 In the example graph shown in Figure 5.1, assuming $\Delta = 10$ and $\epsilon = 0.5$, we can compute the value for θ : $\theta = \frac{0.5 \cdot b_{\min} \cdot o_{\min}}{10} = \frac{1}{20}$. Therefore, the objective value of each edge is scaled to 20 times of its original value. Given the two paths from v_0 to v_4 , i.e., $R_1 = \langle v_0, v_2, v_3, v_4 \rangle$ and $R_2 = \langle v_0, v_2, v_6, v_5, v_4 \rangle$. The label of R_1 is $L_4^0 = (\langle t_1, t_2, t_4 \rangle, 100, 5, 7)$ and the label of R_2 is $L_4^1 = (\langle t_1, t_2, t_4 \rangle, 120, 6, 11)$.

Each partial route is represented by a node label. At each node, we maintain a list of labels, each of which stores the information of a corresponding partial route from the source node to this node, including the query keywords already covered, the scaled objective score, the original objective score, and the budget score of the partial route. Many paths between two nodes may exist, and thus each node may be associated with a large number of labels. However, most of the labels are not necessary for answering KOR. Considering Example 1, at node v_4 , the label L_4^1 could be ignored since L_4^0 has both smaller objective and budget scores. This is because that in the route extended from L_4^1 , we can always replace the partial route corresponding to L_4^1 with that corresponding to label L_4^0 . We say that L_4^0 **dominates** L_4^1 :

Definition 5.4 Label Domination. Let L_i^k and L_i^l be two labels corresponding to two different paths from the source node v_s to node v_i . We say L_i^k **dominates** L_i^l iff $L_i^k.\lambda \supseteq L_i^l.\lambda$, $L_i^k.\hat{OS} \leq L_i^l.\hat{OS}$, and $L_i^k.BS \leq L_i^l.BS$.

Notice that in OSScaling we determine if a label dominates another one with regard to the scaled objective score instead of the original objective score. Therefore, it is likely that the label dominated has smaller original objective score, and hence the optimal route may be missed in this algorithm. This is the reason that OSScaling can only return approximate results. However, by doing so, the maximum number of labels on a node is bounded, which further bounds the complexity of OSScaling. We have the following lemma:

Lemma 5.21 On a node there are at most $2^m \lfloor \frac{\Delta}{b_{\min}} \rfloor \lfloor \frac{o_{\max} \Delta}{\epsilon o_{\min} b_{\min}} \rfloor$ labels, where m is the number of query keywords, ϵ is the scaling parameter, $b_{\min}$, $o_{\max}$, and $o_{\min}$ represent the smallest budget value, the largest objective value, and the smallest objective value of all edges in $\mathcal{G}$, respectively.

Proof: First, given m query keywords, there are at most 2^m keywords subset. Second, given the budget limit Δ , the number of edges in a route checked by our algorithm does not exceed $\lfloor \frac{\Delta}{b_{\min}} \rfloor$. Hence, the objective score of a route in $\mathcal{G}_S$ is bounded by $\lfloor \frac{\Delta}{b_{\min}} \rfloor \hat{o}_{\max} = \lfloor \frac{\Delta}{b_{\min}} \rfloor \lfloor \frac{o_{\max}}{\theta} \rfloor = \lfloor \frac{\Delta}{b_{\min}} \rfloor \lfloor \frac{o_{\max} \Delta}{\epsilon o_{\min} b_{\min}} \rfloor$. In conclusion, we only need to store at most $2^m \lfloor \frac{\Delta}{b_{\min}} \rfloor \lfloor \frac{o_{\max} \Delta}{\epsilon o_{\min} b_{\min}} \rfloor$ labels, because all the rest can be dominated by them.

Note that Lemma 5.21 gives an upper bound of the label number at a node. In practice, the number of labels maintained at a node is usually much smaller than this upper bound. We denote this upper bound by L_{max} .

Next, we introduce how to do the route extension using labels. This step is called *label treatment*:

Definition 5.5 *Label Treatment*. *Given a label L_i^k at node v_i , for each outgoing neighbor v_j of node v_i in $\mathcal{G}$, we create a new label for v_j : $L_j^t = (L_i^k.\lambda \cup v_j.\psi, L_i^k.\hat{OS} + \hat{o}(v_i, v_j), L_i^k.OS + o(v_i, v_j), L_i^k.BS + b(v_i, v_j))$.*

The label treatment step extends a partial route at node v_i forward to all the outgoing neighbor nodes of v_i , and thus more longer partial routes are generated. Note that the label treatment step is applied together with label domination checking.

Another important definition is how we compare the order of two labels:

Definition 5.6 *Label Order*. *Let L_i^k and L_j^t be two labels corresponding to two paths from source node v_s to node v_i and v_j (v_i and v_j can be either the same or different nodes), respectively. We say L_i^k has a lower order than L_j^t , denoted by $L_i^k \prec L_j^t$, iff $|L_i^k.\lambda| > |L_j^t.\lambda|$ or $(|L_i^k.\lambda| = |L_j^t.\lambda| \text{ and } L_i^k.\hat{OS} < L_j^t.\hat{OS})$ or $(|L_i^k.\lambda| = |L_j^t.\lambda|, L_i^k.\hat{OS} = L_j^t.\hat{OS}, \text{ and } L_i^k.BS < L_j^t.BS)$; otherwise, breaking the tie by alphabetical order of v_i and v_j .*

In Example 1, we say that $L_4^0 \prec L_4^1$, because they contain the same number of query keywords, and L_4^0 has smaller objective and budget scores. This definition decides which partial route is selected for extension in each step.

Now we are ready to present our algorithms. The basic idea is to keep creating new partial routes from the best one among all existing partial routes. From the viewpoint of node labels, we first create a label at the source node, and then we keep generating new labels that cannot be dominated by existing ones. We always select the one with the smallest order according to Definition 5.6 to generate new labels. If newly generated labels cannot be dominated by existing labels, they are used to detect and delete the labels dominated by them. We repeat this procedure until all the labels on the target node are generated, and finally the label with the best objective score satisfying the budget limit at the target node is returned. Note that this is not an exhaustive search algorithm and we will analyze the complexity after presenting the algorithm.

The pseudocode is presented in Algorithm 15. We use a min-priority queue Q to organize the labels, which are enqueued into Q according to their orders defined in Definition 5.6. We use variable U to keep track of the upper bound of the objective score, and use LL to store the last label of the current best route. We initialize U as ∞ , and set LL as $NULL$. We create a label at the starting node v_s and enqueue it into Q (lines 2–4).

Algorithm 13: OSScaling Algorithm

```

1 Initialize a min-priority queue  $Q$ ;
2  $U \leftarrow \infty$ ;  $LL \leftarrow NULL$ ;
3 At node  $v_s$ , create a label:  $L_s^0 \leftarrow (v_s.\psi, 0, 0, 0)$ ;
4  $Q.enqueue(L_s^0)$ ;
5 while  $Q$  is not empty do
6    $L_i^k \leftarrow Q.dequeue()$ ;
7   if  $L_i^k.OS + OS(\tau_{i,t}) > U$  then continue;
8   for each edge  $(v_i, v_j)$  do
9     Create a label  $L_j^l$  for  $v_j$ :
9      $L_j^l \leftarrow (L_i^k.\lambda \cup v_j.\psi, L_i^k.\hat{O}S + \hat{o}(v_i, v_j), L_i^k.OS + o(v_i, v_j), L_i^k.BS + b(v_i, v_j))$ ;
10    if  $L_j^l$  is not dominated by other labels on  $v_j$  and  $L_j^l.BS + BS(\sigma_{j,t}) < \Delta$  and
10     $L_j^l.OS + OS(\tau_{j,t}) < U$  then
11      if  $L_j^l$  does not cover all the query keywords then
12         $Q.enqueue(L_j^l)$ ;
13        for each label  $L$  on  $v_j$  do
14          if  $L$  is dominated by  $L_j^l$  then
15            remove  $L$  from  $Q$ ;
16      else
17        if  $L_j^l.BS + BS(\tau_{j,t}) < \Delta$  then
18           $U \leftarrow L_j^l.OS + OS(\tau_{j,t})$ ;
19           $LL \leftarrow L_j^l$ ;
20        else  $Q.enqueue(L_j^l)$ ;
21 if  $U$  is  $\infty$  then return “No feasible route exits”;
22 else Obtain the route utilizing  $LL$  and return it;

```

We keep dequeuing labels from Q until Q becomes empty (lines 5–20). We terminate the algorithm when Q is empty or when all the labels in Q has objective scores larger than U . In each while-loop, we first dequeue a label L_i^k with the minimum label order from Q (line 6). If the objective score of L_i^k plus the best objective score $OS(\tau_{i,t})$ from v_i to the target node v_t is larger than the current upper bound U , then the label definitely cannot contribute to the final result (line 7). Next, for each outgoing neighbor v_j of v_i , we create a new label L_j^l for it according to Definition 5.5 (line 9). If L_j^l can be dominated by other labels on the node v_j or if it cannot generate a feasible route (first, the budget score of L_j^l plus $BS(\sigma_{j,t})$, the best budget score to v_t , is larger than the budget constraint Δ ; second, the objective score of L_j^l plus $OS(\tau_{j,t})$, the best objective score to v_t , is larger than the current upper bound U), we ignore the new label (line 10); Otherwise, if it does not cover all the query keywords, we enqueue it into Q and use it to detect and delete the labels that are dominated by it on v_j (lines 11–15).

When we find that the current label L_j^l already covers all the query keywords, a feasible solution is found and we update the upper bound U (lines 16–20). First, if the

budget score of L_j^l plus the budget score of $\tau_{j,t}$ (the path with the best objective score from v_j to v_t) is smaller than U , we update the upper bound U , and the last label is also updated (lines 18–19); otherwise, we enqueue this label into Q for later processing. Finally, if U is never updated, we know that there exists no feasible route for the given query; otherwise, we can construct the route using the label LL (lines 21–22).

The following example illustrates how this algorithm works.

