Efficient Location-Based Spatial Keyword Query Processing

ZHANG DONGXIANG

Bachelor of Computer Science

Fudan University, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2012

ACKNOWLEDGEMENT

First and foremost, I would like to express my deepest gratitude to my advisor Professor Anthony K. H. Tung. He welcomed me on board when I was still a fresh and shy graduate student. During the entire period of my doctoral study, Professor Tung has provided me with independent research skill, including how to find new and interesting problems, how to write a good research paper and how to organize related works in a coherent manner.

Professor Beng Chin Ooi, my project supervisor, also played an essential role in my research as well as in my life. His strictness has contributed to my growth as a rigorous research. As a system expert, he has helped improve my ability and skills in building systems tremendously. I am greatly impressed by his academic vigor as well as his personalities including diligence, high self motivation and concern with those around him.

I also would like to thank the members of my thesis committee, Professor Kian-Lee Tan and Professor Roger Zimmermann for their valuable reviews, comments and suggestions to improve the quality of the thesis. I appreciate the efforts from all the professors coauthoring with me, including Masaru Kitsuregawa, Divyakant

Agrawal, Gang Chen, Yeow Meng Chee and Anirban Mondal. In addition, I would like to thank my English lecturer Professor Xudong Deng for his passion and efforts in editing my drafts.

Many friends in Singapore have also helped me a lot during my Ph.D pursuit. First, my best friends Huanhuan Lu, Xiangyu Wang and Zhi Zhong came to Singapore with me. We lived together, encouraged each other and had great fun in the past 4 years. I also received useful advice from many senior fellow members and spent joyful time with them. They are Su Chen, Yueguo Chen, Bingtian Dai, Difeng Dong, Shuqiao Guo, Dong Guo, Hao Li, Yingyi Qi, Xianju Wang, Nan Wang, Ji Wu, Sai Wu, Linhao Xu, Ning Ye, Zhenjie Zhang and Shaojie Zhuo. I also would like to express my appreciation to my lab colleagues and basketball team members as we shared a wonderful experience together.

Last but not least, I would like to thank all of my big family: my parents Sunqing Zhang and Xiujie Zhang, my sisters Lizhi Zhang and Yanqing Zhang and my younger brother Dongxu Zhang for their unconditional support and encouragement. I wish my grandmother in heaven would be proud of my achievements.

The most special thanks are reserved for my dearest Yuan Wang for her company and love which has sustained me through the otherwise grueling period of my doctoral study.

ABSTRACT

The emergence of Web 2.0 applications, including social networking sites, wikipedia and multimedia sharing sites, has changed the way of how information is generated and shared. Among these applications, map mashup is a popular and convenient means for data integration and visualization. In recent years, users have contributed a huge amount of spatial objects in various media formats and displayed them on a map. They have also annotated these objects with tags to provide semantic meaning. In order to leverage such a large scale spatial-textual database, we propose efficient location-based spatial keyword query processing strategies in this thesis.

First, we address a novel query, named mCK (m Closest Keywords). The query accepts a set of query keywords and aims at finding a set of spatial tuples matching the keywords and closest to each other. A useful application is to find m closest local service providers using keywords such as "cinema", "seafood restaurant" and "shopping mall", to save the transportation time. To efficiently answer an mCK query, we introduce a new index named bR*-tree which is an extension of R*-tree. Based on bR*-tree, we exploit a priori-based top-down search strategy and propose efficient pruning rules which significantly reduce the search space.

Second, we adopt mCK query to detect the geographical context of web resources. More specifically, we build a uniform model to represent online resources by a set of tags and propose a detection method by tag matching. Since there could be hundreds of thousands of tags, we improve bR*-tree and design an efficient and scalable search algorithm. Furthermore, we propose a new *geo-tf-idf* ranking method to improve the matching precision.

Third, we solve the problem of efficient web image locating when tags are not available. We treat high dimensional image feature as "keyword". Thus, a geo-image can be considered as a set of spatial keywords at the same location. Given a query image, our goal is to find a geo-image in the spatial image database that is most similar to the query image and use its location as the detecting result. To solve the nearest neighbor (NN) query, we propose a new index named HashFile. The index can support approximate NN search in the Euclidean space and exact NN search in L_1 norm. Our experiment results show that it provides better efficiency in processing both types of NN queries.

Finally, we design and develop a new travel mashup system, named **LANGG**, to utilize the above efficient spatial keyword query processing technique and provide location-based services. The main objective of our system is to recommend users a travel destination based on their personal interest. Users can submit a set of travel services they would like to enjoy, an interesting travel blog or even a travel photo with beautiful scene. User feedback shows that our system provides satisfactory search results.

CONTENTS

Acknowledgement **ii**

Abstract **iv**

1 Introduction **1**

1.1 Travel Map Mashup Applications In Web 2.0 2

1.2 Locating m Closest Keywords In a Spatial Database 4

1.3 Locating Web Resources by Spatial Tag Matching 6

1.4 Locating Landmark Photos by Content-Based Matching 11

1.5 LANGG : A Location-Based Travel Mashup System 12

1.6 Contribution of the Thesis 13

1.7 Thesis Organization 14

2 Literature Review **16**

2.1 Finding m-Closest Keywords in Spatial Databases 16

2.2 Locating Web Documents 19

2.3 Landmark Recognition 20

2.3.1 High Dimensional Index for Exact NN Query 21

2.3.2 LSH for Approximate NN Query 22

3 Locating Closest Travel Services 24

3.1 Introduction 25

3.2 bR*-tree: R*-tree With Bitmaps and Keyword MBRs 29

3.3 Search Algorithms 32

3.3.1 Searching In One Node 34

3.3.2 Searching In Multiple Nodes 39

3.3.3 Pruning via Distance Mutex 43

3.3.4 Pruning via Keyword Mutex 45

3.4 Empirical Study 48

3.4.1 Experiments on Synthetic Data Sets 49

3.4.2 Experiments on Real Data Set 55

3.5 Summary 57

4 Locating Web Resources By Spatial Tag Matching 58

4.1 Introduction 59

4.2 Spatial Index and Search Algorithm 62

4.2.1 Light-weight Index Structure 62

4.2.2 Bottom-Up Search Algorithm 65

4.3 Ranking 67

4.3.1 Approximate Ranking Mechanism 70

4.4 Experiment Study 72

4.4.1 Experiments on Synthetic Data Sets 72

4.4.2 Experiments On Real Data Sets 76

4.5 Summary 85

5 Landmark Recognition Using HashFile 86

5.1 Introduction 87

5.2 The Preliminaries 91

5.2.1 Random Projection 91

5.2.2 Distance Constraint for Exact NN Query Using L_1 93

5.3 HashFile Index Structure 98

5.3.1 HashFile Overview 98

5.3.2 Data Insertion 100

5.3.3 Data Deletion 102

5.3.4 Data Update 103

5.4 Exact NN Query Processing 103

5.5 Approximate NN Query Processing 104

5.6 Complexity and Cost Analysis 105

5.6.1 Storage Cost 107

5.6.2 Exact NN Query 107

5.6.3 Approximate NN Query 108

5.7 Experiments 108

5.7.1 Data Set and Query 108

5.7.2 Performance Measurement 109

5.7.3 Parameter Tuning 111

5.7.4 Frequent Insertion 112

5.7.5 Exact NN Query 113

5.7.6 Approximate NN Query 116

5.8 Summary 119

6 LANGG : A Travel Mashup System For Location-Based Services 120

6.1 System Framework 121

6.2 Demonstration 123

6.2.1 Search Closest Travel Services 123

6.2.2 Search Location Using Tags 124

6.2.3 Search Location by Image 125

7 Conclusion and Future Work **128**

7.1 Conclusion 128

7.2 Future work 130

LIST OF TABLES

3.1 Possible sets of $\{A_1, A_2\}$, $\{B_1, B_2\}$, and $\{C_1\}$ 40

3.2 Keyword distribution on Texas data set 56

5.1 Notation table 98

5.2 Parameter Setting 114

5.3 Index storage cost 114

5.4 Top-50 NN query selectivity 115

5.5 Storage cost of HashFile and LSB forest 117

LIST OF FIGURES

1.1 Singapore restaurants displayed on Google Maps 3
1.2 Singapore hotels displayed on Bing Maps 4
1.3 Flick photos in Singapore 6
1.4 Youtube videos in Singapore 7
1.5 A travel blog example with tags 8
1.6 A travel image example with tags “bull”, “bronze” and “sculpture” 9
1.7 Distribution of tag “zoo” 10
1.8 Distribution of tag “USOPEN” 10

3.1 Distribution of beach, seafood restaurant, shopping mall and cinema in Singapore 25
3.2 Node information of bR*-tree 31
3.3 An illustration of search in one node 32
3.4 a priori algorithm applied to search in multiple nodes 41
3.5 Extended a priori algorithm 42
3.6 Example of active MBR 45

3.7 Performance on increasing TK 52

3.8 Performance on increasing DS 53

3.9 Performance on increasing KD 54

3.10 Performance on increasing DM 55

3.11 Performance on two real data sets 57

4.1 The index of R*-tree and inverted index 63

4.2 Bottom-up construction of virtual bR^*-tree 65

4.3 Order of $NodeSet$ candidates checked 67

4.4 Degradation of keyword spatial importance 70

4.5 Approximate model for degradation of keyword spatial importance . 71

4.6 Scalability in terms of m 74

4.7 Scalability in terms of the number of locations 75

4.8 Scalability in terms of the number of tags 76

4.9 Distribution of tag “zoo” 77

4.10 Distribution of tag “USOPEN” 77

4.11 Example tag queries 78

4.12 Example results returned by mCK and Google Maps 80

4.13 Ranking score for tag query 81

4.14 Local service queries 82

4.15 Accuracy result 83

4.16 Example photo queries 84

5.1 A random projection example 94

5.2 Hash value frequency of a color histogram dataset 95

5.3 New frequency distribution of window based hashing 97

5.4 The structure of HashFile and HashNode 99

5.5 Approximate search in the tree node 106

5.6 Tune parameter W 111

5.7 The number of pages in the root node 113

5.8 Performance of the exact top-50 NN search 116

5.9 Approximate NN query results of LSB-tree and HashFile 118

5.10 Approximate NN query results of Multi-probe LSH and HashFile . 118

6.1 The framework of location detecting in Web 2.0 applications 121

6.2 System portal of **LANGG** 123

6.3 Interface of locating closest travel services 124

6.4 Query by “bird park” 125

6.5 Query by ”chicken rice” 126

6.6 Example image query of Merlion 127

6.7 Example image query of Zoo 127

CHAPTER 1

INTRODUCTION

The emergence of Web 2.0 applications [5], including social networking sites, wikipedia and multimedia sharing sites, has changed the way of how information is generated and shared. In these websites, massive amounts of data have been generated by users in a collaborative manner. Data from different sources can be further integrated to create new services, leading to an important type of application named mashup [4]. Mashup applications have attracted great research and commercial interest. They provide the ability to develop new integrated services quickly and make them tangible to the business users through friendly map interfaces. In this thesis, we focus on map mashup application, in which various spatial web resources are integrated and displayed on map. We tackle the problem of efficient location-based spatial keyword query processing and build a travel map mashup system, named **LANGG**, to provide users with location-based services.

1.1 Travel Map Mashup Applications In Web 2.0

In Web 2.0, users are allowed not only to retrieve information from websites, but also to interact and collaborate with each other to contribute new contents. A huge amount of media resources in various formats have been generated and shared in recent years. In July 2006, YouTube was reported to be serving 100 million views and $65,000$ new video uploads per day [10]. In 2007, millions of photos were uploaded in Flickr per day [101]. More recently, in January 2011, Facebook announced that a record-breaking amount of 750 million photos had been uploaded over New Year's weekend [7]. Most of these user-generated contents (UGC) are publicly accessible and constitute a luxuriant database to support many upper layer applications. Among all these applications, mashup plays an important role in improving data usability and accessibility. It combines data from different sources and shows great flexibility in creating new services. A very common mashup application is to combine digital map services like Google Maps, Yahoo Maps or Bing Maps with other useful information resources. For example, Wikipediavision [8] integrates Google Maps and a Wikipedia API [9] to display Wikipedia articles with geographical context on the map. Other common examples include online house rental system, hotel booking system, trip planning system and so on.