Figure 5.2: Steps of Example 5.12

Example 5.12 Consider the example graph in Figure 5.1, the query $\mathcal{Q} = \langle v_0, v_7, \{t_1, t_2\}, 10 \rangle$, and ϵ is set as 0.5. The steps of the algorithm are shown in Figure 5.2 and the contents of the labels generated are in Table 5.1.

Initially, we create a label $L_0^0 = (\emptyset, 0, 0, 0)$ at node v_0 and enqueue it into Q . After we dequeue it from Q , as shown in step (a), we generate the following three labels on all the outgoing neighbors of v_0 : L_1^0 , L_2^0 , and L_3^0 . The three labels are also enqueued into Q .

In the next loop, L_2^0 is selected because $L_2^0 \prec L_3^0 \prec L_1^0$. As shown in Step (b), we generate another two labels L_3^1 and L_6^0 . Note that the best budget score from v_6 to v_7 is 7 ($\text{BS}(\sigma_{6,7})=7$), and thus L_6^0 can be ignored since $L_6^0.\text{BS} + \text{BS}(\sigma_{6,7}) (=11) > \Delta$. According to the pre-processing results, $\text{OS}(\tau_{3,7})=2$ and $\text{BS}(\tau_{3,7})=5$. Therefore, in step (c), we get a feasible route $R_1 = \langle v_0, v_2, v_3, v_4, v_7 \rangle$ with $\text{OS}(R_1)=6$ and $\text{BS}(R_1)=10$. The upper bound U is updated as $\text{OS}(R_1)$, i.e., $U=6$.

Next, L_3^0 on node v_3 is selected. As shown in Step (d), we generate another three labels and enqueue them into Q : L_1^1 , L_4^0 , and L_5^0 . Now label L_5^0 already covers all the query keywords on v_5 . According to the pre-processing results, from v_5 to v_7 , the best objective score is 3 ($OS(\tau_{5,7})=3$) and the budget score of this path is 4. Utilizing the pre-processing results, as shown in step (e), we can obtain another feasible solution $R_2 = \langle v_0, v_3, v_5, v_4, v_7 \rangle$ with $OS(R_2)=8$ and $BS(R_2)(=8) < \Delta$ (Note that suppose $\Delta=7$ in Q , R_2 will not be a feasible result. Instead, we enqueue the label L_5^0 into Q , and in the next loop, we include the edge (v_5, v_7) and get a feasible route $\langle v_0, v_3, v_5, v_7 \rangle$).

The rest labels are treated similarly, and the best route is R_1 .

	L_0^0	L_1^0	L_1^1	L_2^0	L_3^0	L_3^1	L_4^0	L_5^0	L_6^0
λ	$\emptyset$	$\emptyset$	t_1	t_2	t_1	t_1, t_2	t_1	t_1, t_2	t_1, t_2
$\hat{OS}$	0	80	60	20	40	80	60	100	40
OS	0	4	3	1	2	4	3	5	2
BS	0	1	4	3	2	5	4	4	4

Table 5.1: Labels contents

Complexity: In each loop of **OSScaling**, we dequeue one label from Q . Thus, in the worst case we need $|V|L_{max}$ loops according to Lemma 5.21. Within one loop, 1) we generate new labels on a node and check the domination on its outgoing neighbors, taking $O(|E|L_{max})$ time by aggregate analysis; 2) we dequeue one label and the complexity is $O(\lg L_{max})$. Hence, we can conclude that the worst time complexity is $O(|V|L_{max} \lg L_{max} + |E|L_{max})$. In practice, the number of loops is much smaller than the worst case and the number of keywords of a query is quite small. Therefore, the algorithm **OSScaling** is able to return the result efficiently.

By scaling the objective values of edges in $\mathcal{G}$, the algorithm **OSScaling** is able to guarantee an approximation bound.

Approximation Bound: We denote the route found by **OSScaling** as R_{OS} , and the feasible route with the smallest scaled objective score in $\mathcal{G}_S$ as R_{G_S} . We have the following lemma:

Lemma 5.22 $OS(R_{G_S}) \geq OS(R_{OS})$.

Proof: In Algorithm 1, if we use the partial route with the smallest scaled objective score to update the upper bound at node v_j (line 18), the algorithm returns R_{G_S} . We denote the objective score of a route from v_p to v_q as $O_{p,q}$, and we know $O_{s,j}(R_{G_S}) = O_{s,j}(R_{OS})$. According to the algorithm, $O_{j,t}(R_{G_S}) \geq \tau_{j,t} = O_{j,t}(R_{OS})$, and thus $OS(R_{G_S}) = O_{s,j}(R_{G_S}) + O_{j,t}(R_{G_S}) \geq O_{s,j}(R_{OS}) + O_{j,t}(R_{OS}) = OS(R_{OS})$.

We denote the optimal route as R_{opt} . We have:

Theorem 5.12 $\text{OS}(R_{\text{opt}}) \geq (1 - \epsilon)\text{OS}(R_{\text{OS}})$.

Proof: From $\hat{o} = \lfloor \frac{o}{\theta} \rfloor$, we know that $o - \theta \leq \theta \hat{o} \leq o$. Therefore, $\text{OS}(R_{\text{opt}}) = \sum_{e \in R_{\text{opt}}} o_e \geq \sum_{e \in R_{\text{opt}}} \theta \hat{o}_e$, then:

$$\begin{aligned} \text{OS}(R_{\text{opt}}) &\geq \sum_{e \in R_{\text{opt}}} \theta \hat{o}_e \geq \sum_{e' \in R_{\mathcal{G}_S}} \theta \hat{o}_{e'} \geq \sum_{e' \in R_{\mathcal{G}_S}} (o_{e'} - \theta) \\ &\geq \sum_{e' \in R_{\mathcal{G}_S}} o_{e'} - \lfloor \frac{\Delta}{b_{\min}} \rfloor \theta \geq \sum_{e' \in R_{\mathcal{G}_S}} o_{e'} - \epsilon o_{\min} \end{aligned}$$

Because $\sum_{e' \in R_{\mathcal{G}_S}} o_{e'} \geq o_{\min}$, we can conclude that $\text{OS}(R_{\text{opt}}) \geq (1 - \epsilon) \sum_{e' \in R_{\mathcal{G}_S}} o_{e'} = (1 - \epsilon)\text{OS}(R_{\mathcal{G}_S}) \geq (1 - \epsilon)\text{OS}(R_{\text{OS}})$ (according to Lemma 5.22).

We can see that the parameter ϵ affects not only the running time of this algorithm but also the accuracy. There is a tradeoff between the efficiency and accuracy when selecting a value for ϵ . With a larger value of ϵ , **OSScaling** runs faster but the accuracy would drop; on the contrary, with a smaller value for ϵ we can obtain better routes but that needs longer query time.

Optimization: We design the following optimization strategies to further improve Algorithm 1.

Optimization Strategy 1: When processing a label L_i^k at node v_i , in addition to the labels generated by following the outgoing edges of v_i in the graph, we also generate a label on a node v_j such that $\text{BS}(\sigma_{i,j})$ has the smallest value among all the nodes containing a uncovered query keyword and $L_i^k.BS + \text{BS}(\sigma_{i,j}) + \text{BS}(\sigma_{j,t}) \leq \Delta$. The motivation of this strategy is to find a feasible solution as early as possible, and then it is used to update the upper bound and further to prune more labels. The following example illustrates the effectiveness of this strategy.

Example 5.13 Given $\mathcal{Q} = \langle v_2, v_7, \{t_4\}, 8 \rangle$, without this strategy, we first generate two labels: $L_3^0 = (\emptyset, 3, 2)$ on v_3 and $L_6^0 = (\emptyset, 1, 1)$ on v_6 . In the next step L_6^0 is selected, and a new label $L_5^0 = (\emptyset, 3, 7)$ is generated. At the third step we get a label covering some query keywords, i.e., $L_4^0 = (\{t_4\}, 5, 8)$, but this label cannot generate the optimal route ($L_4^0.BS + \text{BS}(\sigma_{4,7}) (=8+3) > \Delta (=8)$). With this strategy, we first generate a label $(\{t_2\}, 4, 4)$ on v_4 , and then get a feasible solution $R = \langle v_2, v_3, v_4, v_7 \rangle$ with $\text{OS}(R) = 5$ and $\text{BS}(R) = 7$. The upper bound U is updated as 5, and it can be used to prune L_6^0 due to $L_6^0.OS + \text{OS}(\tau_{6,7}) (=1+5) > U (=5)$. Thus, the algorithm is accelerated.

Optimization Strategy 2: When the query contains some very infrequent words, we can utilize the nodes that contain them to find the result more efficiently. In Algorithm 15, when we decide if a label L_i^k can be deleted, two specific conditions are checked: 1) if

$L_i^k.OS + OS(\tau_{i,t})$ is smaller than U ; 2) if $L_i^k.BS + BS(\sigma_{i,t})$ is smaller than Δ . We utilize the scores of the two pre-processed routes from v_i to the target node v_t . But if the path from v_i to the nodes containing the infrequent words have large objective or budget scores, we will waste a lot of time on extending the route from v_i . The reason is that, although the label L_i^k cannot be pruned by the two conditions, it cannot generate useful labels, and this is not known until we reach the nodes containing the infrequent words. We first obtain all the nodes containing the least infrequent word (which must be below a frequency threshold, such as appearing in less than 1% nodes) utilizing the inverted file; after we generate a label L_i^k , if it does not cover the least infrequent word, for each node l , we check two conditions: 1) $L_i^k.OS + OS(\tau_{i,l}) + OS(\tau_{l,t}) > U$; 2) $L_i^k.BS + BS(\sigma_{i,l}) + BS(\sigma_{l,t}) > \Delta$. If on each node containing infrequent words at least one condition is satisfied, this label can be discarded. The following example illustrates the effectiveness of this strategy.

Example 5.14 Given $\mathcal{Q} = \langle v_2, v_4, \{t_5\}, 10 \rangle$. The best route in $\mathcal{G}_S$ is $R = \langle v_2, v_3, v_1, v_3, v_4 \rangle$ with $OS(R) = 7$. Without this strategy, the label $(\emptyset, 1, 1)$ generated from v_2 on v_6 is enqueued into the queue. Suppose t_5 is the least infrequent word of $\mathcal{Q}$. Utilizing this strategy, we can see that the best objective score of the path from v_6 to v_4 passing v_1 containing t_5 is $OS(\tau_{6,1}) + OS(\tau_{1,4}) = 8$, which is already larger than Δ , and thus this label can be ignored. No more useless labels are generated from it.

5.2.3 Approximation Algorithm BucketBound

In the algorithm **OSScaling**, after we find a feasible solution, we still have to keep searching for a better route until all the feasible routes are checked. We propose a more efficient approximate method denoted by **BucketBound** with provable approximation bounds which is also based on scaling the objective scores into integers.

Before describing the proposed algorithm, we introduce the following lemma which lays a foundation of this algorithm.

Lemma 5.23 Given a label L_i^k at node v_i , the best possible objective score of the feasible routes that could be extended from the partial path represented by L_i^k is $L_i^k.OS + OS(\tau_{i,t})$. We denote the score by $LOW(L_i^k)$.

Proof: If $\tau_{i,t}$ and L_i^k cover all query keywords collectively, they constitute the best route extending from L_i^k and its objective score is equal to $L_i^k.OS + OS(\tau_{i,t})$. Otherwise, another route from v_i to v_t covering more keywords must be selected to construct a feasible route. This route has larger objective score than that of $\tau_{i,t}$, which results in a larger objective score of the final route.

In this algorithm, we divide the traversed partial routes into different “buckets” according to their best possible objective scores. We define the buckets as follows:

Definition 5.7 Label Buckets. The label buckets organize labels. Each bucket is associated with an order number and corresponds to an objective score interval—the r th bucket B_r corresponds to the following interval: $[\beta^r \text{OS}(\tau_{s,t}), \beta^{r+1} \text{OS}(\tau_{s,t})]$, where $\text{OS}(\tau_{s,t})$ is the best objective score from v_s to v_t and β is a specified parameter. A label L_i^k is in the bucket B_r if:

$$\beta^r \text{OS}(\tau_{s,t}) \leq \text{LOW}(L_i^k) < \beta^{r+1} \text{OS}(\tau_{s,t})$$

With this important definition, we proceed to present the approximation algorithm **BucketBound**. We denote the route found by **OSScaling** as R_{OS} . The basic idea is as follows: We keep selecting labels (partial routes) from the buckets. When selecting a label, we always choose the non-empty bucket with the smallest order number, and then select a label with the lowest label order from it. After a label L_i^k is generated, we compute the score $\text{LOW}(L_i^k)$ and we place this label to the corresponding bucket according to Definition 5.7. Utilizing the label buckets enables us to find a novel way to detect if a feasible route found is in the same bucket as R_{OS} . If we find such a route during the above procedure, we return it as the result. We denote the route found by **BucketBound** as R_{BB} .

We proceed to explain how to determine if the bucket where we find a feasible route contains R_{OS} .

This algorithm follows the basic label generation and selection approach in **OSScaling**. However, the strategies of generating and selecting labels are different. With such changed label generation and selection strategies, we have the following lemma:

Lemma 5.24 *If all the buckets $B_i (i = 0, \dots, r)$ are empty and no feasible solution is found yet, the objective score of R_{OS} satisfies: $\text{OS}(R_{OS}) \geq \beta^{r+1} \text{OS}(\tau_{s,t})$.*

Proof: Since any bucket $B_i (i \leq r)$ is empty, we know the label corresponding to R_{OS} must be selected from the subsequent buckets. Therefore, $\text{LOW}(L_j^l) > \beta^{r+1} \text{OS}(\tau_{s,t})$. According to Lemma 5.23, we know $\text{OS}(R_{OS}) \geq \text{LOW}(L_j^l) \geq \beta^{r+1} \text{OS}(\tau_{s,t})$.

Based on Lemma 5.24, we have Lemma 5.25. When the condition in Lemma 5.25 is satisfied, a feasible route and R_{OS} fall into the same bucket, and the algorithm terminates.

Lemma 5.25 *When a feasible route R_{BB} is found in the bucket B_{r+1} and all the buckets $B_0, B_1, \dots, B_r$ are empty, the route R_{OS} found by **OSScaling** is also contained in B_{r+1} .*

Proof: Because any bucket $B_i (i \leq r)$ is empty, according to Lemma 5.24, $\text{OS}(R_{OS}) \geq \beta^{r+1} \text{OS}(\tau_{s,t})$. Since $\text{OS}(R_{OS}) \leq \text{OS}(R_{BB})$ (R_{BB} is one feasible solution found in **OSScaling**), we know $\beta^{r+1} \text{OS}(\tau_{s,t}) \leq \text{OS}(R_{OS}) \leq \text{OS}(R_{BB}) < \beta^{r+2} \text{OS}(\tau_{s,t})$. According to Definition 5.7, R_{OS} also falls in B_{r+1} .

Figure 5.3: Procedure of Algorithm 2

Figure 5.3 illustrates the basic process of the proposed approximation algorithm **BucketBound**. As shown in the figure, we first select the label L_i^k from the bucket B_0 , and after the label treatment the new label is put into the bucket B_3 . Since B_0 becomes empty now, we proceed to select labels from B_1 . If B_0 , B_1 , and B_2 all become empty, according to Lemma 5.24 we can know $\text{OS}(R_{OS}) \geq \beta^3 \text{OS}(\tau_{s,t})$. If now we find a feasible route R_{BB} in the bucket B_3 , according to Lemma 5.25, it is assured that R_{OS} also falls into B_3 , and we return R_{BB} as the result.

Unlike Algorithm 15, the approximation algorithm terminates immediately when Lemma 5.25 is satisfied, which means a feasible solution is found. Note that the feasible solution may be different from the first feasible solution found by Algorithm 15. This algorithm is also capable of determining if a feasible route exists. If all buckets are empty during the label selection step and no feasible route found yet, there exists no result for **KOR**. This is because that when all buckets are empty, all the labels generated do not satisfy the budget constraint, which means that all the partial routes generated from the source node exceed the budget limit Δ .

The algorithm is detailed in Algorithm 16. It uses a min-priority queue for each bucket to organize the labels in the bucket. We initialize the first min-priority queue B_0 (corresponding to the first bucket with boundary $[\text{OS}(\tau_{s,t}), \beta \text{OS}(\tau_{s,t})]$); U and LL are initialized as in Algorithm 1. We initialize the flag *Found* as false, which records if a feasible route is found. We create a label at the source node v_s and enqueue it into B_0 (lines 1–4). The algorithm terminates when the flag *Found* is true. We keep dequeuing labels from B_r which represents the non-empty bucket with the smallest order number until we find a solution or no result exists (lines 5–23). If all queues become empty, it is assured that no feasible route exists (line 7). After we select a label L_i^k on node v_i , for each outgoing neighbor v_j of v_i , we create a new label for it (line 10). When a new label L_j^l is generated, we check: 1) if it can be dominated by other labels on v_j ; 2) if it cannot generate results definitely. If so, we ignore it (line 11); Otherwise, we use it

Algorithm 14: BucketBound Algorithm

```

1 Initialize a min-priority queue  $B_0$ ;
2  $LL \leftarrow NULL$ ;  $Found \leftarrow false$ ;
3 At node  $v_s$ , create label  $L_s^0 \leftarrow (v_s.\psi, 0, 0, 0)$ ;
4  $B_0.enqueue(L_s^0)$ ;
5 while  $Found$  is false do
6    $B_r \leftarrow$  the queue of the first non-empty bucket;
7   if All queues are empty then return “No feasible route exist”;
8    $L_i^k \leftarrow B_r.dequeue()$ ;
9   for each edge  $(v_i, v_j)$  do
10    Create a new label  $L_j^l$  for  $v_j$ :
11     $L_j^l \leftarrow (L_i^k.\lambda \cup v_j.\psi, L_i^k.\hat{OS} + \hat{o}(v_i, v_j), L_i^k.OS + o(v_i, v_j), L_i^k.BS + b(v_i, v_j))$ ;
12    if  $L_j^l$  is not dominated by other labels on  $v_j$  and  $L_j^l.BS + BS(\sigma_{j,t}) < \Delta$  then
13      Find  $B_s$  that  $L_j^l$  falls into;
14      if  $B_s$  does not exist then
15        Initialize a priority queue  $B_s$ ;
16         $B_s.enqueue(L_j^l)$ ;
17      for each label  $L$  on  $v_j$  do
18        if  $L$  is dominated by  $L_j^l$  then
19          remove  $L$  from the corresponding queue;
20      if  $L_j^l$  covers all the query keywords then
21        if  $B_r$  and  $B_s$  are the same queue then
22          if  $L_j^l.BS + BS(\tau_{j,t}) \leq \Delta$  then
23             $Found \leftarrow true$ ; // Lemma 5.25
24             $LL \leftarrow L_j^l$ ;
25 Obtain the route utilizing  $LL$  and return the route;

```

to delete labels on v_j that can be dominated by it, and we enqueue this label to the corresponding bucket according to its best possible objective score (lines 12–18). When L_j^l already covers all the query keywords and also falls into B_r , we still need to test if the path corresponding to $LOW(L_j^l)$ satisfies the budget constraint. If so, we find a solution and exit the loop according to Lemma 5.25 (lines 19–23).

Theorem 5.13 *Algorithm 16 offers the approximation ratio $\frac{\beta}{1-\epsilon}$.*

Proof: Assume that the solution R_{BB} is found in B_k . According to Lemma 5.25, the route found by OSScaling R_{OS} is also contained in B_k . Thus, we have $OS(R_{OS}) \geq \beta^k OS(\tau_{st})$ and $OS(R_{BB}) < \beta^{k+1} OS(\tau_{st})$. According to Theorem 5.12, we can get: $\frac{OS(R_{BB})}{OS(R_{opt})} = \frac{OS(R_{BB})}{OS(R_{OS})} \frac{OS(R_{OS})}{OS(R_{opt})} < \frac{\beta^{k+1} OS(\tau_{st})}{\beta^k OS(\tau_{st})(1-\epsilon)} = \frac{\beta}{1-\epsilon}$

The following example explains how Algorithm 16 works.