In travel market, there also exist a large number of mashup systems whose main objective is to provide users with valuable travel guide information. A common assumption in these systems is that users have already got a clear travel destination and wish to retrieve useful guide information in that location. For example, a user who is going to travel in Singapore is interested to know the famous sight attractions, affordable and well located hotels and food outlets serving local favorite dishes. To fulfill the requirement, these systems need to collect as many travel resources as possible, integrate them into a spatial database and provide a user-

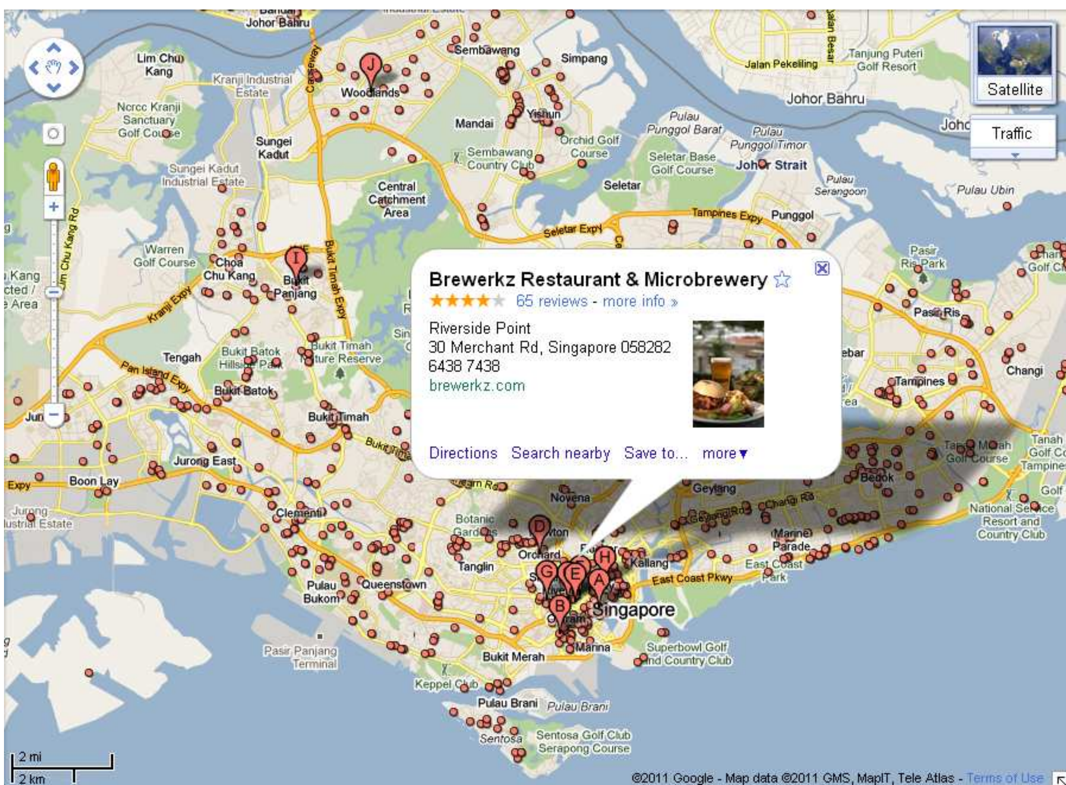


Figure 1.1: Singapore restaurants displayed on Google Maps

friendly interface for navigation. Figure 1.1 shows an example of restaurant mashup system based on Google Maps. Each marker on the map represents one restaurant at a location. When a marker is clicked, an infowindow containing basic information of that restaurant such as name, address, telephone number, user ranking and reviews is popped up. From this information, users can get a general idea and make a decision about whether to go there for dinner. Figure 1.2 shows another travel mashup system which displays hotels of Singapore on Bing Maps. It adopts a similar visualization strategy. When a hotel marker is clicked, summary information of that hotel including price and user ranking score is displayed for a quick decision making. Such a map visualization tool is common for presenting travel resources and is convenient for users to browse travel information. Users can easily tell from the map where the point of interest (POI) is located and get the textual description

Figure 1.2: Singapore hotels displayed on Bing Maps

and summary of that POI.

1.2 Locating m Closest Keywords In a Spatial Database

These map mashup systems generate a huge amount of spatial items in various formats, including documents, photos and videos. They are often associated with both textual and spatial attributes. In order to leverage such a large scale spatial-textual database that is publicly accessible, keyword queries with spatial constraints have received significant attention from the spatial database research community and the industry. Typical queries include:

- **Nearest Neighbor Query [95]:** Find the nearest restaurant which serves chilly crab from a given hotel.

- **Range Query [87]:** Find a 7-Eleven convenient store within 300 meters from a given location.

- **Closest Pair Query [43]:** Find a pair of gas station and shopping mall closest to each other.

- **Spatial Join [34]:** Find all pairs of cinema and French restaurant located in the same shopping mall in Singapore.

In this thesis, we address a novel query named mCK (m Closest Keywords), which aims at finding a location where m query keywords are closest to each other. A useful application is to find m closest travel service providers. Here, travel service could refer to “spa”, “skiing”, “hiking”, “seafood restaurant” or any travel related service that users could be interested in. Each service is represented by a keyword. For example, in Figure 1.1, the markers on the map indicate the locations where keyword “restaurant” appears. It is possible that multiple service providers are at the same location. To facilitate the statement of our problem, we treat them as multiple spatial tuples, each with one service, at the same location. Locating closest services is a very useful function to save the transportation cost between the service providers and allow users to enjoy all the desired services when they have limited staying time in a place. For example, when a user is travelling in Singapore, he can submit a query “beach”, “chilly crab”, “shopping mall” and “cinema”. Our system will return a beach, a seafood restaurant serving chilly crab, a shopping mall and a cinema that are closest to each other. The user can take a swim in the sea, enjoy the chilly crab, go shopping and watch a movie conveniently.

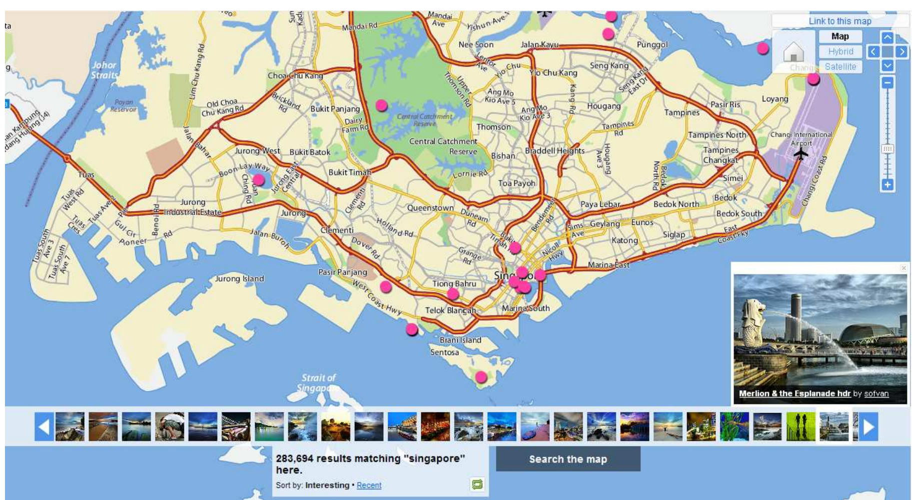


Figure 1.3: Flick photos in Singapore

1.3 Locating Web Resources by Spatial Tag Matching

Besides textual information, multimedia objects such as photos and videos can also be displayed on the map. Users can upload their photos to Flickr and share with their friends. When they upload a photo, they can also add additional spatial and textual information for the photo. For example, they are allowed to create a marker on the map to indicate where the photo was taken[1]. Figure 1.3 illustrates such a photo mashup example in Flickr. We can see from the figure that there are 283, 694 photos that match the keyword "Singapore" and have geographic location to display on the map. Each marker represents a photo and users can tell where a photo was taken from the location of the marker. When a marker is clicked, other information like author and title are available. Similarly, an example of video mashup is shown in Figure 1.4 which displays videos from Youtube in Singapore on Google Maps.

[1]If the camera is equipped with GPS, this geographic information can be automatically read from the EXIF [3] data embedded in the photo.

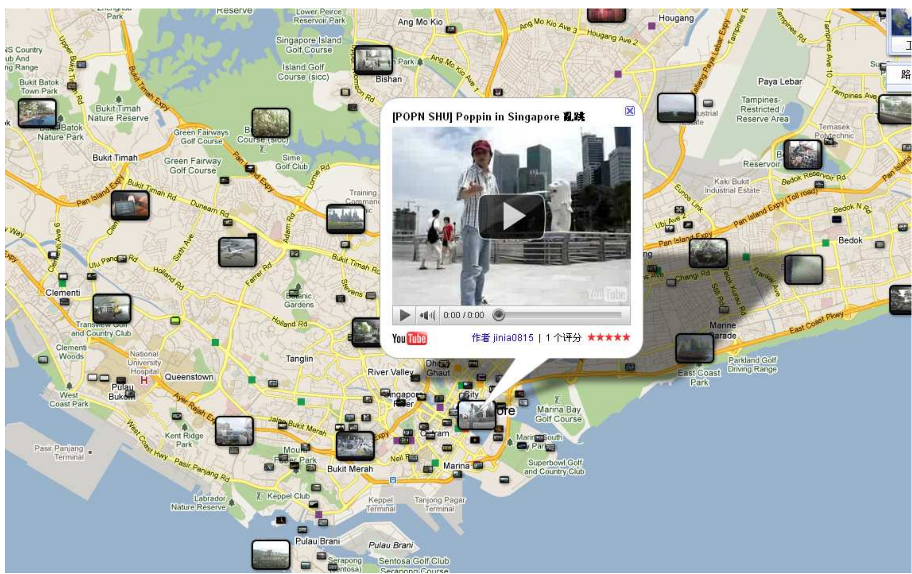


Figure 1.4: Youtube videos in Singapore

With these online resources as the underlying database, location detecting service has attracted significant interest in recent years due to its commercial potential to the search engine in providing local or personalized service to customers. Existing methods take advantage of gazetteer terms in the text body [36, 47, 86, 21, 109, 33, 23] and hyperlink structure [47, 86, 109]. In this thesis, we propose a spatial tag matching method which utilizes tags as a new information source for location detection if the web resource is associated with tags. In Web 2.0, tagging is a popular means to annotate various resources, including news, blogs, speeches, photos and videos. Users are encouraged to add extra textual terms as semantic description or summarization for the objects. With human intelligence involved, the tags are well phrased. An example of travel blog about Sentosa Island in Singapore is shown in the Figure 1.5. The author of the blog contributes four tags. “Palawan” is a gazetteer term, indicating the name of the beach. “beach”, “island”, and “wooden bridge” are representative scenes in Sentosa. Another annotation example is shown in Figure 1.6. The photo is associated with tags “bull”, “bronze” and “sculpture”,

which are used as description of the object in the photo. From these two examples, we observe that although documents are essentially different from other media in textual context, we can use tags as a uniform semantic wrapper so that different types of web resources can be treated equally. Tagging provides a means to build a uniform model for our spatial database :

Definition 1.1 (Uniform Mapped Resource Model). *Let $\mathcal{S}$ be the d-dimensional geographical space and $\mathcal{T}$ be the tag space. Each object o can be represented as $o = [ref, c_1, ..., c_d, t_1, ..., t_n]$ where $[c_1, ..., c_d] \in \mathcal{S}$, $t_i \in \mathcal{T}$ and ref is the reference to the object itself.*

A travel blog example

We reached the beaches and decided on 'Palawan' as this is a name of an hammock ***Sentosa*** *island in the* ***Philippines*** *that we are planning on visiting, so we will be able to compare at a later date! The sand was soft and beautiful but you could tell the beach was man-made as you only had to dig your hands down a little to find a layer of concrete. The sea was pretty dark and there was a lot of smog in the atmosphere due to the busy port not far away. Despite this we had a really nice day on the beach, and for all that the island has to offer this beach is more than decent enough to spend a few days if you were on a family holiday, which we guess the resort is aimed towards.*

The real beauty of the place is that it is like ***Disneyworld****. Everything is designed how you would imagine it in a fairytale. The beach was a beautiful shaped bay with a little island which could be reached by a wooden bridge. We crossed the bridge and climbed a pagoda style wooden lookout point to get a good view over the beach. After our day sunning, we took the monorail back to the mainland as if was*

Tag : beach, island, Palawan, wooden bridge

Figure 1.5: A travel blog example with tags

Figure 1.6: A travel image example with tags “bull”, “bronze” and “sculpture”

Based on the uniform tag model, our spatial database collects user-generated web resources. Some of them are associated with tags and spatial attributes which are referred to as geo-tags. Based on our observation, the geo-tag data set is of acceptable quality. If we consider Flickr for instance, most of the photos are associated with relevant tags and are correctly marked in the map. For a spatial subject, its related geo-tags are clustered around the real location. We illustrate some examples in Figure 4.9 and 4.10. As shown in Figure 4.9, the tag “zoo” is mainly distributed in three spatial clusters corresponding to Bronx Zoo, Central Park Zoo and Queens Zoo Wildlife Center respectively. Similarly, in Figure 4.10, there are a large number of “USOPEN” tags gathering around the Arthur Ashe Stadium where the tennis match is held. These geo-tag clusters can be utilized to identify the locations of popular resources and events because related tags will emerge around that area. It also provides us with new opportunities to locate resources in a more precise geographical scale.

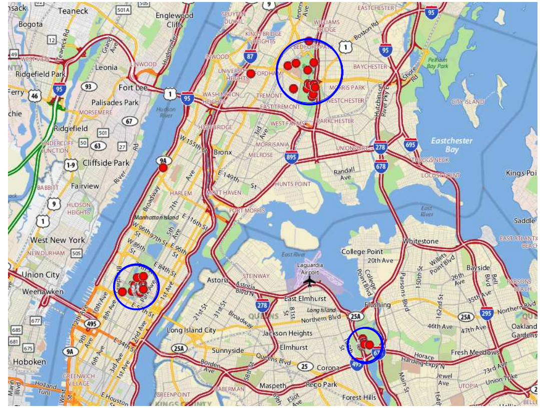


Figure 1.7: Distribution of tag “zoo”

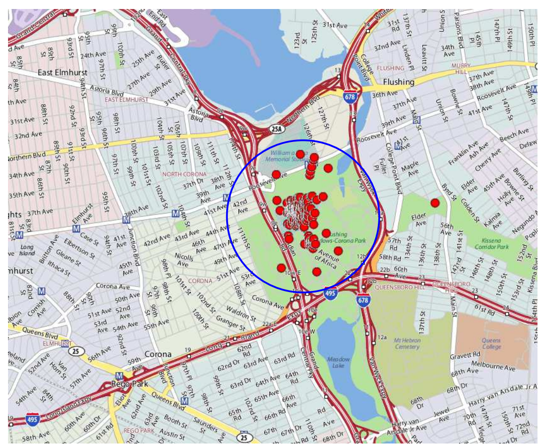


Figure 1.8: Distribution of tag “USOPEN”

If the query is a tagged item, we use a spatial tag matching method to detect the location. More specifically, we aim at finding a location on the map that best matches the query tags. We adopt the idea of keyword search in relational database [18, 59, 78, 82, 74], XML [55, 41, 113, 80, 80] and graph [31, 63, 75] that returned results by default contain all the query keywords and smaller result size is preferred as the keywords are considered to be closer to each other. In other words, we want to find tuples in the space matching the query tags and these tuples are as close to each other as possible. This problem is similar to locating closest travel services except that the number of travel services in a city is not

large but there could be hundreds of thousands of tags. Therefore, the spatial tag matching solution must be scalable in terms of the number of tags so as to survive in the Web 2.0 environment. Moreover, only measuring the relevance between tags is not enough. The spatial relevance of tags with respect to the detected location also needs to be taken into account. Thus, we propose a new *geo-tf-idf* ranking mechanism to improve the detection precision.