Example 5.15 Given $\mathcal{Q} = \langle v_0, v_3, \{t_5\}, 5 \rangle$ and $\beta = 1.5$. First, we get $\text{OS}(\tau_{0,3})=2$ and we create three queues B_0 , B_1 , and B_2 corresponding to the intervals $[2, 3)$, $[3, 4.5)$, $[4.5, 6.75)$, respectively.

Initially, we create a label $L_0^0 = (\emptyset, 0, 0)$ on node v_0 and enqueue it into B_0 . After we dequeue it from B_0 , we generate the following three labels on all the outgoing neighbors of v_0 : $L_1^0 = (\{t_5\}, 4, 1)$ on v_1 (enqueued into B_2), $L_2^0 = (\emptyset, 1, 3)$ on v_2 (ignored), and $L_3^0 = (\emptyset, 2, 2)$ on v_3 (enqueued into B_0). In the next loop, L_3^0 is selected from B_0 . We generate another three labels: $L_1^1 = (\{t_5\}, 3, 4)$ on node v_1 (enqueued into B_2), $L_4^0 = (\emptyset, 3, 4)$ on v_4 (ignored), and $L_5^0 = (\emptyset, 5, 4)$ on v_5 (ignored). Now B_0 and B_1 are empty, we begin to select labels from B_2 . We select L_1^1 because $L_1^1 \prec L_1^0$. After the treatment, we obtain a feasible route $R_{BB} = \langle v_0, v_3, v_1, v_3 \rangle$ with objective score 5. The route found by OSScaling is $R_{OS} = \langle v_0, v_3, v_1, v_3 \rangle$, which is the same with R_{BB} .

Although **BucketBound** has the same worst case complexity as Algorithm 1, it processes much fewer labels and is more efficient in practice. Note that the two optimization strategies in **OSScaling** are still applicable in **BucketBound**.

5.2.4 Greedy Algorithm

We propose an approximation algorithm using the greedy approach to solve **KOR**. It has no performance guarantee.

There are three constraints in the **KOR** problem: a) a set of keywords must be covered; b) the objective score must be minimized; c) the budget limit Δ must be satisfied. As discussed in Section 5.1, by considering only two of them, the problem is still NP-hard. Therefore, a greedy approach normally cannot guarantee that two constraints are satisfied. Since the keyword and budget constraints are hard constraints, we design a greedy algorithm such that it is able to find a route either covering all the query keywords or satisfying the budget constraint, while minimizing the objective score greedily.

The idea is that we start from the source node, and keep selecting the next best node according to a certain strategy until we finally reach the target node. The strategy of selecting the next node affects the results significantly. We design a greedy strategy that takes into account all the three constraints simultaneously to find the best next node: a) the node contains uncovered query keywords; and b) the best route that can be generated after including this node into the current partial route is expected to have a small objective score and fulfill the budget constraint. We use a parameter α to balance the importance of the objective and budget scores when selecting a node: at node v_i , when we extend the current partial route R_i ending at v_i , we select the node v_j that minimizes the following score:

$$\text{score}(v_j, R_i) = \alpha(R_i.OS + \text{OS}(\tau_{i,j}) + \text{OS}(\tau_{j,t})) + (1 - \alpha)(R_i.BS + \text{BS}(\tau_{i,j}) + \text{BS}(\tau_{j,t})) \quad (5.1)$$

When $\alpha = 0$, we select a node only based on the budget score, *i.e.*, selecting the node such that the budget score of the corresponding partial route plus the best budget score from the node to the target node v_t is the smallest. When $\alpha = 1$, the algorithm finds a node such that the objective score of the corresponding partial route plus the best objective score from the node to v_t is minimized.

Algorithm 15: Greedy Algorithm

```

1  $nodeSet \leftarrow \emptyset$ ;  $wordSet \leftarrow \mathcal{Q}.\Omega \setminus v_s.\psi$ ;
2  $v_{pre} \leftarrow v_s$ ;  $OS \leftarrow 0$ ;  $BS \leftarrow 0$ ;
3 for each word  $w_t \in wordSet$  do
4   | Get the location set  $lSet$  containing  $w_t$ ;
5   |  $nodeSet \leftarrow nodeSet \cup lSet$ ;
6 while  $wordSet$  is not empty do
7   |  $minS \leftarrow \operatorname{argmin}_{v_m \in nodeSet} \operatorname{score}(v_m, R_{pre})$ ;
8   |  $OS \leftarrow OS + OS(\tau_{pre,m})$ ;  $BS \leftarrow BS + BS(\tau_{pre,m})$ ;
9   |  $v_{pre} \leftarrow v_m$ ;
10  |  $wordSet \leftarrow wordSet \setminus v_m.\psi$ ;
11  | Remove the locations containing  $v_m.\psi$  from  $nodeSet$ ;
12  $OS \leftarrow OS + OS(\tau_{pre,t})$ ;  $BS \leftarrow BS + BS(\tau_{pre,t})$ ;
13 Return the route found with scores  $OS$  and  $BS$ ;
```

The pseudocode is outlined in Algorithm 17. We use $wordSet$ to keep track of the uncovered query keywords and $nodeSet$ to store all the locations containing uncovered query keywords (line 1). v_{pre} denotes the node where the current partial path ends and is initialized as v_s . OS and BS are used to store the objective and budget scores and both initialized to 0 (line 2). We utilize inverted file to find locations for $nodeSet$ (lines 3–5). The algorithm terminates when $wordSet$ is empty. While it is not empty, we find the best node according to Equation 5.1 (line 7), extend the partial route (line 8–9), and update $wordSet$ and $nodeSet$ (lines 10–11). After we exit the loop, we add the last segment from the partial route’s last node to the target node v_t to construct the final route and return (lines 12–13).

Algorithm 17 may fail to find a feasible route even if there exists a feasible one. In each step, it selects the next best node. If we find more nodes at each step, the accuracy will be better while the search space becomes much larger. Hence, it is a tradeoff between the accuracy and efficiency. In the experiments, we study the performance of Algorithm 17 when the best 2 nodes are selected at each step. We denote the algorithm selecting one node by **Greedy-1**, and the algorithm selecting two nodes by **Greedy-2**. The worst time complexity of **Greedy-1** is $O(mn)$ and for **Greedy-2** it is $O(2^m n)$, where m is the number of query keywords and n is the number of nodes in the graph.

Algorithm 17 guarantees that the query keywords are always covered while the budget limit may not be satisfied. This is desirable when the query keywords are important to

users (*e.g.*, the users do not want to miss any type of locations in their plan). However, if the budget score is very important (*e.g.*, the users cannot overrun their money budget), we modify this algorithm slightly to accommodate the need. We return a route with budget score not exceeding Δ while the query keywords may not be totally covered. We break the while-loop when the current partial route cannot be extended any more. That is, in line 6 in Algorithm 17, we check if $L.BS + BS(\sigma_{l,t}) > \Delta$ instead of if *wordSet* is empty.

5.2.5 Keyword-aware Top- k Optimal Route Search

We further extend the KOR query to the *keyword-aware top- k route* (KkR) query. Instead of finding the optimal route defined in KOR, the KkR query is to return the top- k routes starting and ending at the given locations such that they have the best objective scores, cover all the query keywords, and satisfy the given budget constraint. We introduce how to modify the OSScaling algorithm and the BucketBound algorithm for solving KkR approximately.

It is relatively straightforward to extend the two approximation algorithms OSScaling and BucketBound for processing the KkR query. Therefore, we only briefly present the extension. We need to introduce the definition of “ k -dominate”. A label is “ k -dominated” if at least k labels dominate it. In the pseudocode of OSScaling algorithm, we need to replace “dominate” by “ k -dominate.” Moreover, instead of keeping track of only the current best result, we need to track the current best k results. The budget score of the k th best route is used as the upper bound U to prune unnecessary labels. Similarly, in the BucketBound algorithm, we also apply “ k -dominate”. Moreover, instead of returning immediately when we find a feasible route in the bucket containing R_{OS} , the algorithm terminates when we find k feasible routes from the non-empty bucket with the smallest order number.

Note that we do not extend the greedy algorithm for solving KkR. The greedy approach is not able to guarantee that a feasible route can be found. Therefore, it is meaningless to return k routes using such a method.

5.3 Experimental Study

5.3.1 Experimental Settings

Algorithms. We study the performance of the following proposed algorithms: the approximation algorithm OSScaling in Section 5.2.2, the approximation algorithm BucketBound in Section 5.2.3, and the greedy algorithms in Section 5.2.4, denoted by Greedy-1 and Greedy-2 corresponding to selecting the top-1 and top-2 best locations, respectively.

Additionally, we also implemented a naive brute-force approach discussed in Section 5.2.2. However, it is at least 2 orders of magnitude slower than **OSScaling** and cannot finish after 1 day, and thus is omitted.

Data and queries. We use five datasets in our experimental study. The first one is a real-life dataset collected from Flickr using its public API. We collected 1,501,553 geo-tagged photos taken by 30,664 unique users in the region of the New York city in the United States. Each photo is associated with a set of user-annotated tags. The latitude and the longitude of the place where the photo is taken and its taken time are also collected. Following the work [65], we utilize a clustering method to group the photos into locations. We associate each location with tags obtained by aggregating the tags of all photos in that location after removing the noisy tags, such as tags contributed by only one user. Finally, we obtain 5,199 locations and 9,785 tags in total. Each location is associated with a number of photos taken in the location. Next, we sort the photos from the same user according to their taken time. If two consecutive photos are taken at two different places and the taken time gap is less than 1 day, we consider that the user made a trip between the two locations, and we build an edge between them.

On each edge, the Euclidean distance between its two vertices (locations) serves as the budget value. We compute a popularity score for each edge following the idea of the work [25]. The popularity of an edge (v_i, v_j) is estimated as the probability of the edge being visited: $Pr_{i,j} = \frac{Num(v_i, v_j)}{TotalTrips}$, where $Num(v_i, v_j)$ is the number of trips between v_i and v_j and $TotalTrips$ is the total number of trips. The total popularity score of a route $R = (v_0, v_1, \dots, v_n)$ is computed as: $PS(R) = \prod_{i=1}^n Pr_{i-1,i}$. However, the popularity score should be maximized. To transform the maximization problem to the minimization problem as defined in **KOR**, we compute the objective score on each edge (v_i, v_j) as: $o(v_i, v_j) = \log(\frac{1}{Pr_{i,j}})$. Therefore, if $OS(R)$ is minimized, $PS(R)$ is maximized.

The other 4 datasets are generated from real data, mainly for scalability experiment. By extracting the subgraph of the New York road network ¹, we obtain 4 datasets containing 5,000, 10,000, 15,000, and 20,000 nodes, respectively. Each node is associated with a set of randomly selected tags from the real Flickr dataset. The travel distance is used as the budget score, and we randomly generate the objective score in the range (0,1) on each edge to create the graphs for the four datasets.

We generate 5 query sets for the Flickr dataset, in which the number of keywords are 2, 4, 6, 8, and 10, respectively. The starting and ending locations are selected randomly. Each set comprises 50 queries. Similarly, we also generate 5 query sets for each of the 4 other datasets.

All algorithms were implemented in VC++ and run on an Intel(R) Xeon(R) CPU X5650 @2.66GHz with 4GB RAM.

¹<http://www.dis.uniroma1.it/~challenge9/download.shtml>

5.3.2 Experimental Results

5.3.2.1 Efficiency of Different Algorithms

The objective of this set of experiments is to study the efficiency of the proposed algorithms with variation of the number of query keywords and the budget limit Δ (travel distance). We set the default value for the scaling parameter ϵ in **OSScaling** and **BucketBound** at 0.5 (the middle of the value range of ϵ), the specified parameter β at 1.2 for **BucketBound** (can achieve good accuracy), and the default value for α in **Greedy** at 0.5 (the middle of the value range of α). We conduct the experiment to study the runtime when varying the value of ϵ for **OSScaling**, and the experiment to study the runtime when varying the value of β for **BucketBound** ($\epsilon=0.5$). Note that the runtime of **Greedy** is not affected by α .

Figure 5.4: Runtime (Flickr)

Figure 5.5: Runtime (Flickr)

Varying the number of query keywords. Figure 5.4 shows the runtime of the four algorithms on the Flickr dataset when we vary the number of query keywords. For each number, we report the average runtime over five runs, each using a different Δ , namely 3, 6, 9, 12, and 15 kilometers, respectively. Note that the y-axis is in logarithmic scale. We can see that all the algorithms are reasonably efficient on this dataset. As expected, the algorithm **OSScaling** runs much slower than the other three algorithms. **BucketBound** is usually 8-10 times faster than **OSScaling**, although **OSScaling** and **BucketBound** have the same worst time complexity. This is because **BucketBound** terminates immediately when a feasible route is found in the bucket containing R_{OS} , the route found by **OSScaling**, and thus it generates much fewer labels than does **OSScaling**. The worst time complexity of both **OSScaling** and **BucketBound** is exponential in the number of query keywords. However, as shown in the experiment, the runtime does not increase dramatically as the number of query keywords is increased. This is due to the two optimization strategies employed in both algorithms. Without employing the optimization strategies, both algorithms will be 3-5 times slower.

Greedy-1 is the fastest since it only selects the best node in each step. However, as to be shown, its accuracy is the worst. **Greedy-1** is not affected significantly by the number of query keywords. The runtime of **Greedy-2** increases dramatically with the increase of query keywords. This is because **Greedy-2** selects the best 2 nodes at each step, and its asymptotically tight bound complexity is exponential in the number of query keywords.

Varying the budget limit Δ . Figure 5.5 shows the runtime of the four approaches on the Flickr dataset with the variation of Δ . At each Δ , the average runtime is reported over 5 runs, each with a different number of query keywords from 2 to 10. The runtime of **OSScaling** grows when Δ increases from 3 km to 6 km as a smaller Δ can prune more routes. However, as Δ continues to increase, the runtime decreases slightly. This is due to the fact that with a larger Δ , **OSScaling** finds a feasible solution earlier (since Δ is more likely to be satisfied), and then the feasible solution can be used to prune the subsequent search space. The saving dominates the extra cost incurred by using larger Δ (notice that larger Δ deteriorates the worst-case performance rather than the average performance). As for the other approximation algorithms, their runtime is almost not affected by the budget limit as shown in the figure.

Figure 5.6: Runtime (Flickr)

Figure 5.7: Relative Ratio (Flickr)

Varying the parameter ϵ for OSScaling. Figure 5.6 shows the runtime of **OSScaling** when we vary the value of ϵ . We set Δ as 6 km and the number of query keywords as 6. It is observed that **OSScaling** runs faster as the value of ϵ increases. This is because when ϵ becomes larger, L_{max} , the upper bound of the number of labels on a node is decreased, and thus more labels (representing partial routes) can be pruned during the algorithm. This is consistent with the complexity analysis of **OSScaling**, which shows that **OSScaling** runs linearly in $\frac{1}{\epsilon}$.

Varying the parameter β for BucketBound. Figure 5.8 shows the runtime of **BucketBound** when we vary the value of β , the specified parameter. In this set of experiments, $\Delta=6$ km, $\epsilon=0.5$, and the number of query keywords is 6. As expected, **BucketBound** runs

Figure 5.8: Runtime (Flickr)

Figure 5.9: Relative ratio (Flickr)

faster as the value of β increases. This is because when β becomes larger, the interval of each bucket becomes larger and each bucket can accommodate more labels. Hence, it is faster for **BucketBound** to find a feasible solution in the bucket containing the best route in $\mathcal{G}$.

5.3.2.2 Accuracy of Approximation Algorithms

The purpose of this set of experiments is to study the accuracy of the approximation algorithms. The brute-force method discussed in Section 5.2.2 failed to finish for most of settings after more than 1 day. We note that in the very few successful cases (small Δ and keywords), the practical approximation ratios of **OSScaling** and **BucketBound** are a lot smaller than their theoretical bounds, compared with the exact results by the brute-forth method,. To make the experiments tractable, we study the relative approximation ratio. We use the result of **OSScaling** with $\epsilon=0.1$ (which has the smallest approximation ratio in the proposed methods) as the base and compare the relative performance of the other algorithms with it. We compute the *relative ratio* of an algorithm over **OSScaling** with $\epsilon=0.1$ as follows: For each query, we compute the ratio of the objective score of the route found by the algorithm to the score of the route found by **OSScaling** with $\epsilon=0.1$, and the average ratio over all queries is finally reported as the measure.

With the measure, we study the effect of the following parameters on accuracy, namely the number of query keywords, the budget limit Δ , the scaling parameter ϵ in **OSScaling**, the specified parameter β in **BucketBound**, and the parameter α which balances the importance of the objective and budget scores during the node selection, for **Greedy**.

Varying the number of query keywords or Δ . Figure 5.10 shows the relative ratio compared with the results of **OSScaling** with $\epsilon=0.1$ for the experiment in Figure 5.4, in which we vary the number of query keywords. Figure 5.11 shows the relative ratio for the experiment in Figure 5.5, in which we vary the value of budget limit Δ , respectively. Note that $\epsilon=0.5$ and $\beta=1.2$ in the two experiments.

Figure 5.10: Relative ratio (Flickr)

Figure 5.11: Relative ratio (Flickr)

Since the greedy algorithms fail to find a feasible solution on about 10%–20% queries, for greedy algorithms we measure the relative ratio only on the queries where **Greedy-1** and **Greedy-2** are able to find feasible routes. For **OSScaling** and **BucketBound**, the reported results are based on all queries, which are similar to the results if we only use the set of queries for which **Greedy** returns feasible solutions. We observe that the relative ratio of **BucketBound** compared with the results of **OSScaling** is always below the specified parameter β . It can also be observed that **BucketBound** can achieve much better accuracy than do **Greedy-1** and **Greedy-2**, especially when the number of query keywords or the value of Δ is large.

Varying the parameter ϵ for OSScaling. Figure 5.7 shows the effect of ϵ on the relative ratio in **OSScaling**. We set Δ as 6 kilometers and the number of query keywords at 6. We can observe that the relative ratio becomes worse as we increase ϵ , which is consistent with the result of Theorem 5.12, *i.e.*, the performance bound of **OSScaling** is $\frac{1}{1-\epsilon}$.

Varying the parameter β for BucketBound. Figure 5.9 shows the effect of β on the relative ratio in **BucketBound**, while the corresponding runtime is reported in Figure 5.8, where we set $\epsilon=0.5$, $\Delta=6$ km, and the number of query keywords as 6. As expected, the relative ratio becomes worse as we increase β . Note that relative ratio of **BucketBound** compared to the results of **OSScaling** is consistently smaller than the specified β .

Varying the parameter α for Greedy. Figure 5.12 shows the relative ratio of **Greedy-1** and **Greedy-2** compared with the results of **OSScaling** when we vary α , and Figure 5.13 shows the percentage of failed queries. In this set of experiments, we set Δ as 6 kilometers, and the average performance is reported over 5 runs, each with a different number of query keywords from 2 to 10. Note that the relative ratio is computed based on the set of queries where **Greedy-1** and **Greedy-2** are able to find feasible routes over the set of queries with feasible solutions (**OSScaling** and **BucketBound** guarantee to return feasible results if any). We observe that as the value of α increases the relative ratio becomes

Figure 5.12: Relative ratio (Flickr)

Figure 5.13: Failure percentage (Flickr)

worse for both Greedy-1 and Greedy-2, but they succeed in finding feasible routes for more queries. When α is set as 0, which means that the objective score is the only criterion when selecting the node in each step of Greedy, both Greedy-1 and Greedy-2 achieve the best average ratio while the failure percentage is the largest. When $\alpha=1$, the next best node is selected merely based on the budget score. Hence, Greedy is able to find feasible results on more queries, but the relative accuracy becomes much worse on the queries for which Greedy is able to return feasible solutions. Greedy-2 outperforms Greedy-1 consistently, because more routes are checked in Greedy and it is likely to find more feasible and better routes.