1.4 Locating Landmark Photos by Content-Based Matching

The problem of locating web resource is more difficult when tags are not available. To solve it, content based methods have to be used. If the input is a textual document, we can simply use existing methods to detect the geographical context of web documents or first adopt existing key phrase extraction methods to automatically detect the terms with geographical context [51, 112, 107, 46, 114]. These terms are considered as tags for the document and the problem becomes locating a web document using spatial tag matching, which has been discussed above. If the input is an image, this is essentially a landmark recognition problem, which has attracted immerse attention in recent years. In the Web 2.0 age, users have created, tagged and shared large amounts of customized photos. Such large-scale, well-organized and publicly-accessible web data are retrieved as the underlying image database. The landmark recognition problem is usually tackled by comparing the input photo with all the photos in the database. The nearest neighbor is retrieved as the match result. Thus, efficient access to multimedia objects needs to be supported in order for the Web users to benefit from such data.

In this thesis, we explicitly address the problem of image similarity search and

design an efficient index to process the image data. We treat high dimensional image feature as "keyword" and represent a geo-image as a set of spatial keywords at the same location. Nearest neighbor (NN) is used for image match. Processing strategies for both exact and approximate NN queries of image data are investigated. The former has wide applications in similarity search, pattern recognition, clustering and classification. The latter is particularly suitable for efficient retrieval in a large scale database at the risk of certain loss in quality. Since video can be modelled as a sequence of images via key frame extraction, we will not cover how to detect the location of a video specifically.

1.5 LANGG : A Location-Based Travel Mashup System

To utilize the above efficient spatial keyword query processing technique, we design and develop a new type of travel mashup system, named **LANGG** to provide location-based services. The main objective of our system is to recommend users a travel destination based on their personal interests. Users can submit a set of travel services they would like to enjoy, an interesting travel blog or even a travel photo with beautiful scene. Our system is able to detect a location on the map that best matches the input so that this location can be recommended to users as the travel destination. The system supports three main applications :

- **Application I :** locating closest travel services. In this application, the input consists of a set of travel services and the user's goal is to return a location where the submitted services are closest to each other. Such a query is essentially useful to save the transportation time if the user only has very limited staying time.

- **Application II :** detecting the geographical context of travel media with tags. In this application, users are allowed to submit a travel media associated with a set of annotated tags. The detected location of the media will be returned. A location that best matches the query tags is returned as the detection result. Such a query is helpful to assist users in finding a desired travel destination.

- **Application III :** detecting the geographical context of travel media without tags. This application is similar to Application II except that tags are not available here and content based matching methods have to be used. We focus on the landmark recognition problem and design an new index to efficiently answer nearest neighbor query in a large scale image database.

1.6 Contribution of the Thesis

In this thesis, we mainly address efficient location-based spatial keyword query processing strategies. First, we introduce a novel query, named mCK, to locate m closest keywords in a spatial database. Such a query is very useful to find closest local services in a travel destination when users have limited staying time in a place. To answer an mCK query, we propose efficient index and search algorithm which significantly reduce the search space. We also adopt mCK query as a spatial tag matching method to detect the geographical context of web resources. We build a uniform model to represent various types of web resources and improve the index and search algorithm to support tag matching in a spatial database with hundreds of thousands tags. In addition, we tackle the landmark recognition when tags are not available. Efficient index is proposed and experimental results show that it provides better efficiency than state-of-the-art works.

With the efficient spatial keyword processing technique, we build a travel mashup system to recommend users a travel destination based on their personal interest. Users can submit a set of travel services they would like to enjoy, an interesting travel blog or even a travel photo with beautiful scene to find the related location. Our system can be considered as a complement to existing travel mashups. Users can first use our system to find a desired travel destination and then turn to other systems for more travel guide information.

1.7 Thesis Organization

The rest of the thesis is organized as follows:

In Chapter 2, we review existing works that are related to three locating functions provided by our system. The literature review falls into three categories : finding m-closest keywords in spatial databases, locating web documents and image recognition.

In Chapter 3, we address how to efficiently find the closest travel services. We introduce a new spatial index called the bR*-tree, which is an extension of the R*-tree. Based on the index, we propose efficient search algorithm with effective pruning rules that can significantly reduce the search space.

In Chapter 4, we tackle the problem of locating web resources using spatial tag matching. We further improve the method in Chapter 3 to be scalable in terms of total number of tags. Moreover, we propose a new *geo-tf-idf* ranking mechanism to measure the geographical relevance of query tags.

In Chapter 5, we focus our discussion on the landmark recognition problem and propose a novel index structure, named **HashFile**, for efficient retrieval in a large image database.

In Chapter 6, we present the framework and design issues of our system and finally, we conclude the whole thesis in Chapter 7.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we conduct a literature review over location-based spatial keyword query processing technique. First, we review the existing works about how to find m-closest keywords in a spatial database. Then, we examine how to detect the geographical context of web document and images.

2.1 Finding m-Closest Keywords in Spatial Databases

The topic of keyword search in spatial databases has been well studied in recent years [57, 50, 54, 42, 38, 68]. The spatial keyword search is considered as the combination of spatial query [52, 95, 87] and keyword search. Thus, it contains both spatial and textual constraints. In order to efficiently process the spatial keyword search, various hybrid index structures have been proposed by integrating R-tree [56] or its variants [98, 26] with inverted index or signature file. Hariharan et al. [57] introduced a spatial keyword query with range constraints. Each

spatial document returned is required to intersect with the query MBR (Minimum Bounding Rectangle) and matches all the user-specified keywords. They proposed a hybrid index of R*-tree and inverted index, called KR*-tree, to answer the query. Felipe et al. [50] proposed a similar query type by combining k-Nearest Neighbor (kNN) query and keyword search, and used IR^2, a hybrid index of R-tree and signature file, for query processing. Göbel proposed a more general hybrid index for geo-textual searches [54]. Only the most frequent terms are indexed in the extended R-tree and the filtering strategy relies on the frequency of the query keyword.

Since the ranking methodology of spatial keyword search in the above methods is based on either the distance to the query point [50] or the relevance with respect to the query keywords [57], it is necessary to seamlessly combine both the spatial and textual features in the ranking function. To fill this gap, Khodaei et al. [68] developed a new distance measure named spatial tf-idf and proposed an index structure called Spatial-Keyword Inverted File for efficient processing based on the distance measure. Cao et al. also proposed that both location proximity and text relevancy should be taken into account during the ranking [42, 38]. They developed an efficient framework for top-k spatial document retrieval.

The extension to the traditional keyword search is divided into two categories. The first category relaxes the keyword search constraint to handle approximate spatial keyword search [115, 20, 19], which is especially useful when users have no idea of the correct spelling of some keywords. To handle approximate spatial keyword search, MHR-tree was proposed in [115] to augment R-tree nodes with min-wise signature [35]. Alternatively, Alsubaiee et al. [20, 19] took advantage of R-tree and gram-based [108, 73] inverted index and built system prototypes to demonstrate the practicality of their solution. The other category attempts to combine keyword search with more complex spatial queries. Besides the popular

kNN query and range query, closest-pair queries for spatial data using R-trees have also been investigated [43, 44, 106]. Users can submit two different keywords in order to find the closest pair in the spatial database. In this thesis, we extend the closest pair query to a more general case and propose a novel query, named mCK, to find m closest keywords in the database. In other words, our mCK query allows more than two keywords. The tuples matching all the keywords and with minimum diameter are considered as the best result. Another type of query similar to mCK query is named optimal sequenced route query [100, 99]. The query aims at finding a route of minimum length starting from a given source point and covering all the typed keywords. In comparison, our mCK query does not have such a start point. and the distance measure used in the ranking is also different.

The mCK query can be answered by adopting the idea of Multi-Way Spatial Join (MWSJ) [90, 89, 84, 88]. The MWSJ works in the way that given m keywords, multiple R*-trees, one for each keyword, will be built. Candidate spatial windows for mCK query result can be identified among these R*-trees. The join condition here becomes "closest in space" instead of "overlapping in space". Unfortunately, as m increases, this approach suffers from two serious drawbacks. First, it incurs high disk I/O cost in identifying the candidate windows (due to synchronous multiway traversal of R*-trees) since it does not inherently support effective summarization of keyword locations. Second, it may not be able to identify a "tight" set of candidate windows since it determines candidate windows in an approximate manner based on the leaf-node MBRs of R*-trees without considering the actual objects. To process mCK queries in a more scalable manner, we use one R*-tree to index all the spatial objects as well as their keywords. Integrating all the information in a single R*-tree provides more opportunities for efficient search and pruning.

2.2 Locating Web Documents

Location detecting service has attracted significant interest in recent years due to its commercial potential to the search engine in providing local or personalized service to customers. Many existing geographical search engines [116, 85, 91, 2, 1] can benefit from this process to organize and index the web documents more appropriately. In order to detect geographic locations of web resources, various geographic information sources are exploited. The most straightforward method is to analyze the textual contents and extract the geographic entities [36, 47, 86, 21, 109, 33, 23]. First, a gazetteer dictionary is built. The dictionary contains authorized gazetteer information from various sources such as USGS Geographic Names Information System [6], World Gazetteer [11], UNSD [12], USPS [13], Yahoo Regional [14]. Recently, geographical entities in Wikipedia have also been considered as a new source [111]. Given such a dictionary, the next step is to extract the entities with geographical context in the web document, including postal code, telephone number, geographical feature names. During this process, it is important to eliminate the ambiguities. In [21], Amitay et al. tackled the problem of geographic name disambiguation. They distinguished both geo/non-geo and geo/geo ambiguities. Besides the web page content, hyperlink structure [47, 86, 109] and server ip address [36] are useful information to infer the location of the web page. A page that is popularly accessed or cited by other local pages or users is considered to be relevant to a local area. In this thesis, we propose a location detecting method on a new information source. If the document is associated with user contributed tags, we use these tags as the query and find a spatial location that best matches these tags. Otherwise, we can adopt existing methods [49, 51, 112, 107, 46, 114] to extract key phrases from a web document and treat the key phrases as the tags. Given the location candidates, we need to measure their relevance with the input keywords,

which is a geo-ranking problem. Geo-ranking mechanisms were proposed in [47, 24] to measure the relevance of tags with respect to a location. In these works, a similar strategy to PageRank was proposed to measure the local popularity using back link locations. To further emphasize the local importance, geographic power and spread measurements are defined in different context. Geographic power refers to the popularity of a page in a local area and is measured by the normalized number of desired links to the page. Spread measures how uniform are the distribution of page's back links. The back links of the resources are however not available in most cases. As such, a new geo-ranking mechanism is required to measure the relevance of geo-tags to the area that they are located in.

2.3 Landmark Recognition

Given a query image without annotations, the problem of location detecting can be considered as landmark recognition. Suppose we have built a spatial photo database using photo clustering so that images belonging to the same landmark are clustered. To achieve this goal, most of the existing works [45, 67, 93, 76, 79] take advantage of the associated geo-location, descriptive tags as well as their visual contents to group similar photos together and assign each group a class label. As a more general approach for the situation when the associated information is not available, Zheng et al. cluster the landmark photos purely based on the visual contents. After the raw clustering, the duplicate or near-duplicate photos can be removed using the methods proposed in [65, 40].

Given the landmark photos taken across the whole planet, building a world-scale landmark recognition engine becomes an interesting but challenging problem. The state-of-the-art systems [58, 92, 118] use the same matching mechanism (image-to-

image) but based upon different visual features. In [58], a bucket of features such as color histograms, texton histogram, line features, gist descriptor and geometric context are taken into consideration. In [92, 118], SIFT feature is the main concern. However, as has been argued, image-to-image matching is not scalable to a vast collection of photos. In this thesis, we aim at building an efficient index to process the nearest neighbor search.

In this following part, we briefly review the index structures that are widely used to process exact or approximate NN queries in multimedia databases.

2.3.1 High Dimensional Index for Exact NN Query

There exist mainly three types of approaches to answering exact NN query based on the idea of space partitioning [94, 29, 64, 77], data approximation [110, 96, 70, 27] or one dimensional transformation [28, 61, 117].

The most common index structures are based on the notion of space partitioning, resulting in various types of tree-based index structures such as k-d-b tree [94], X-tree [29], SR-tree [64] and TV-tree [77]. In these trees, the pruning power of these methods degrades as the dimensionality of data increases. This can be offset by maintaining trees with very large height, but in that case since the number of internal nodes grows exponentially with the tree height, tremendous storage overhead is incurred. The performance of such trees degrades to be worse than linear scan.