5.3.2.3 Comparing OSScaling and BucketBound

Figure 5.14: Runtime (Flickr)

Figure 5.15: Relative ratio (Flickr)

The aim of this set of experiment is to compare the performance of OSScaling and BucketBound when they have the same theoretical approximation ratio. In this set of experiments, $\Delta=6$ km, $\beta=1.2$, and the number of query keywords is 6. The values of ϵ are

computed according to different performance bounds for both algorithms. Figures 5.14 and 5.15 show the runtime and relative ratio of **OSScaling** and **BucketBound** when we vary the performance bound, respectively. We observe that **BucketBound** runs consistently faster than **OSScaling** over all performance bounds while **OSScaling** always achieves better relative ratio.

5.3.2.4 Performance of Algorithms for KkR

Figure 5.16: Runtime (KkR)

Figure 5.17: Scalability

We study the performance of the modified versions of the two approximation algorithms, *i.e.*, **OSScaling** and **BucketBound** for processing KkR . We set $\epsilon=0.5$, $\beta=1.2$, $\Delta=6$ km, and the average runtime is reported over 5 runs, each with a different number of query keywords from 2 to 10. The results are shown in Figure 5.16. **BucketBound** always outperforms **OSScaling** in terms of runtime. As expected, both algorithms run slower as we increase the value of k . In **OSScaling**, more labels need to be generated for larger k , which leads to longer runtime. Algorithm **BucketBound** terminates only after the top- k feasible routes are found, thus needing longer query time.

5.3.2.5 Experiments on More Datasets

We also conduct experiments on the synthetic dataset containing 5,000 nodes. Figure 5.18 and 5.19 show the runtime when we vary the number of query keywords and the value of Δ , respectively. We set ϵ as 0.5 and β as 1.2. The comparison results are consistent with those on the Flickr dataset.

5.3.2.6 Pre-processing Costs

We study the time and space costs of the pre-processing presented in section 5.2.1. We use the Floyd-Warshall algorithm, a well known all pair shortest paths finding algorithm,

Figure 5.18: Runtime (Synthetic)

Figure 5.19: Runtime (Synthetic)

to find two paths for each pair of nodes (v_i, v_j) : the path with the smallest objective score and its budget score, and the path with the smallest budget score and its objective score. The time complexity of this algorithm is $O(|V|^3)$, and the space complexity of this algorithm is $O(|V|^2)$. On both real and synthetic dataset containing 5,000 nodes, the pre-processing time is about 20 minutes, and the pre-processing results occupies around 600 Mb in the disk.

5.3.2.7 Scalability

Figure 5.17 shows the runtime of the proposed algorithms (the number of query keywords is 6 and $\Delta=30$ km). They all scale well with the size of the dataset. The relative ratio changes only slightly and thus is not reported.

5.3.2.8 Real Example

Figure 5.20: Example route 1

Figure 5.21: Example route 2

We use one example found in the Flickr dataset to show that KOR is able to find routes according to users' various preferences. We set the starting location at the Dewitt Clinton

park and the destination at United Nations Headquarters, and the query keywords are “jazz,” “imax,” “vegetarian,” and “Cappuccino,” *i.e.*, a user would like to find a route such that he can listen to jazz music, watch a movie, eat vegetarian food and have a cup of Cappuccino. When we set the distance threshold Δ as 9 km, the route shown in Figure 5.20 is returned by **OSScaling** as the most popular route that covers all query keywords and satisfies distance threshold. We find that according to the historical trips, this route has the most visitors among all routes covering all the query keywords shorter than 9 km. However, when Δ is set as 6 km, the route shown in Figure 5.21 is returned. This route has the most visitors among all feasible routes given $\Delta=6$ km. In the case, the route in Figure 5.20 exceeds the limit $\Delta=6$ km and is pruned during the execution of **OSScaling** algorithm.

5.3.2.9 Summarization of Empirical Study

From the experiment results, we can conclude as follows: the algorithm **Greedy** runs the fastest, but it cannot guarantee a feasible result and its accuracy is the worst. The algorithm **BucketBound** runs faster than **OSScaling**, because it can terminate early if a certain condition is met, but its accuracy is worse than that of **OSScaling**. As the number of query keywords and the distance threshold increase, all algorithms perform worse in terms of query processing time.

5.4 Additional Related Work

Travel route search: The travel route search problem has received a lot of attention. Li et al. [69] propose a new query called Trip Planning Query (TPQ) in spatial databases, in which each spatial object has a location and a category, and the objects are indexed by an R-tree. A TPQ has three components: a start location s , an end location t , and a set of categories C , and it is to find the shortest route that starts at s , passes through at least one object from each category in C and ends at t . It is shown that TPQ can be reduced from the Traveling Salesman problem, which is NP-hard. Based on the triangle inequality property of metric space, two approximation algorithms including a greedy algorithm and an integer programming algorithm are proposed. Compared with TPQ, KOR studied in this chapter includes an additional constraint (the budget constraint), and thus is more expressive. The algorithms in the work [69] cannot be used to process KOR.

Sharifzadeh et al. [94] study a variant problem of TPQ [69], called optimal sequenced route query (OSR). In OSR, a total order on the categories C is imposed and only the starting location s is specified. The authors propose two elegant exact algorithms L-LORD and R-LORD. Under the same setting [69] that objects are stored in spatial

databases and indexed by an R-tree, metric space based pruning strategies are developed in the two exact algorithms.

Chen et al. [21] considers the multi-rule partial sequenced route (MRPSR) query, which is a unified query of TPQ and OSR. Three heuristic algorithms are proposed to answer MRPSR queries. Li et al. [71] also studies the same problem as defined in the work [21]. They devise novel and efficient solutions based on two methodologies: backward search and forward search. The former computes the optimal route from the last point to the first, while the latter follows the first-to-last order of points. KOR is different from OSR and MRPSR and their algorithms are not applicable to process KOR.

Kanza et al. [62] consider a different route search query on the spatial database: the length of the route should be smaller than a specified threshold while the total text relevance of this route is maximized. Greedy algorithm is proposed without guaranteeing to find a feasible route. Their subsequent work [60] develops several heuristic algorithms for answering a similar query in an interactive way. After visiting each object, the user provides feedback on whether the object satisfies the query, and the feedback is considered when computing the next object to be visited. In the work [68], approximate algorithms for solving OSR [94] in the presence of order constraints in an interactive way are developed. Kanza et al. also study the problem of searching optimal sequenced route in probabilistic spatial database [61]. Lu et al. [75] consider the same query [62] and propose a data mining-based approach. The queries considered in these works are different from KOR and these algorithms cannot be used to answer KOR.

Malviya et al. [78] tackle the problem of answering continuous route planning queries over a road network. The route planning [78] aims to find the shortest path in the presence of updates to the delay estimates. Roy et al. [90] consider the problem of interactive trip planning, in which the users give feedbacks for the already suggested points-of-interests, and the itineraries are constructed iteratively based on the users' preferences and time budget. Obviously, these two problems are different with KOR.

Yao et al. [108] propose the multi-approximate-keyword routing (MARK) query. A MARK query is specified by a starting and an ending location, and a set of (*keyword*, *threshold*) value pairs. It searches for the route with the shortest length such that it covers at least one matching object per keyword with the similarity larger than the corresponding threshold value. Obviously, MARK has different aims with that of the KOR query.

Travel route recommendation: Recent works on travel route recommendation aim to recommend routes to users based on users' travel histories. Lu et al. [77] collect geo-tagged photos from Flickr and build travel routes from them. They define popularity scores on each location and each trip, and recommend a route that has the largest popularity

score within a travel duration in the whole dataset for a city. The recommendation in this work is not formulated as queries and the recommendation algorithm runs in an extreme long time. The work [25] finds popular routes from users' historical trajectories. The popularity score is defined as the probability from the source location to the target location estimated using the absorbing Markov model based on the trajectories. Yoon et al. [109] propose a smart recommendation, based on multiple user-generated GPS trajectories, to efficiently find itineraries. The work [65] predicts the subsequent routes according to the user's current trajectory and previous trajectory history. None of these proposals takes into account the keywords as we do in this work.

5.5 Conclusion

We define the problem of keyword-aware optimal route query, denoted by **KOR**, which is to find an optimal route such that it covers a set of user-specified keywords, a specified budget constraint is satisfied, and the objective score of the route is optimized. The problem of answering **KOR** queries is NP-hard. We devise two approximation algorithms, *i.e.*, **OSScaling** and **BucketBound** with provable approximation bounds for this problem. We also design a greedy approximation algorithm. Results of empirical studies show that all the proposed algorithms are capable of answering **KOR** queries efficiently, while the algorithms **BucketBound** and **Greedy** run faster. We also study the accuracy of approximation algorithms.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Most existing proposals on spatial keyword querying treat the geo-textual objects independently. This can be interpreted from two aspects: first, the aim is to search for single geo-textual objects each of which satisfies the query independently; second, during the query processing, the geo-textual objects are processed independently and one object does not affect the relevance between another object and the query. Therefore, real user needs may not be satisfied. This thesis describes three different approaches to answering spatial keyword queries beyond the single object granularity.

Chapter 3 proposes the top- k prestige-based spatial keyword (k PSK) query, which retrieves single geo-textual objects, taking into account the inter-object relationship. The prestige is first computed by the text relevance between the locations and the query, and then it propagates among the locations. This means that the prestige involves mutual reinforcement, *i.e.*, the prestige of one location is affected by the prestige of its neighbor locations. The propagation between two locations is determined by both the spatial proximity and the text relevance between them. The intuition is that if two locations are close and are also textually relevant to each other, they are more likely to affect each other. A k PSK query takes into account both the prestige score and the location proximity when ranking the geo-textual objects. We devise two algorithms to process the query: ES-EBC that incorporates the bookmark-coloring algorithm [10] and the threshold algorithm [39], and S-EBC that utilizes the graph partition and decomposes the prestige scoring over the whole graph into sub-graphs.

Chapter 4 proposes the spatial group keyword (SGK) query, which searches for a group of geo-textual objects such that they collectively cover all the query keywords and the cost computed by a certain function is minimized. The cost function takes into account both the distance between the query and the group and the inter-object distance among

the objects in the group. We study two instances of this problem, both of which are NP-hard. The first type of the **SGK** query computes the aggregated distances of the group to the query as the cost. The cost of the second type of the **SGK** query is computed by the sum of the maximum distance between the query and any object in the group and the maximum distance between any pair of objects in the group. We design both exact algorithms and approximation algorithms with performance guarantees for them.

Chapter 5 proposes the keyword-aware optimal route (**KOR**) query, which retrieves the route such that it contains all the query keywords, its budget score satisfies a certain constraint, and its objective score is minimized. In the **KOR** query, the geo-textual objects are connected by road segments, and a sequence of them form a route to answer the geo-textual query. Since this problem is NP-hard, we propose two approximation algorithms with provable approximation ratios and a heuristic approach. The first approximation algorithm scales the objective score of the routes into integers, and then we extend the routes from the source location to the target location. On each object, if two routes cover the same set of query keywords and have the same scaled objective score, we only keep the one with smaller budget score. The second approximation algorithm further improve the first algorithm, and the approximation bound can be specified by users. We still extend the routes from source to target. However, it can early terminates once a route is found that satisfies the pre-defined bound utilizing some techniques. The heuristic algorithm greedily extends the routes from source to target considering both the budget and objective scores.

All of these approaches can efficiently and effectively answer spatial keyword queries beyond the single object granularity.

6.2 Future Work

6.2.1 Extension of *kPSK* Query

There are two main issues need to be addressed in the future work for the *kPSK* query.

First, it is necessary to reduce the size of the pre-processing results used for solving the *kPSK* query. Because each border node has a PR vector, which stores the prestige distribution after propagating a unit PR score from this border node to the whole graph, the space cost is $O(|V| \cdot |H|)$ (where $|V|$ is the number of locations and $|H|$ is the number of hub nodes). If there are many hub nodes, the pre-processing costs large space, and the query processing also takes a long time since many PR vectors need to be read from disk. There could be two possible ways to solve this problem. One solution is to minimize the number of border nodes when partitioning the location graph into subgraphs of almost equal size. However, this problem is NP-hard, and thus only some approximate approaches can be utilized [3]. Another method is to first partition the location graph

into a hierarchy, and then make use of this hierarchy to compute PR vectors. The PR vector of a border node only keeps the distribution of a unit PR score to the subgraph in the hierarchy it belongs to, rather than to the whole graph. Then, the hierarchical PR vectors are used to compute the PR scores during query time.

Second, if a new location is inserted into or deleted from the dataset, all the pre-computed PR vectors need to be re-computed. Hence, we need to find an efficient way to handle frequent updates. Exact PR vectors are not possible in such case, since either inserting or deleting a location renders all existing exact PR vectors invalid. We need an approximate method to accommodate the frequent updates.

Third, it is of interests to diversify the results of the *kPSK* query. Result diversification can improve the quality of search results [1, 37, 118]. How to diversify the spatial keyword search on road networks has been studied by Zhang et al. [112]. The *kPSK* query can be viewed as retrieving “centers” of regions with many relevant objects. According to the experimental results, the top-*k* objects of a query do not locate in the same area. However, we can incorporate the concept of “diversification” into the *kPSK* to make sure that the top-*k* results are not close to each other.

6.2.2 Extension of SGK Query

In the current work, we only consider two types of SGK query. There could be various types of SGK queries according to different requirements. For example, the cost can be computed by the sum of the minimum distance between the query and any object in the group and the diameter of this group. Hence, it is of interests to think over more types of SGK queries and design both exact and approximate algorithms for them.

Second, it is necessary to extend the algorithms to find top-*k* groups, which can provide users more options. An open problem here is to what extent we should allow the overlap among top-*k* groups. It seems to be reasonable to allow a certain degree of overlap. However, it might not be useful to return groups sharing too many locations. One possible way is to define some functions to compute the diversity of the top-*k* groups, and we can try to maximize the diversity score of top-*k* groups.

Second, treating all query keywords equally may not be suitable in some application scenarios. Thus, we can consider an information retrieval ranking model (such as the vector space model) to compute the text relevance for a result group, and the ranking should consider both the text relevance and the cost of a group.

6.2.3 Extension of KOR Query

The approximation algorithms highly rely on the pre-processing results. The current pre-processing method computes the smallest objective score and the smallest budget score

between each pair of nodes, and thus the time complexity is $O(|V|^3)$ and the space cost is $O(|V|^2)$ ($|V|$ is the number of nodes in the graph). Hence, it is necessary to improve the current pre-processing approach. We can employ a graph partition algorithm to divide a large graph into several subgraphs. Next, we only do the pre-processing within each subgraph instead of on the whole graph. We also compute and store the best objective and budget score between every pair of border nodes. Thus, the path with the best objective or budget score can be obtained. We believe that this approach can greatly reduce the time and space costs of pre-processing.

In KOR queries, the keywords are treated equally, which may not be suitable in real applications. For example, terms such as “hotel” can reveal what a location is, and terms such as “five-star” indicate the quality of a location. Hence, it is better to distinguish the keywords according to the extent how they can represent locations. The query should allow users to first specify the “major” keywords such as “hotels” or “restaurants”, which can be viewed as the categories of locations. Then, for each category the detailed requirements can be specified. For example, it must be a “five-star” hotel, or the restaurant must serve “Japanese food”. A candidate route should be formed by a sequence of locations satisfying the detailed requirements from each category.

6.2.4 Other Approaches to Spatial Keyword Querying beyond the Single Object Granularity

In k PSK queries, we consider the common spatial phenomenon that geo-textual objects with similar textual descriptions tend to cluster. This is called the “cluster effect”¹. For example, from a real dataset of places, we can identify a “restaurant” area, a “pub” area, etc. A region is expected to better satisfy the user needs than does a single object. For example, an area with several restaurants can attract much more customers comparing to a single isolated restaurant.

This inspires the spatial keyword region query. Specifically, we can consider the following three attributes of a region given a query, *i.e.*, the distance of the region to the query location, the text relevance of the objects in the region to the query texts, and the density of this region. For example, if a person wants to go to the town center and buy Christmas presents, she is perhaps not looking for a specific store, but rather a shopping region. Hence, queries for single stores are not meaningful in this case. A dense region that contains many shops and close to the town center should be returned to the her.

By taking into account the prestige the k PSK query addresses the cluster effect, but it still returns individual objects. The SGK query returns a group of objects, but it aims to find a group of objects covering all the query keywords with the optimal cost. The

¹http://en.wikipedia.org/wiki/Cluster_effect

aim is different with that of the spatial keyword region query that searches for the best region considering the distance to the query location, the total text relevance to the query keywords, and the density of a region.

List of Publications

- Lisi Chen, Gao Cong, **Xin Cao**. An efficient query indexing mechanism for filtering geo-textual data. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD 2013**), pages 749-760, New York, NY, USA, June 22-27, 2013.
- **Xin Cao**, Lisi Chen, Gao Cong, Jihong Guan, Nhan-Tue Phan, Xiaokui Xiao. KORR: Keyword-aware Optimal Route Search System (**Demo**). 29th IEEE International Conference on Data Engineering (**ICDE 2013**), pages 1340-1343, Brisbane, Australia, April 8-12, 2013.
- **Xin Cao**, Lisi Chen, Gao Cong, Xiaokui Xiao. Keyword-aware Optimal Route Search. Proceedings of the VLDB Endowment (**PVLDB**), Volume 5, Issue 11, pages 1136-1147, 2012.
- **Xin Cao**, Gao Cong, Christian S. Jensen, Jun Jie Ng, Beng Chin Ooi, Nhan-Tue Phan, Dingming Wu. SWORS: A System for the Efficient Retrieval of Relevant Spatial Web Objects (**Demo**). Proceedings of the VLDB Endowment (**PVLDB**), Volume 5, Issue 12, pages 1914-1917, 2012.
- **Xin Cao**, Gao Cong, Bin Cui, Christian S. Jensen, Quan Yuan. Approaches to Exploring Category Information for Question Retrieval in Community Question-Answer Archives. ACM Transactions on Information Systems (**TOIS**), Volume 30, Issue 2, pages 7:1-7:38, 2012.
- **Xin Cao**, Lisi Chen, Gao Cong, Christian S. Jensen, Qiang Qu, Anders Skovsgaard, Dingming Wu, Man Lung Yiu. Spatial Keyword Querying. 31st International Conference on Conceptual Modeling (**ER 2012**), pages 16-29, Florence, Italy, October 15-18, 2012.
- **Xin Cao**, Gao Cong, Christian S. Jensen, Beng Chin Ooi. Collective spatial keyword querying. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD 2011**), pages 373-384, Athens, Greece, June 12-16, 2011.

- Hua Lu, **Xin Cao**, Christian S. Jensen. A Foundation for Efficient Indoor Distance-Aware Query Processing. IEEE 28th International Conference on Data Engineering (**ICDE 2012**), pages 438-449, Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012.
- **Xin Cao**, Gao Cong, Christian S. Jensen. Retrieving Top-k Prestige-Based Relevant Spatial Web Objects. Proceedings of the VLDB Endowment (**PVLDB**), Volume 3, Issue 1, pages 373-384, 2010.
- **Xin Cao**, Gao Cong, Christian S. Jensen. Mining Significant Semantic Locations From GPS Data. Proceedings of the VLDB Endowment (**PVLDB**), Volume 3, Issue 1, pages 1009-1020, 2010.