VA-file and iDistance are the representative indexes for data approximation and one dimensional transformation respectively. Weber et al. proposed VA-file[110] which uses bit encoding for pruning and takes advantage of linear scan for query processing. A look-up on the real data file is triggered when a point cannot be pruned based on the compressed representation. A proper compression rate must

be specified for the best performance. Otherwise, the performance would become CPU-bound or IO-bound when it is set too large or too small. The major drawback of VA-file is the lack of flexibility in a dynamic environment as the data in the two files are sequentially stored. iDistance [61, 117] attempts to solve the problem by building a light-weight index using B^+-tree. A collection of reference points, which can be dynamically or statically determined, are selected implicitly to partition the space in Voronoi cells. Instead of splitting the data space, iDistance indexes the distance to these reference points. The advantage is that the index size is relatively small and it also demonstrates satisfactory pruning power. However, its performance is sensitive to the selection of the reference points and too much random access of the disk pages is required as the selectivity is coarse for NN query in the high dimensional space.

In contrast, HashFile combines the advantage of random projection and linear scan. Random projection is useful to filter away the data points that are far away and the remaining candidates are processed efficiently using a linear scan. In our experiments, we compare HashFile with VA-file and iDistance to show the superiority of our index.

2.3.2 LSH for Approximate NN Query

LSH [60, 53] has been widely applied to answer the approximate NN query and shown to be quite effective for similarity search in multimedia databases including text data [103], audio data [37], images [66] and videos [48]. The query cost grows sub-linearly with the data set size in the worst case. However, it is a trivial job to tune a good tradeoff between the precision and recall. In practice, hundreds of hash tables have to be built for a high search accuracy [53]. To reduce the number of hash tables, Lv et al. proposed multi-probe LSH [83], which can obtain the same

search quality with much less tables. Since multi-probe LSH is adhoc and without theoretical guarantee, Tao [105] recently has proposed LSB-tree to address both the quality and the efficiency of multimedia retrieval. The hash values are represented as one-dimensional Z-order values and indexed in a B^+-tree. Multiple trees can be built to improve the result quality. Compared to existing LSH methods, HashFile only recursively partitions dense buckets to achieve more balanced data partitions. Each bucket hosts similar number of objects. In addition, HashFile takes advantage of linear scan, which is more efficient than random access used in LSB-tree.

CHAPTER 3

LOCATING CLOSEST TRAVEL SERVICES

In this chapter, we address a novel spatial keyword query called m-closest keywords (mCK) query to locate closest travel services. To efficiently answer an mCK query, we introduce a new index named bR*-tree, which is an extension of the R*-tree. Based on bR*-tree, we exploit a priori-based search strategy to effectively reduce the search space. We also propose two monotone constraints, namely the distance mutex and keyword mutex, as our a priori properties to facilitate effective pruning. Our performance study demonstrates that our search strategy is indeed efficient in reducing query response time. Moreover, it demonstrates remarkable scalability in terms of the number of query keywords.

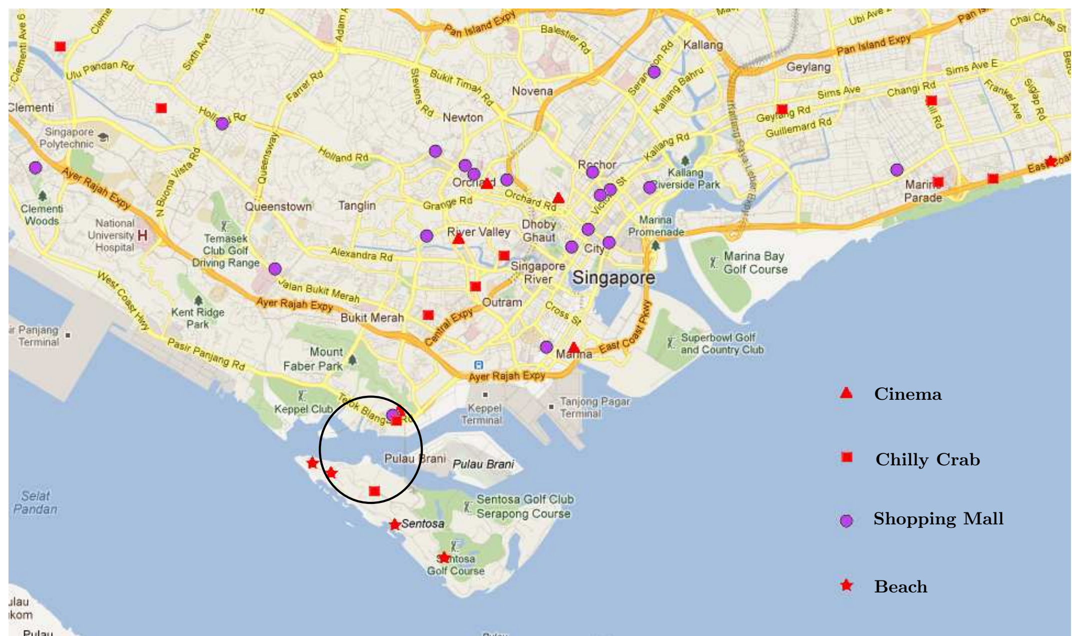


Figure 3.1: Distribution of beach, seafood restaurant, shopping mall and cinema in Singapore

3.1 Introduction

A user travelling in Singapore may make a schedule like this: First, he wants to take a swim in the sea. After that, he would like to go shopping and find a seafood restaurant for the delicious chilly crab. Finally, he would watch movie in a cinema to end the day trip. The desired recommendation is a place on the map where a beach, a shopping mall, a seafood restaurant with chilly crab and a cinema are close to each other. Note that in this application, we represent each type of travel service as a unique keyword. Thus, “beach”, “shopping mall”, “chilly crab” and “cinema” are considered as four spatial keywords.

Figure 3.1 shows the spatial distribution of the four travel services in Singapore. We use different shapes to represent different types of services and each shape in the map indicates that there is a corresponding service provider in that location. Our goal is to find a location in the map where there are four points with different

shapes closest to each other[1]. In this example, we can recommend **Vivo City** as the result for these four keywords. He can enjoy the sun bath on the Sentosa island and go shopping, dining and watching movie in Vivo City. We call this problem mCK Query Problem and formally define it as follows:

Definition 3.1 (mCK Query Problem)**.** *Suppose we have a spatial database with d-dimensional tuples represented in the form* $(c_1, c_2, \ldots, c_d, w)$*, where* c_i *is the coordinate in the i-th dimension and w is the service keyword. Given a set of m query keywords* $Q = \{w_{q_1}, w_{q_2}, \ldots, w_{q_m}\}$*, the mCK Query Problem is to find m tuples* $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$*,* $T_i.w \in Q$ *and* $T_i.w \neq T_j.w$ *if* $i \neq j$*, and* $diam(\mathcal{T})$ *is minimum.*

The closeness for a set of m tuples can be measured as the maximum distance between any two of the tuples:

Definition 3.2 (Diameter)**.** *Let* $\mathcal{S}$ *be a tuple set endowed with a distance metric* $dist(\cdot,\cdot)$*. The* diameter *of a subset* $\mathcal{T} \subseteq \mathcal{S}$ *is defined by*

$$diam(\mathcal{T}) = \max_{T,T' \in \mathcal{T}} dist(T, T').$$

Different distance metric will give rise to different geometry of the query response:

- If $dist(\cdot,\cdot)$ is the ℓ_1-distance metric, then the response containing all the keywords of the query is a square oriented at a 45° angle to the coordinate axes.
- If $dist(\cdot,\cdot)$ is the ℓ_2-distance (Euclidean distance) metric, then the response containing all the keywords of the query is a circle of minimum diameter.

[1] Tuples with multiple service keywords can be treated as multiple tuples, each with a single keyword and located in the same position.

- If $dist(\cdot, \cdot)$ is the ℓ_∞-distance metric, then the response containing all the keywords of the query is a minimum bounding square.

In this example, the diameter for the four keywords is precisely the diameter of the circle drawn in Figure 3.1. Users can specify their respective mCK queries according to their requirements.

A naive mCK query processing approach is to exhaustively examine all possible sets of m tuples of objects matching the query keywords. We can build m inverted lists for each of m keywords with each list having only spatial objects that contain the corresponding keyword. The exhaustive algorithm can be implemented in a multiple nested loop fashion. If the number of objects matching keyword i is $D(i)$, then the number of tuples to be examined is $\prod_{i=1}^{m} D(i)$. This is prohibitively expensive when the number of objects and/or m is large.

Spatial data is almost always indexed to facilitate fast retrieval. We can adopt the idea of Papadias et al. [89] to answer mCK query. Given N R*-trees, one for each keyword, candidate spatial windows for mCK query result can be identified by executing multiway spatial joins (MWSJ) among the R*-trees. The join condition here becomes "closest in space" instead of "overlapping in space" [89]. When m is very small, this approach accesses only a small portion of the data and returns the result relatively quickly. However, as m increases, this approach suffers from two serious drawbacks. First, it incurs high disk I/O cost for identifying the candidate windows (due to synchronous multiway traversal of R*-trees) since it does not inherently support effective summarization of keyword locations. Second, it may not be able to identify a "tight" set of candidate windows since it determines candidate windows in an approximate manner based on the leaf-node MBRs of R*-trees without considering the actual objects. To process mCK query in a more scalable manner, we propose to use one R*-tree to index all the spatial objects as

well as their keywords. Integrating all the information in a single R*-tree provides more opportunities for efficient search and pruning. The other non-overlapping space partitioning method like PR-tree [22] or quadtree [97] could be used but may not perform better than R*-tree. The reason is that we need to check all the possible combinations of tree nodes and the design principle of R*-tree is to guarantee that objects closer to each other are more likely to be stored in the same node. On the other hand, R*-tree is a balanced tree. Compared to quadtree, it has worst case performance bound.

The main contributions of this work are as follows.

- We propose a novel spatio-keyword query, called the *mCK query*, which is useful to locate a group of travel services close to each other on the map.
- We propose a new index, called *bR*-tree*, for query processing. bR*-tree extends the R*-tree to effectively summarize keywords and their spatial information.
- We incorporate efficient a priori-based search strategy, which significantly reduces the combinatorial search space.
- We define two monotone constraints, namely the distance mutex and keyword mutex, as a priori properties for pruning. We also provide low-cost implementations for the examination of these constraints.
- We conduct extensive experiments to demonstrate that our algorithm not only is effective in reducing mCK query response time, but also exhibits good scalability in terms of the number of query keywords.

The remainder of this chapter is organized as follows. Section 3.2 introduces bR*-tree. Section 3.3 proposes a priori-based mCK query processing strategies and

two monotone constraints used as a priori properties to facilitate pruning. Efficient implementations for the examination of these two constraints are also provided. Section 3.4 reports our performance study. Finally, we conclude our chapter in Section 3.5.

3.2 bR*-tree: R*-tree With Bitmaps and Keyword MBRs

To process mCK queries in a more scalable manner, we propose to use one R*-tree to index all the spatial objects and their keywords. In this section, we discuss the proposed index structure called the *bR*-tree*.

bR*-tree is an extension of the R*-tree. Besides the node MBR, each node is augmented with additional information. A straightforward extension is to summarize the keywords in the node. With this information, it becomes easy to decide whether m query keywords can be found in this node. If there are N keywords in the database, the keywords for each node can be represented using a bitmap of size N, with a "1" indicating its existence in the node and a "0" otherwise. For example, a bitmap $B = 01001$ reveals that there are five keywords in the database and the current node can only be associated with the keywords in the second and fifth positions of the bitmap. This representation incurs little storage overhead. Moreover, it can accelerate the checking process of keyword constraints due to the relatively high speed of binary operations. Given a query $Q = 00110$, if we have B AND $Q = 0$, it implies that the given node does not have any query keywords and thus, this node can be eliminated from the search space.

Besides the keyword bitmap, we also store the *keyword MBR* in the node to set up more powerful pruning rules. The keyword MBR of keyword w_i is the MBR for

all the objects in the nodes that are associated with w_i. It summarizes the spatial locations of w_i in the node. Using this information, we know the approximate area in the node which each keyword is distributed. If M is the node MBR and M_i is the keyword MBR for w_i, we have $M_i \subseteq M$.

When N is a large number, the cost for storing the keyword MBR is very high. For example, suppose there are a total of 100 keywords in the database and the objects are three-dimensional data. Spatial coordinates are usually stored in double precision, which occupies eight bytes per coordinate. It would therefore take $100 \times 3 \times 8 \times 2 = 4800$ bytes to store the keyword MBRs in one node. To reduce the storage cost, we split the node MBR into segments along each dimension. Each keyword MBR is represented approximately by the start and end offsets of the segments along each dimension. The range of an offset that occupies n bits is $[0, 2^n - 1]$. In our implementation, we set $n = 8$ (resulting in 256 segments) and found that it provided satisfactory approximation.

After being augmented with bitmap and keyword MBR, non-leaf nodes of bR*-tree contain entries in the form of (*ptrs, mbr, bmp, kwd_mbr*), where

- *ptrs* represents pointers to child nodes;
- *mbr* is the node MBR that covers all the MBRs in the child nodes;
- *bmp* is a keyword bitmap, each bit of which corresponding to a specific keyword, and marked "1" if the node contains that keyword and "0" otherwise;
- *kwd_mbr* is the vector of keyword MBR for all the keywords contained in the node.

Fig. 3.2 depicts an example of an internal node containing three keywords w_1, w_2, w_3 represented as 111. The node also maintains three keyword MBRs for w_1, w_2 and w_3. The keyword MBR of w_i is actually the spatial bound of all the

objects with associated keyword w_i. Leaf nodes contain entries of the form (*oid, loc, bmp*), where

- *oid* is a pointer to an object in the database;
- *loc* represents the coordinates of the object;
- *bmp* is the keyword bitmap.