Bibliography

- [1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.
- [2] E. Amitay, N. Har’El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *SIGIR*, pages 273–280, 2004.
- [3] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.
- [4] E. M. Arkin and R. Hassin. Minimum-diameter covering problems. *Networks*, 36(3):147–155, 2000.
- [5] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.
- [6] Z. Bar-Yossef and L.-T. Mashiach. Local approximation of PageRank and reverse PageRank. In *CIKM*, pages 279–288, 2008.
- [7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.
- [8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
- [9] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.
- [10] P. Berkhin. Bookmark-coloring algorithm for personalized PageRank computing. *Internet Math.*, 3(1):41–62, 2006.

- [11] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.
- [12] N. Bourgeois, B. Escoffier, and V. T. Paschos. Efficient approximation of min set cover by moderately exponential algorithms. *Theor. Comput. Sci.*, 410(21-23):2184–2195, 2009.
- [13] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.
- [14] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.
- [15] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *ER*, pages 16–29, 2012.
- [16] X. Cao, G. Cong, B. Cui, and C. S. Jensen. A generalized framework of exploring category information for question retrieval in community question answer archives. In *WWW*, pages 201–210, 2010.
- [17] X. Cao, G. Cong, B. Cui, C. S. Jensen, and C. Zhang. The use of categorization information in language models for question retrieval. In *CIKM*, pages 265–274, 2009.
- [18] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *SSDBM*, pages 87–95, 2010.
- [19] S. Chakrabarti. Dynamic personalized PageRank in entity-relation graphs. In *www*, pages 571–580, 2007.
- [20] W. W. Chang and H.-J. Schek. A signature access method for the starburst database system. In *VLDB*, pages 145–153, 1989.
- [21] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *GIS*, pages 10:1–10:10, 2008.
- [22] L. Chen, G. Cong, and X. Cao. An efficient query indexing mechanism for filtering geo-textual data. In *SIGMOD Conference*, pages 749–760, 2013.

- [23] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *PVLDB*, pages 217–228, 2013.
- [24] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.
- [25] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.
- [26] A. G. Chentsov and L. N. Korotayeva. The dynamic programming method in the generalized traveling salesman problem. *Mathematical and Computer Modelling*, 25(1):93–105, 1997.
- [27] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.
- [28] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.
- [29] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
- [30] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
- [31] G. Cong, H. Lu, B. C. Ooi, D. Zhang, and M. Zhang. Efficient spatial keyword search in trajectory databases. *CoRR*, abs/1205.2880, 2012.
- [32] G. Cong, L. Wang, C.-Y. Lin, Y.-I. Song, and Y. Sun. Finding question-answer pairs from online forums. In *SIGIR*, pages 467–474, 2008.
- [33] M. Cygan, L. Kowalik, and M. Wykurz. Exponential-time approximation of weighted set cover. *Inf. Process. Lett.*, 109(16):957–961, 2009.
- [34] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
- [35] M. Desrochers and F. Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *Information Systems Research*, 26(1):191–212, 1988.

- [36] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In *VLDB*, pages 545–556, 2000.
- [37] M. Drosou and E. Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *PVLDB*, 6(1):13–24, 2012.
- [38] I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42:135–153, 2003.
- [39] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
- [40] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.
- [41] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *PODS*, pages 247–252, 1989.
- [42] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. SEAL: Spatio-textual similarity search. *PVLDB*, 5(9):824–835, 2012.
- [43] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.
- [44] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [45] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized PageRank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3), 2005.
- [46] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1st edition, 1979.
- [47] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

- [48] Google. *Google Places: Stats & Facts*. <http://sites.google.com/a/pressatgoogle.com/googleplaces/metrics>.
- [49] M. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.
- [50] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.
- [51] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, page 16, 2007.
- [52] A. Henry-Labordere. The record balancing problem - a dynamic programming solution of a generalized traveling salesman problem. *Revue Francaise D Informatique De Recherche Operationnelle*, B(2):43–49, 1969.
- [53] D. Hilbert. Über die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.
- [54] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.
- [55] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.
- [56] G. Jeh and J. Widom. Scaling personalized web search. In *www*, pages 271–279, 2003.
- [57] C. B. Jones, A. I. Abdelmoty, D. Finch, G. Fu, and S. Vaid. The SPIRIT spatial search engine: Architecture, ontologies and spatial indexing. In *GIScience*, pages 125–139, 2004.
- [58] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford University, 2003.

- [59] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *SIGMOD Conference*, pages 546–557, 2002.
- [60] Y. Kanza, R. Levin, E. Safra, and Y. Sagiv. An interactive approach to route search. In *GIS*, pages 408–411, 2009.
- [61] Y. Kanza, E. Safra, and Y. Sagiv. Route search over probabilistic geospatial data. In *SSTD*, pages 153–170, 2009.
- [62] Y. Kanza, E. Safra, Y. Sagiv, and Y. Doytsher. Heuristic algorithms for route-search queries over geographical data. In *GIS*, pages 1–10, 2008.
- [63] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA*, pages 450–466, 2010.
- [64] A. Khodaei, C. Shahabi, and C. Li. SKIF-P: a point-based indexing and ranking of web documents for spatial-keyword search. *GeoInformatica*, 16(3):563–596, 2012.
- [65] T. Kurashima, T. Iwata, G. Irie, and K. Fujimura. Travel route recommendation using geotags in photo sharing sites. In *CIKM*, pages 579–588, 2010.
- [66] O. Kurland and L. Lee. PageRank without hyperlinks: structural re-ranking using links induced by language models. In *SIGIR*, pages 306–313, 2005.
- [67] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.
- [68] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv. Interactive route search in the presence of order constraints. *PVLDB*, 3(1):117–128, 2010.
- [69] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.
- [70] G. Li, J. Feng, and J. Xu. DESKS: Direction-aware spatial keyword search. In *ICDE*, pages 474–485, 2012.
- [71] J. Li, Y. D. Yang, and N. Mamoulis. Optimal route queries with arbitrary order constraints. *IEEE Trans. Knowl. Data Eng.*, 25(5):1097–1110, 2013.

- [72] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.
- [73] S. Liu, G. Li, and J. Feng. Star-join: spatio-textual similarity join. In *CIKM*, pages 2194–2198, 2012.
- [74] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In *SIGMOD*, pages 689–700, 2013.
- [75] E. H.-C. Lu, C.-Y. Lin, and V. S. Tseng. Trip-Mine: An efficient trip planning approach with travel time constraints. In *MDM*, pages 152–161, 2011.
- [76] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *SIGMOD*, pages 349–360, 2011.
- [77] X. Lu, C. Wang, J.-M. Yang, Y. Pang, and L. Zhang. Photo2Trip: generating travel routes from geo-tagged photos for trip planning. In *MM*, pages 143–152, 2010.
- [78] N. Malviya, S. Madden, and A. Bhattacharya. A continuous query system for dynamic route planning. In *ICDE*, pages 792–803, 2011.
- [79] B. Martins, M. J. Silva, and L. A. Ribeiro. Indexing and ranking in Geo-IR systems. In *GIR*, pages 31–34, 2005.
- [80] K. S. McCurley. Geospatial mapping and navigation of the web. In *WWW*, pages 221–229, 2001.
- [81] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., 1966.
- [82] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.
- [83] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.
- [84] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

- [85] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases - with applications to GIS*. Elsevier, 2002.
- [86] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *TREC*, pages 109–126, 1994.
- [87] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørnvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.
- [88] J. B. Rocha-Junior and K. Nørnvåg. Top-k spatial keyword queries on road networks. In *EDBT*, pages 168–179, 2012.
- [89] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.
- [90] S. B. Roy, G. Das, S. Amer-Yahia, and C. Yu. Interactive itinerary planning. In *ICDE*, pages 15–26, 2011.
- [91] SearchEngineLand. *Microsoft: 53 Percent Of Mobile Searches Have Local Intent*. <http://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556>.
- [92] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.
- [93] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.
- [94] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.
- [95] M. J. Silva, B. Martins, M. S. Chaves, A. P. Afonso, and N. Cardoso. Adding geographic scopes to web resources. *Computers, Environment and Urban Systems*, 30(4):378–399, 2006.
- [96] H.-W. Six and P. Widmayer. Spatial searching in geometric databases. In *ICDE*, pages 496–503, 1988.
- [97] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, 174(1):38–53, 2006.

- [98] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [99] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.
- [100] G. Vic, M. Campbell, and M. Campbell. The design of text signatures for text retrieval systems. Technical report, Deakin University, 1994.
- [101] D. Wu, G. Cong, and C. S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.
- [102] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.
- [103] D. Wu, M. L. Yiu, and C. S. Jensen. Moving spatial keyword queries: Formulation, methods, and analysis. *ACM Trans. Database Syst.*, 38(1):7, 2013.
- [104] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, pages 541–552, 2011.
- [105] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.
- [106] X. Xiao, Q. Luo, Z. Li, X. Xie, and W.-Y. Ma. A large-scale study on map search logs. *TWEB*, 4(3):1–33, 2010.
- [107] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On socio-spatial group query for location-based social networks. In *KDD*, pages 949–957, 2012.
- [108] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in GIS data. In *GIS*, pages 201–210, 2011.
- [109] H. Yoon, Y. Zheng, X. Xie, and W. Woo. Smart itinerary recommendation based on user-generated GPS trajectories. In *UIC*, pages 19–34, 2010.
- [110] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.
- [111] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top-k spatial keyword search. In *ICDE*, pages 901–912, 2013.

- [112] C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang. Diversified spatial keyword search on road networks. In *EDBT*, pages 367–378, 2014.
- [113] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.
- [114] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in Web 2.0. In *ICDE*, pages 521–532, 2010.
- [115] D. Zhang, K.-L. Tan, and A. K. H. Tung. Scalable top-k spatial keyword search. In *EDBT*, pages 359–370, 2013.
- [116] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. Towards efficient search for activity trajectories. In *ICDE*, pages 230–241, 2013.
- [117] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.
- [118] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.
- [119] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.
- [120] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.