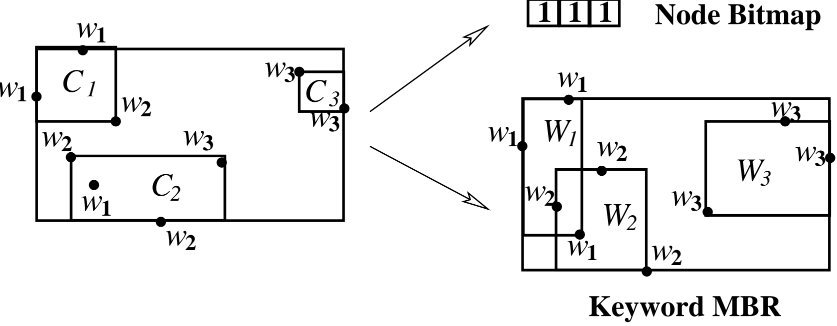


Figure 3.2: Node information of bR*-tree

A bR*-tree can be maintained similar to R*-tree. In R*-tree, insertion works as follows: new tuples are added to leaves, overflowing nodes are split and the changes are propagated upward in the tree. The propagation process is called AdjustTree and the parent node is updated based on the property that its MBR is tightly bound to the MBRs of its child nodes. The bitmap and keyword MBR also have similar properties for convenient information update in the parent node. The set of keywords of the parent node is the union of the sets of keywords in the child nodes. If w_i appears in a child node, it must also appear in the parent node. On the other hand, the keyword MBR of w_i in the parent node is actually the minimum bound of the corresponding keyword MBRs in the child nodes. If the parent node's MBR does not tightly enclose all its child MBRs, or its keywords or keyword MBRs are not consistent with those in the child nodes, AdjustTree is invoked. Hence, we can construct our bR*-tree by means of the original R*-tree algorithm [26] by

adding the operations of updating keywords and keyword MBR when AdjustTree is invoked. In a similar vein, the operations of update and delete in bR*-tree can also be naturally extended from the original implementations.

3.3 Search Algorithms

Suppose a hierarchical bR*-tree has been built on all the data objects. mCK query aims at finding m closest keywords in the leaf entries matching the query keywords. Our search algorithm starts from the root node. The target keywords may be located within one child node or across multiple child nodes of the root. Hence, we need to check all possible subsets of the child nodes. The candidate search space consists of two parts:

- the space within one child node;
- the space across multiple (> 1) child nodes.

If a child node contains all m query keywords, we treat it as a candidate search space. Similarly, if multiple child nodes together can contribute all the query keywords and they are close to each other, then they are also included in the search space.

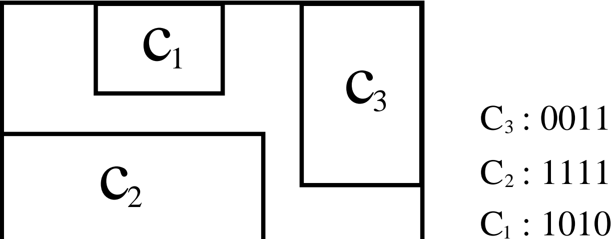


Figure 3.3: An illustration of search in one node

To give an intuition of how the search space looks like, let us look at Fig. 3.3. The node has three child nodes C_1, C_2, and C_3, and they are close to each other.

C_1 is associated with w_2 and w_4, C_2 with all the keywords, and C_3 with w_1 and w_2. Their bitmap representations are also shown in the figure. If the query is 1111, our candidate search space includes the subsets $\{C_2\}$, $\{C_1, C_2\}$, $\{C_2, C_3\}$ and $\{C_1, C_2, C_3\}$. The target keywords may be located in these nodes. $\{C_1\}$, $\{C_3\}$ and $\{C_1, C_3\}$ are pruned because they lack certain query keywords.

After exploring the root node, we obtain a list of candidate subsets of its child nodes. In order to find m closest keywords located at the leaf entries, we need to further explore these candidates and traverse down the bR*-tree. For example, C_2 will be processed in a similar manner to the root node. Subsets of child nodes of C_2 are checked and all those that may possibly contribute a closer result are preserved. The search space for multiple nodes, such as $\{C_1, C_2\}$, is also turned into combinations of subsets of their child nodes. Each combination consists of child nodes from both C_1 and C_2. We can consider this process as node set $\{C_1, C_2\}$ being replaced by subsets of their child nodes and spawning a larger number of new node sets. The number of nodes in the new node set is nondecreasing and their nodes are one level lower in the bR*-tree. If we meet a set of leaf nodes, we retrieve all the combinations of m tuples from the leaf entries and calculate the distance of each tuple set to see if a closer result can be found. Note that during the whole search process, the number of nodes in a node set will never exceed m because our target m tuples can only reside in at most m child nodes. This provides an additional constraint to reduce the search space.

Algorithms 1 and 2 summarize our approach for finding m closest keywords. The first step is to find a relatively small diameter for branch-and-bound pruning before we start the search. We start from the root node and choose a child node with the smallest MBR that contains all the query keywords and traverse down that node. The process is repeated until we reach the leaf level or until we are unable

to find any child node with all the query keywords. Then we perform exhaustive search within the node we found and use the diameter of the result as our initial diameter for searching. Our experiments show that we can find a result of relatively small diameter in a very short time in this manner. We shall henceforth use δ^* to denote the smallest diameter of a result that has been found so far.

With this initial δ^*, we start our search from the root node. Since we are dealing with search in one node or multiple nodes, for the sake of uniformity, we use $NodeSet$ to denote a set of nodes as candidate search space, regardless of the number of nodes in it. The function `SubsetSearch` traverses the tree in a depth-first manner so as to visit the data objects in leaf entries as soon as possible. This increases the chance of finding a small δ^* at an early stage for better pruning. If $NodeSet$ contains leaf nodes, we retrieve all the objects in the leaf entries and exhaustively search for the closest keywords. Otherwise, we apply search strategies according to the number of nodes contained in $NodeSet$. In the following subsection, we discuss these strategies.

Algorithm 1 — Finding m Closest Keywords

Input: m query keywords, bR*-tree
Output: Distance of m closest keywords
1. Find an initial δ^*
2. return `SubsetSearch`($root$)

3.3.1 Searching In One Node

When searching in one node, our task is to enumerate all the subsets of its child nodes in which it is possible to find m closer tuples matching the query keywords. The subsets which contain all m keywords and whose child nodes are close to each other are considered as candidates. There is also a constraint that the number of nodes in a subset should not exceed m. Therefore, the number of candidate subsets

Algorithm 2 — `SubsetSearch`: Searching in a Subset of Nodes

Input: current subset $curSet$
Output: Distance of m closest keywords

```
1.  if curSet contains leaf nodes then
2.      δ = ExhaustiveSearch(curSet)
3.      if δ < δ* then
4.          return δ
5.  else
6.      if curSet has only one node then
7.          setList = SearchInOneNode(curSet)
8.          for each S ∈ setList do
9.              δ* = SubsetSearch(S)
10.     if curSet has multiple nodes then
11.         setList = SearchInMultiNodes(curSet)
12.         for each S ∈ setList do
13.             δ* = SubsetSearch(S)
```

that may get further explored could reach $\sum_{i=1}^{m}\binom{n}{i}$ for a node with n child nodes.

An effective strategy for reducing the number of candidate subsets is of paramount importance as each subset will later spawn an exponential number of new subsets. Incidentally, a priori algorithm of Agrawal and Srikant [17] has been an influential algorithm for reducing search space for combinatorial problems. It was designed for finding frequent itemsets using candidate generation via a lattice structure and has the following advantages:

1. Each candidate itemset is generated once because the way of generating new candidates is fixed and ordered. The k-itemset is joined by two $(k-1)$-itemsets with the same $(k-2)$-length prefix. Therefore, given a candidate itemset, such as $\{a, b, c, d\}$, we can infer that it is joined by $\{a, b, c\}$ and $\{a, b, d\}$.

2. For a k-itemset, we only need to check whether all its $(k-1)$-itemset subsets are frequent in level $k-1$. The cost is $O(n)$. This is due to the a priori property that all nonempty subsets of a frequent itemset must also be frequent.

It is not necessary to check all its subsets at lower levels, the cost of which would be exponential.

In order to take advantage of a priori algorithm, we define two monotonic constraints called **distance mutex** and **keyword mutex**. If a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$ is distance mutex or keyword mutex, then any superset of $\mathcal{N}$ is also distance mutex or keyword mutex and can be pruned.

Definition 3.3 (Distance Mutex)**.** *A node set $\mathcal{N}$ is distance mutex if there exist two nodes $N, N' \in \mathcal{N}$ such that $dist(N, N') > \delta^*$.*

The definition of distance mutex is based on the observation that if the minimum distance between two node MBRs of N and N' is larger than δ^*, then the node set $\{N, N'\}$ does not give a result with diameter better than δ^*. This is obvious because the distance between any two tuples from N and N' must be larger than δ^*. Hence, we have the following lemma.

Lemma 3.1. *If a node set $\mathcal{N}$ is distance mutex, then it can be pruned.*

Proof. If $\mathcal{N}$ is distance mutex, then there exist two nodes $N, N' \in \mathcal{N}$ with $dist(N, N') > \delta^*$. For any m tuples $T_1, T_2, \ldots, T_m$ found in this node set that match m query keywords , we can find at least one T_u from N and T_v from N' because each node has to contribute at least one tuple for the result. Since the distance between T_u and T_v must be larger than δ^*, any candidate set of m tuples has diameter larger than δ^*. □

Lemma 3.2. *Distance mutex is a monotone property.*

Proof. Suppose $\mathcal{N}$ is distance mutex. Then there exist two nodes $N, N' \in \mathcal{N}$ with $dist(N, N') > \delta^*$. Any superset of $\mathcal{N}$ must also contain N and N' and hence must have diameter exceeding δ^*. □

If all the nodes in node set $\mathcal{N}$ are close to each other, we can still take advantage of the stored keyword MBR for pruning. Here, we consider the problem from the perspective of contribution of keywords. Each node in the set must contribute a distinct subset of query keywords and all the contributed keywords constitute a complete set of query keywords. For example, given a set of two nodes N and N' and a query of three keywords 0111, if the closest keywords exist in this set, there are six cases of different contributions of query keywords by N and N'. N contributes one of the query keywords and N' contributes the other two. This generates three cases: $(w_1, w_2w_3), (w_2, w_1w_3), (w_3, w_1w_2)$. If N contributes two and N' contributes one, there are another three cases: $(w_1w_2, w_3), (w_1w_3, w_2), (w_2w_3, w_1)$. If the distance of any two different keywords (w_i, w_j) is larger than δ^*, where w_i is from N and w_j is from N', then the diameters of the six cases above are all larger than δ^*. We say that the node set is keyword mutex. The distance of (w_i, w_j) can be measured by the minimum distance of the two corresponding keyword MBRs. More generally, the concept of keyword mutex is defined as follows:

Definition 3.4 (Keyword Mutex)**.** *Given a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, for any n different query keywords $(w_{q_1}, w_{q_2}, \ldots, w_{q_n})$ in which w_{q_i} is uniquely contributed by node N_i, there always exist two different keywords w_{q_i} and w_{q_j} such that $dist(w_{q_i}, w_{q_j}) > \delta^*$, then $\mathcal{N}$ is called keyword mutex.*

Keyword mutex has properties similar to distance mutex.

Lemma 3.3. *If a node set $\{N_1, N_2, \ldots, N_n\}$ is keyword mutex, then it can be pruned.*

Proof. For any candidate of m tuples $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ matching the query keywords, we want to prove $diam(\mathcal{T}) > \delta^*$. Since each node is required to contribute at least one tuple and $m \geq n$, we can extract n different keywords $\{w_{s_1}, w_{s_2}, \ldots, w_{s_n}\}$,

each w_{s_j} coming from node N_j. According to our definition of keyword mutex, there exist two keywords w_{s_i} and w_{s_j} whose distance is larger than δ^*. Two tuples T_u and T_v in candidate $\mathcal{T}$, associated with w_{s_i} and w_{s_j} respectively, can be found to be located within the two corresponding keyword MBRs with distance larger than δ^*. Therefore, $diam(\mathcal{T}) > \delta^*$ and the node set can be pruned. □

Lemma 3.4. *Keyword mutex is a monotone property.*

Proof. Suppose $\mathcal{N}$ is keyword mutex and $\mathcal{N}'$ is its superset with t nodes. For any t different keywords $\{w_{s_1}, w_{s_2}, \ldots, w_{s_t}\}$ where w_{s_i} is contributed by node N_i, we can find two keywords w_{s_j} and w_{s_k} from nodes N_j and $N_k (N_j, N_k \in \mathcal{N})$, such that $dist(w_{s_j}, w_{s_k}) > \delta^*$. Hence $\mathcal{N}'$ is also keyword mutex.

□

Algorithm 3 — SearchInOneNode: Searching in One Node

Input: A node N in bR*-tree
Output: A list of new NodeSets

1. L_1 = all the child nodes in N
2. **for** i from 2 to m **do**
3. **for** each $NodeSet\ C_1 \in L_{i-1}$ **do**
4. **for** each $NodeSet\ C_2 \in L_{i-1}$ **do**
5. **if** C_1 and C_2 share the first $i-1$ nodes **then**
6. $C = NodeSet(C_1, C_2)$
7. **if** C has subset not appear in L_{i-1} **then**
8. continue
9. **if** C is not distance mutex **then**
10. **if** C is not keyword mutex **then**
11. $L_i = L_i \cup C$
12. **for** each $NodeSet\ S \in \cup_{i=1}^{m} L_i$ **do**
13. **if** S contains all the query keywords **then**
14. add S to $cList$
15. return $cList$

The method for searching in one node is shown in Algorithm 3. First (in line 1), we put all the child nodes in the bottom level of the lattice. The lattice is built

level by level with increasing number of child nodes in the $NodeSet$. In level i, each $NodeSet$ contains exactly i child nodes. For a query with m keywords, we only need to check $NodeSet$ with at most m nodes, leading to a lattice with at most m levels. Lines 5–6 show two sets C_1 and C_2 in level $i-1$ being joined. They must have $i-2$ nodes in common. Lines 7–14 check if any of its subsets in level $i-1$ is pruned due to distance mutex or keyword mutex. If all the subsets are legal, we check whether this new candidate itself is distance mutex or keyword mutex for pruning. If it is not pruned, we add it to level i. In lines 19–22, after all the candidates have been generated, we check each one to see if it contains all the query keywords. Those missing any keywords are eliminated. We do not check this constraint while building the lattice because if a node does not contain all the query keywords, it can still combine with other nodes to cover the missing keywords. As long as it is neither distance mutex nor keyword mutex, we keep it in the lattice.

3.3.2 Searching In Multiple Nodes

Given a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, the search in $\mathcal{N}$ needs to check all the possible combinations of child nodes from each N_i to explore the search space in the lower level. The number of child nodes in the newly derived sets should not exceed m. For example, given a node set $\{A, B, C\}$ where $A = \{A_1, A_2\}$, $B = \{B_1, B_2\}$ and $C = \{C_1\}$. A_i, B_i and C_i are child nodes in A, B, and C, respectively. Assume all the pair distances of child nodes are less than δ^*. All the candidate combinations of child nodes are shown in Fig. 3.1. Every new node set contains child nodes from all the three nodes. If $m = 3$, the candidates are those in the first column. Each query keyword is contributed by exactly one of the child nodes. If $m = 5$, the search space includes all the node sets listed in the figure.

The a priori algorithm can still be applied to this situation. Fig. 3.4 shows the

Table 3.1: Possible sets of $\{A_1, A_2\}$, $\{B_1, B_2\}$, and $\{C_1\}$

3 nodes	4 nodes	5 nodes
$A_1B_1C_1$	$A_1A_2B_1C_1$	$A_1A_2B_1B_2C_1$
$A_1B_2C_1$	$A_1A_2B_2C_1$	
$A_2B_1C_1$	$A_1B_1B_2C_1$	
$A_2B_2C_1$	$A_2B_1B_2C_1$	

lattice to generate candidates for the above node set $\{A, B, C\}$. The sets with child nodes from all three nodes are marked with bold lines. The nonbold nodes cannot be candidates. Given m query keywords, only the bottom m levels of the lattice is built. The properties of distance mutex and keyword mutex are also applicable during generation of the new candidates. The algorithm returns those candidates in the bold nodes, which are neither distance mutex nor keyword mutex. However, this approach creates many unnecessary candidates and incurs additional cost in checking these candidates. For example, if $m = 3$, we know from Fig. 3.1 that there are only four candidate sets that need to be generated. But the a priori algorithm will create a whole level for candidates with three nodes, thereby resulting in ten candidates.

Alternatively, we propose a new algorithm which does not generate any unnecessary candidates, but still keeps the advantages of a priori algorithm. For a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, we reuse the n lists of candidate node sets generated by applying a priori algorithm to search in each node. The ith list contains the sets of child nodes in N_i. The sets are ordered from lower levels in the lattice to higher levels. For example, if N_i has three child nodes $\{C_1, C_2, C_3\}$, the sets of child nodes in the corresponding list may be ordered in the following way: $\{C_1\}, \{C_2\}, \ldots, \{C_1, C_2, C_3\}$. An initial filtering is done on N_i's list by only considering the child nodes that are close to all the other N_j. If C_k in N_i is far away

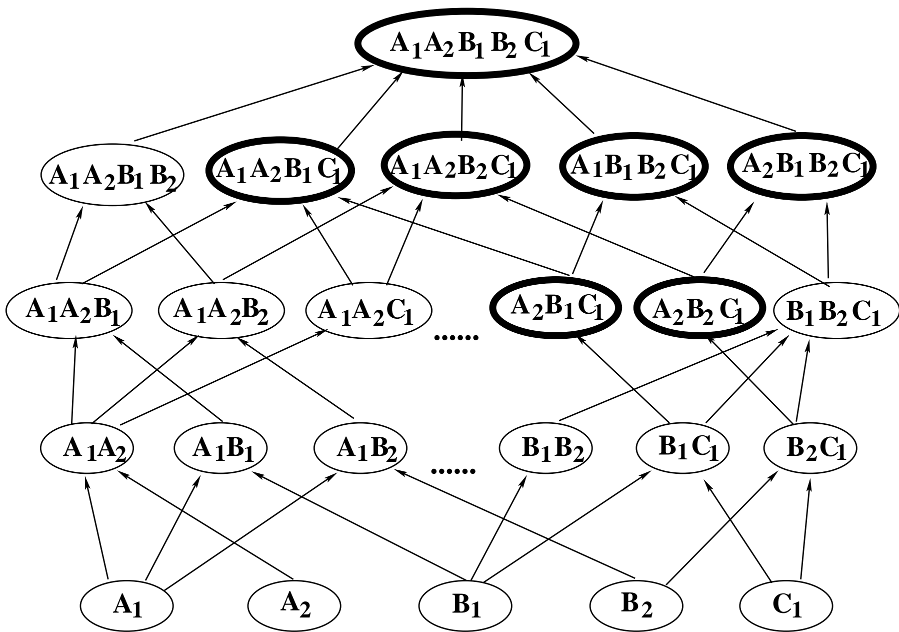


Figure 3.4: a priori algorithm applied to search in multiple nodes

from any other node N_j, all the sets in the ith list containing C_k are pruned.

To generate new candidates, we enumerate all the possible combinations of child node subsets from these n lists. Fig. 3.5 illustrates our approach. At the bottom level, we have three lists of child node subsets from nodes A, B, and C. Combinations of the subsets from these three lists are enumerated to retrieve new candidate sets. As shown in Fig. 3.5, all the nine candidate sets are directly retrieved from the subsets in the bottom level. In this manner, our algorithm does not generate unnecessary candidates. Moreover, the enumeration process is ordered, as shown by the dashed arrows. A new candidate is enumerated only after all of its subsets have been generated. For example, $A_1A_2B_1C_1$ must be generated after $A_1B_1C_1$ and $A_2B_1C_1$ because the subsets of child nodes in each list are ordered by the node number. As a consequence, we can efficiently generate the candidates and still preserve the advantages of a priori algorithm:

1. Each candidate item is generated once. For example, given a candidate item

$\{A_1A_2B_1C_1\}$, we know that it is combined by $\{A_1, A_2\}$, $\{B_1\}$ and $\{C_1\}$. No duplicate candidates will appear in the results.

2. For a k-item, we only need to check its $(k-1)$-item subsets. Since the candidates in each N_i generated by a priori algorithm are ordered, all its subsets must have been examined when we are processing current k-item.

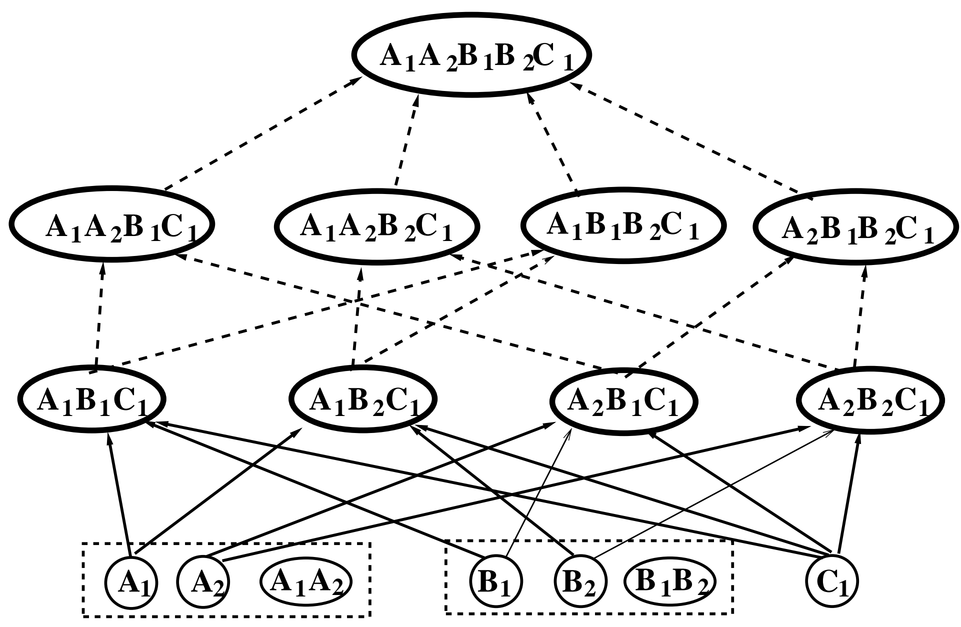


Figure 3.5: Extended a priori algorithm

Algorithm 4 shows how a set of n nodes $\{N_1, \ldots, N_n\}$ is explored. First, n lists of ordered subsets of child nodes are obtained. Then Algorithm 5 is invoked to enumerate all the candidate sets. It is implemented in a recursive manner. Each time an enumerated candidate is generated, we check if it contains all the query keywords to decide whether to prune it or to put it in the candidate list(see Lines 1–4). Lines 5–8 indicate the beginning of the recursion process. It starts from each child node subset in list L_n and makes it as our current partial node set $curSet$. $curSet$ recursively combines with other child node subsets until it finally contains child nodes from $\{N_1, \ldots, N_n\}$. In each recursion, we iterate the child node sets in

list L_i to combine with $curSet$ and generate a new set denoted as $newSet$. Lines 12–13 show that if $newSet$ already has more than m child nodes, we stop the iteration. The child node subsets which are not checked could only have more child nodes and will result in even more nodes in $newSet$. Otherwise, we check if any subsets of $newSet$ have been pruned due to distance mutex or keyword mutex. If not, we continue checking whether this new $NodeSet$ itself is distance mutex or keyword mutex. All these checking functions are shown in Lines 14–17. If $newSet$ is not pruned, we set it as $curSet$ and continue the recursion. Finally, the algorithm returns all the candidates that were not pruned. In the following subsections, we propose two novel methods to efficiently check whether a set is distance mutex or keyword mutex.

Algorithm 4 — SearchInMultiNodes: Search In Multiple Nodes

Input: A set of $\{N_1, \ldots, N_n\}$ in bR*-tree
Output: A list of new NodeSets
1. **for** each node N_i **do**
2. L_i = SearchInOneNode(N_i)
3. perform an initial filtering on L_i
4. return Enumerate($L_1, \ldots, L_n$, n, NULL)

3.3.3 Pruning via Distance Mutex

The diameter of a candidate of m tuples matching the query keywords is determined by the maximum distance between any two tuples. The candidate can be discarded if we found two tuples in it with distance larger than δ^*. Similarly, as we are traversing down the tree, we can eliminate the node sets in which the minimum distance between two nodes is larger than δ^*. A candidate which is not distance mutex requires each pair of nodes to be close. It takes $O(n^2)$ time to check the distance between all pairs of a set of n nodes.

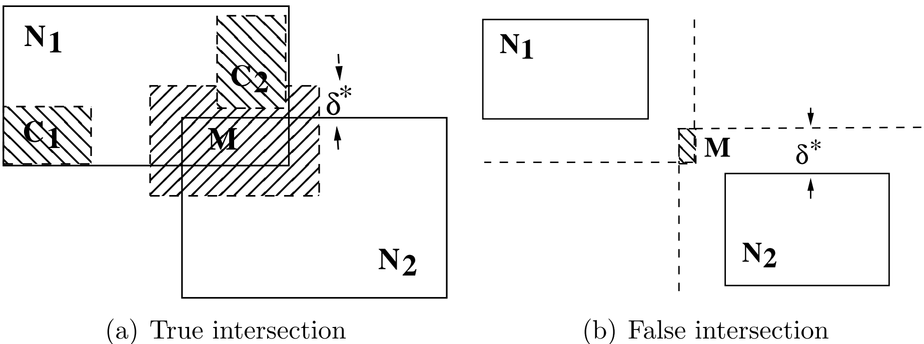
To facilitate more efficient checking, we introduce a concept called **active**

Algorithm 5 — Enumerate: Enumerate All Possible Candidates

Input: n lists of sets of child nodes $L_1, \ldots, L_n$, count and $curSet$
Output: A list of new NodeSets

```
 1. if count = 0 then
 2.   if curSet contains all the query keywords then
 3.     push curSet into the candidate list cList
 4.     return
 5. if count = n then
 6.   for each NodeSet S ∈ L_n do
 7.     curSet = S
 8.     Enumerate(L_1, ..., L_n, count−1, curSet)
 9. else
10.   for each NodeSet S ∈ L_n do
11.     newSet = NodeSet(curSet, S)
12.     if newSet contains more than m nodes then
13.       break
14.     if newSet has any illegal subset candidate then
15.       continue
16.     if newSet is not distance mutex then
17.       if newSet is not keyword mutex then
18.         Enumerate(L_1, ..., L_n, count−1, newSet)
19. return cList
```

MBR. Fig. 3.6(a) illustrates this concept with a set of two nodes $\{N_1, N_2\}$. First, we enlarge these two MBRs by a distance of δ^*, and their intersection is marked by the shaded area M in the figure. We can restrict our search area within area M because any tuple outside M cannot possibly combine with tuples of the other node to achieve a smaller diameter than δ^*. In this example, the child node C_1 does not participate because it does not intersect with M. The objects in C_2 but outside M need not be taken into account as well. We call M the active MBR of N_1 and N_2 because a candidate of m tuples can only reside within the area covered by M. However, we should also check for false intersections, which is shown in Fig. 3.6(b). The intersection actually lies outside both N_1 and N_2. If this happens, the set does not have an active MBR and becomes distance mutex. Hence, we can prune it away.



(a) True intersection (b) False intersection

Figure 3.6: Example of active MBR

When a third node N_3 combines with N_1 and N_2, we only need to check whether N_3 intersects with M, without having to calculate the distance from N_3 to N_2 and N_1. Any tuple outside M is either far away from N_1 or far away from N_2. Therefore, if N_3 does not intersect with M, we can conclude that the set $\{N_1, N_2, N_3\}$ is distance mutex. Otherwise, we update the active MBR for this new set to be its intersection with the enlarged N_3. This property greatly facilitates the checking of distance mutex. When we are checking a new candidate "joined" by two sets C_1 and C_2 in a priori algorithm, we only need to check whether the active MBR of C_1 intersects with that of C_2. Moreover, as more nodes participate in the set, the active MBR becomes smaller and smaller, and is likely to be pruned. This helps to reduce the cost of search by avoiding the enumeration of large number of nodes.

3.3.4 Pruning via Keyword Mutex

A set of n nodes is said to be keyword mutex if for any n different keywords, each from one node, we can always find two keywords whose distance is larger than δ^*. We use the keyword MBR stored in each node to check for keyword mutex. We present a simple example by considering a set of two nodes $\{A, B\}$. Given four query keywords, we construct a 4×4 matrix $\mathsf{M}(A, B) = (m)_{ij}$ to describe the keyword

relationship between A and B: m_{ij} indicates whether tuples with keyword w_i in A can be combined with tuples with keyword w_j in B. If the minimum distance between these two keyword MBRs is smaller than δ^*, then $m_{ij} = 1$; otherwise, $m_{ij} = 0$. If w_i does not appear in A, or w_j does not appear in B, then also $m_{ij} = 0$. Moreover, $m_{ii} = 0$ since each keyword in the mCK result can only be contributed by one node. If $\mathsf{M}(A, B)$ is the zero matrix, we can conclude that the set is keyword mutex. For any two different keywords w_i and w_j from A and B, its distance must be larger than δ^*.

Generally, for a set of $n \geq 3$ nodes $\{N_1, N_2, \ldots, N_n\}$, we define $\mathsf{M}(N_1, \ldots, N_n)$ recursively as follows: for $n \geq 3$,

$$\begin{aligned}\mathsf{M}(N_1, \ldots, N_n) =& (\mathsf{M}(N_1, N_2) \times \mathsf{M}(N_2, \ldots, N_n)) \otimes \\ & (\mathsf{M}(N_1, \ldots, N_{n-1}) \times \mathsf{M}(N_{n-1}, N_n)) \otimes \\ & \mathsf{M}(N_1, N_n),\end{aligned}$$

where $\times$ is the ordinary matrix multiplication, and $\otimes$ is elementwise multiplication. The base case when $n = 2$ has already been defined in the paragraph above.

As the lemma below shows, we need only check whether $\mathsf{M}(N_1, \ldots, N_n) = \mathsf{0}$ to determine if $\{N_1, \ldots, N_n\}$ is keyword mutex.

Lemma 3.5. *If* $\mathsf{M}(N_1, \ldots, N_n) = \mathsf{0}$*, then the set of nodes* $\{N_1, N_2, \ldots, N_n\}$ *is keyword mutex.*

Proof. Suppose $\mathsf{M}(N_1, \ldots, N_n) = \mathsf{0}$ but $\{N_1, \ldots, N_n\}$ is not keyword mutex. Then there must exist n different keywords $k_1, \ldots, k_n$ from nodes $N_1, \ldots, N_n$, respectively, such that all pairs of keywords are at distance less than δ^*. We have $\mathsf{M}(N_i, N_j)_{k_i k_j} =$

1 for $1 \leq i < j \leq n$. First, we prove

$$\mathsf{M}(N_u, \ldots, N_v)_{k_u k_v} \geq \prod_{u \leq i < j \leq v} \mathsf{M}(N_i, N_j)_{k_i k_j} \tag{3.1}$$

by induction on $v - u$

When $v - u = 1$, (3.1) clearly holds. For $v - u > 1$, consider the inequalities:

$$\mathsf{M}(N_u, \ldots, N_{v-1})_{k_u k_{v-1}} \geq \prod_{u \leq i < j \leq v-1} \mathsf{M}(N_i, N_j)_{k_i k_j}$$

and

$$\mathsf{M}(N_{u+1}, \ldots, N_v)_{k_{u+1} k_v} \geq \prod_{u+1 \leq i < j \leq v} \mathsf{M}(N_i, N_j)_{k_i k_j},$$

which hold by the induction hypothesis. Since the matrix entries are all nonnegative, we have

$$\begin{aligned}
&\mathsf{M}(N_u, \ldots, N_v)_{k_u k_v} \\
\geq &\left(\mathsf{M}(N_u, N_{u+1})_{k_u k_{u+1}} \cdot \left(\prod_{u+1 \leq i < j \leq v} \mathsf{M}(N_i, N_j)_{k_i k_j}\right)\right) \cdot \\
&\left(\left(\prod_{u \leq i < j \leq v-1} \mathsf{M}(N_i, N_j)_{k_i k_j}\right) \cdot \mathsf{M}(N_{v-1} N_v)_{k_{v-1} k_v}\right) \\
&\cdot \mathsf{M}(N_u, N_v)_{k_u k_v} \\
\geq &\prod_{u \leq i < j \leq v-1} \mathsf{M}(N_i, N_j)_{k_i k_j}.
\end{aligned}$$

Therefore,

$$\mathsf{M}(N_1, \ldots, N_n)_{k_1 k_n} \geq \prod_{1 \leq i < j \leq n} \mathsf{M}(N_i, N_j)_{k_i k_j} = 1,$$

which is nonzero. This is a contradiction. □

The advantage of the matrix implementation is that it can be naturally integrated into our a priori-based search strategy. When dealing with set $\{N_1, \ldots, N_n\}$, the matrices involved in the above formula will already have materialized in most cases. Therefore, checking for keyword mutex requires only two matrix multiplications and two matrix elementwise products, which can be achieved at a low computation cost.

3.4 Empirical Study

This section provides an extensive performance study of our query strategy using one bR*-tree to integrate all the spatial and keyword information. We use MWSJ approach [89] as reference which works in this way: If there are N keywords existing in the spatial database, N separate R*-trees are built. Given m query keywords, we pick m corresponding R*-trees $T_1, T_2, \ldots, T_m$. The trees are ordered by the number of objects in the tree. The search process starts from the smallest R*-tree T_1 with fewest objects. For any leaf MBR M_1 in T_1, we search the leaf MBRs in T_2 that are close enough to M_1. The idea of active MBR can be applied to speed up the search. In T_3, the search space has been shrunk to the active MBR of M_1 and M_2. Only MBRs intersecting with this active MBR will be taken into account. This process lasts until all the leaf MBRs near M_1 in all other R*-trees have been explored. Then, we move to other leaf MBRs in T_1 until all the combinations of objects in each R*-tree have been explored completely. Such an implementation outperforms the traditional top-down strategy used in answering closest-pair queries [43].

We implement both algorithms in C++ using its standard template library. Our bR*-tree is implemented by extending the R*-tree code [2]. All the experiments are conducted on a server with Intel Xeon 2.6GHz CPU, 8GB memory, running

[2] http://research.att.com/∼marioh/spatialindex/

Ubuntu 7.10. Both synthetic and real life data sets are used for performance testing. We use average response time (ART) as our performance metric: ART = $(1/N_Q)\sum_{i=1}^{N_Q}(T_f - T_i)$, where T_i is the time of query issuing, T_f is time of query completion, and N_Q is total number of times the given mCK query was issued. Note that ART is equivalent to the elapsed time including disk I/O and CPU-time.

3.4.1 Experiments on Synthetic Data Sets

The synthetic data generator generates spatial data points in a random manner. Each point is randomly distributed in the d-dimensional space $[0, 1]^d$ and assigned a fixed number of random keywords. We fix the number of keywords on each data point so that it is more convenient to analyze the performance when a data point is associated with multiple keywords. In our implementation of bR*-tree, the page size is set to 4K bytes and the maximum number of entries in internal nodes is set to 30. However, the number of entries in leaf nodes is set to be the same with total number of keywords to allow flexibility in handling different number of keywords. In the implementation of MWSJ, we also use a page size of 4K and set the maximum number of entries in all nodes at 30. bR*-tree takes more time than MWSJ in building the index for two reasons. First, in MWSJ, each tuple is inserted into a small R*-tree with the same keyword as the tuple. In bR*-tree, each tuple is inserted into the whole tree. This results in much higher cost for each insertion, including choosing a leaf, invoking more split and AdjustTree operations. Second, bR*-tree maintains additional information, such as keywords and keyword MBR, which need to be updated during insertions.

In the following experiments, we adjust four parameters to generate different data sets. The parameters are

- TK : total number of keywords in the database;

- DS : data size;
- DM : the dimension of data point;
- KD : the number of keywords associated with each data.

In each experiment, we compare the performance of bR*-tree with MWSJ on different synthetic data sets using ART as the performance metric.

Effect of TK

We ran the first experiment on four data sets to test scalability in terms of the number of query keywords m. We generated data sets with total number of keywords 50, 100, 200 and 400, respectively. Each data is two-dimensional and associated with one keyword. In each data set, there are 3,000 data points associated with the same keyword.

Fig. 4.6 shows the ART of two algorithms with respect to the number of query keywords. When m is small, we can see that MWSJ outperforms bR*-tree and this advantage becomes clearer as total number of keywords increases. It only accesses m of total N R*-trees that occupy a small portion of the whole data set. The query can be processed relatively quickly. However, our search process needs to access all the nodes in the entire bR*-tree because the data with different keywords are randomly distributed in the leaf nodes. This results in relatively poor performance as compared to that of MWSJ.

As m increases to large values, the performance of MWSJ starts to degrade dramatically. The search space is expanded exponentially and MWSJ incurs high disk I/O cost for identifying the candidate windows since it does not inherently support effective summarization of keyword locations. However, our algorithm

demonstrates remarkable scalability as m increases[3]. Our bR*-tree summarizes the keywords and their locations in each node, and this plays an important role in effectively pruning the search space. The a priori-based search strategy also restricts the candidate search space from growing too quickly.

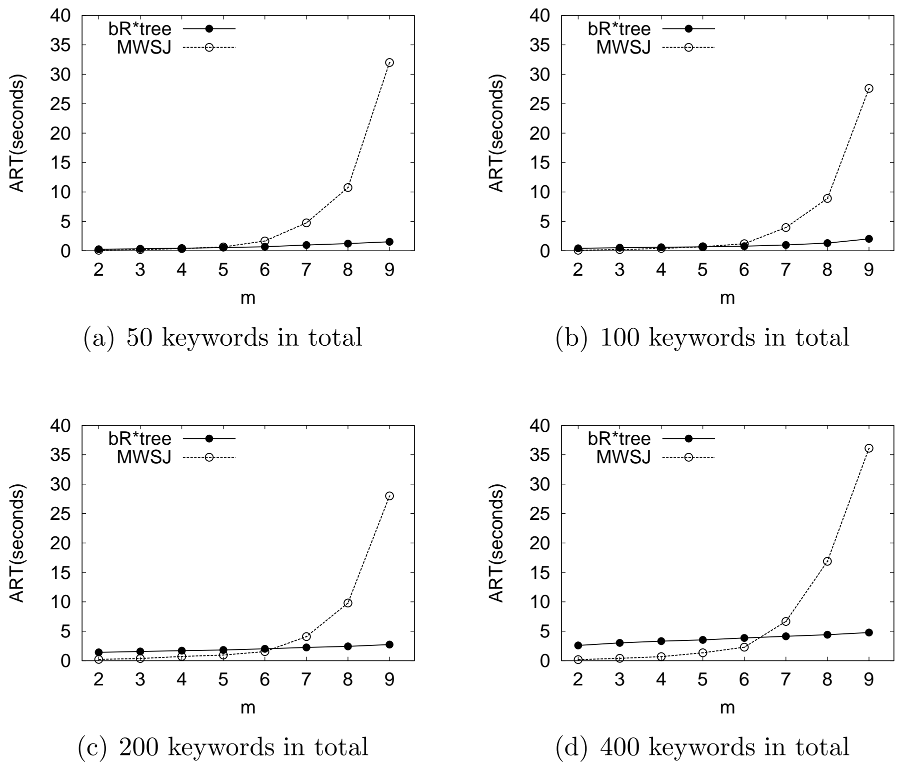
Note that the overall performance trend of MWSJ across the four data sets is similar. The reason is that the data sets have the same number of data points associated with each keyword and the size of R*-tree is the same. However, the performance of bR*-tree degrades slightly with the increase of data size and total number of keywords. It integrates all the data points in one tree, leading to a higher access cost.

Effect of DS

In the above experiment, the number of data points associated with each keyword is fixed. In this experiment, we fix total number of keywords at 100 and increase data size from 100,000 to 3,000,000 to examine the performance of bR*-tree and MWSJ.

Fig. 3.8 shows how ART increases with data size in answering the same number of query keywords. When m is small, e.g. $m = 3$ and $m = 5$, both algorithms demonstrate similar rate of increase in ART. The spatial index and the pruning using active MBR did take effect to suppress the expansion of search space caused by the increase of data size. However, when m becomes large, e.g. $m = 7$ and $m = 8$, a small amount of increase in the size of the R*-tree in MWSJ can lead to a remarkable increase in the search space. We can observe from Fig. 3.8 that MWSJ becomes sensitive to the increase of data size and the performance declines dra-

[3]Note that this make our algorithm particularly useful for purpose like geotagging of documents where a mCK query with large number of keywords are issued by an automatic search algorithm. In addition, for systems in which the number of keywords in a submitted query can varies greatly, our approach will provide very stable performance compared to MWSJ.



(a) 50 keywords in total

(b) 100 keywords in total

(c) 200 keywords in total

(d) 400 keywords in total

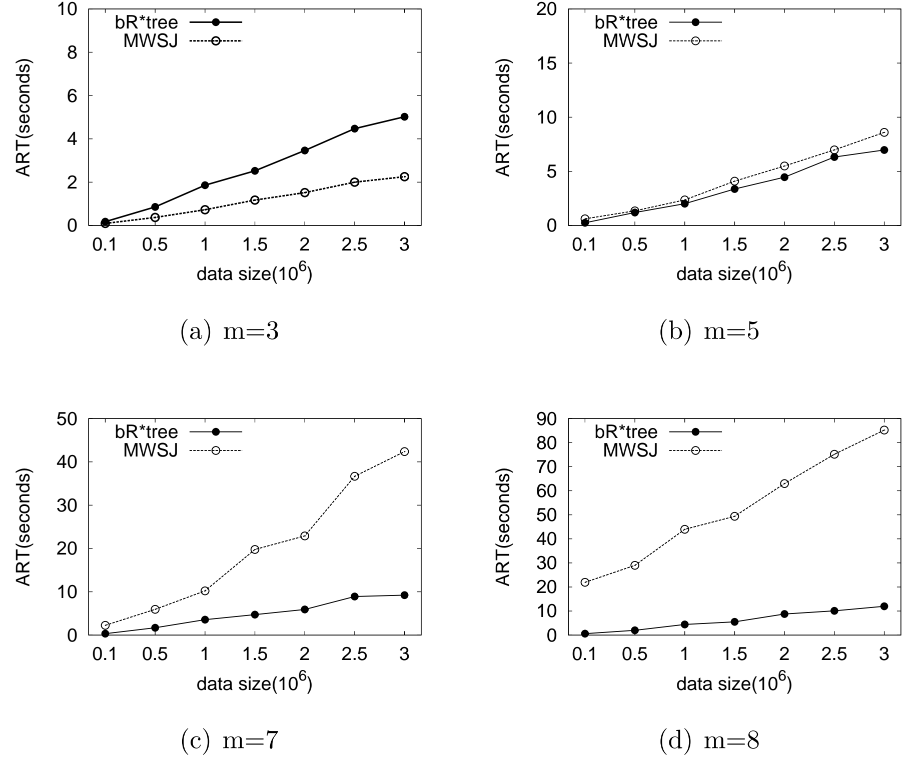
Figure 3.7: Performance on increasing TK

matically especially when data size is large. In contrast, bR*-tree scales smoothly in a stable manner, thereby validating the effectiveness of our search strategy.

Effect of KD

In many applications, a spatial object is associated with a set of keywords rather than only one keyword. Under a fixed data size, if we increase the number of keywords associated with each data point, the search space increases as well. For each keyword, there are more data points associated with it, and hence, a larger R*-tree is needed for indexing in the case of MWSJ. However, the size of bR*-tree is not affected because the bitmap in the node only gets more bits set, but still incurs a fixed storage cost.

In this experiment, we generate 1,000,000 two-dimensional data points. There



(a) m=3

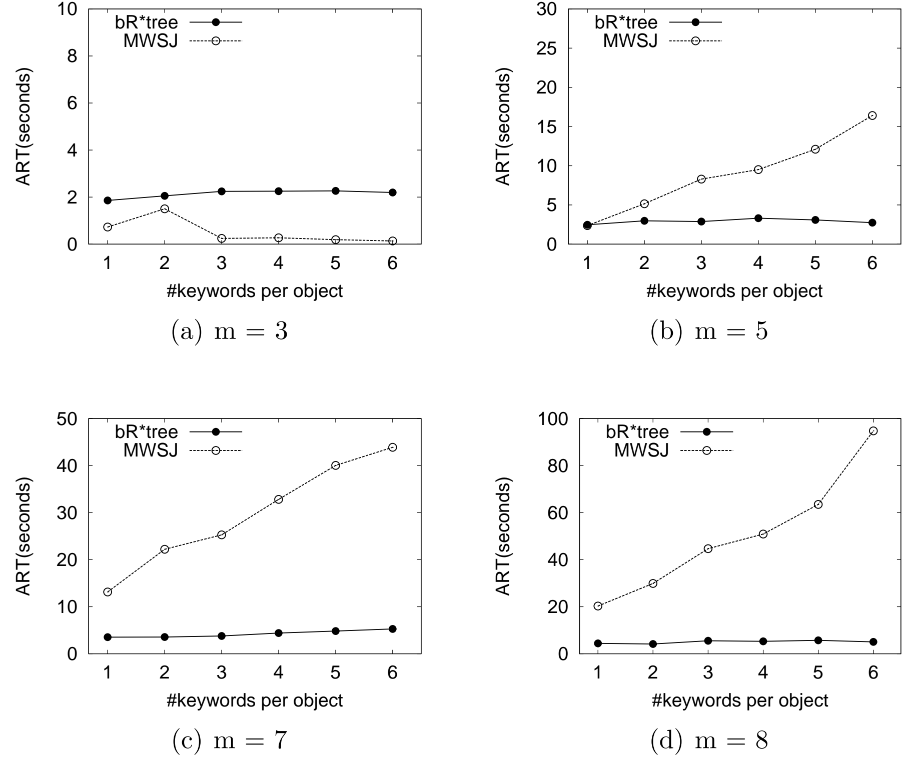
(b) m=5

(c) m=7

(d) m=8

Figure 3.8: Performance on increasing DS

are a total of 100 keywords in the data set. We increase the value of KD from one to six. The results, depicted in Fig. 3.4.1, show that bR*-tree always demonstrates good stability when KD increases. However, MWSJ suffers from serious performance degradation as KD increases because it does not inherently support effective summarization of keyword locations. Note that when $m = 3$, the performance of MWSJ has a sudden improvement when $KD = 3$, i.e. each object is associated with three keywords. An object with all three query keywords is very likely to be found in the data set giving $\delta^* = 0$. This greatly facilitates the pruning in the unexplored search space. When $m = 5$, this improvement is not shown clearly because the probability of finding an object with all the query keywords in the early search stage is low.



(a) m = 3 (b) m = 5

(c) m = 7 (d) m = 8

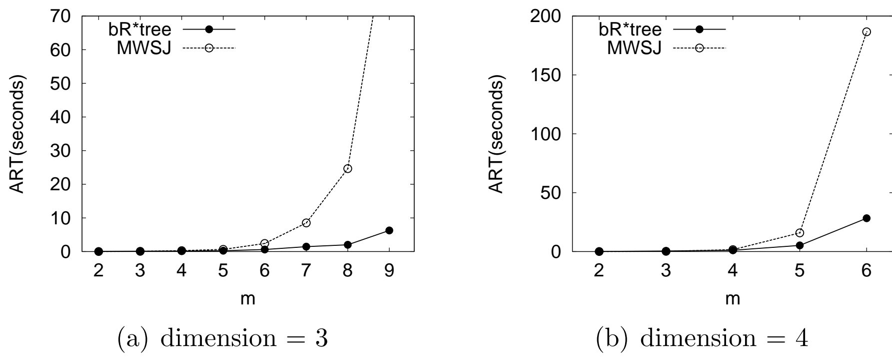
Figure 3.9: Performance on increasing KD

Effect of DM

In the above experiments, we only handle two-dimensional data. In some applications, the data may have multiple attributes and are mapped to higher-dimensional space. For example, a notebook may be mapped to a five-dimensional value arising from attributes such as CPU, memory, hard disk, weight and price. The closest notebooks from different manufacturers may be serious competitors in the market. Therefore, it is meaningful to test how the algorithms perform on higher dimensional spaces.

We test the performance on three- and four-dimensional data with a small data size of 50,000. There are 100 keywords in total and each data is associated with one keyword. The ART results are shown in Fig. 3.10. It is clear that MWSJ

performs poorly on higher dimensional data set because its pruning is based on only the distance constraint. Our bR*-tree takes advantage of both distance and keyword constraints of mCK query for pruning and shows much better scalability. As m increases, the performance of MWSJ rapidly declines and can be orders of magnitude worse than bR*-tree.



(a) dimension = 3 (b) dimension = 4

Figure 3.10: Performance on increasing DM

3.4.2 Experiments on Real Data Set

We use TIGER (Topologically Integrated Geographic Encoding and Referencing system) (downloadable from *http://www.census.gov/geo/www/tiger*) as our real data set. The data set consists of numerous complicated geographic and cartographic information of the entire United States. Since we are concerned with point data in our mCK query, we simply extract the landmark data, which can be custodial facility (hospitals, orphanages, federal penitentiaries, etc.), educational, cultural or religious institutions, etc. Each point in the data set is associated with a census feature class code to identify its noticeable characteristic. For example, *D85* is the class code for keyword *Park*.

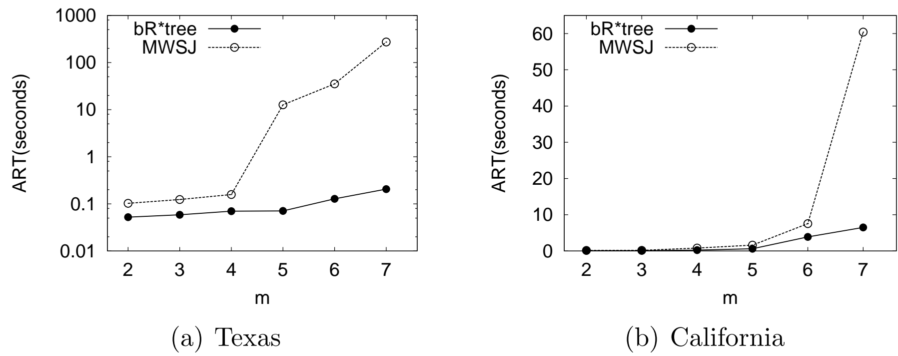
After cleaning and format transformation on the raw data, we extracted two data sets, Texas and California, with 15,179 and 13,863 data points, respectively.

Table 3.2: Keyword distribution on Texas data set

D1	4	D28	206	D43	1956	D71	75
D10	3	D29	1	D44	6092	D73	1
D20	23	D31	266	D51	364	D81	177
D21	872	D32	3	D53	2	D82	3291
D22	2	D33	34	D61	929	D83	21
D23	167	D35	6	D62	21	D84	1
D24	2	D36	17	D63	21	D85	295
D25	20	D37	5	D64	21	D90	78
D26	26	D41	2	D65	120		
D27	44	D42	2	D66	9		

Both data sets have dozens of keywords. The distribution for each keyword is highly skewed. Table 3.2 shows the keyword distribution in Texas. Some landmark may get thousands of points while others may have only one data point. For example, *D43* represents educational institutions, including academy, school, college and university. These institutions are widely distributed and are well recorded in the raw data set. However, landmarks like water tower (*D71*) are a rarity and only one such landmark appears in our extracted data set.

In our experiments, we ignore infrequent keywords and submit queries with the most frequent keywords. Fig. 3.11 shows the ART with respect to the number of query keywords in both data sets. We can see that bR*-tree outperforms MWSJ even when m is small. The reason is that the number data associated with each keyword is highly skewed. When a query has frequent keywords, MWSJ loses the advantage of having to access only a small portion of the data set. When m increases to large values, its performance still degrades dramatically. Our bR*-tree not only answers the frequent keywords query in a shorter time, but also exhibits good scalability. Therefore, bR*-tree performs significantly better than MWSJ in answering queries with frequent keywords.



(a) Texas

(b) California

Figure 3.11: Performance on two real data sets

3.5 Summary

In this chapter, we addressed mCK query, which aims at retrieving m closest travel services. We used keyword to represent travel service and proposed bR*-tree to effectively summarize keyword locations, thereby facilitating pruning. We also proposed effective a priori-based search strategy for mCK query processing. Two monotone constraints and their efficient implementations were discussed. Our performance study on both synthetic and real data sets demonstrates that the proposed bR*-tree answers mCK queries efficiently within relatively short query response time. Furthermore, it demonstrates remarkable scalability in terms of the number of query keywords. It significantly outperforms existing MWSJ approach when m is large.

CHAPTER 4

LOCATING WEB RESOURCES BY SPATIAL TAG MATCHING

In this chapter, we focus on the fundamental application of locating geographical web resources and propose an efficient tag-centric query processing strategy. The result is a location that is most relevant to the query tags. In particular, we aim at finding a set of nearest co-located travel objects which together match the query tags. Given the fact that there could be a large number of travel objects and tags, we develop an efficient search algorithm that can scale up in terms of the number of objects and tags. Further, to ensure that the results are relevant, we also propose a geographical context sensitive *geo-tf-idf* ranking mechanism. Our experiments on synthetic data sets demonstrate its scalability while the experiments using a real life data set confirm its practicality.

4.1 Introduction

In Web 2.0, users are free to upload diverse kinds of travel resources and mark them in the map to indicate their relevance to the area using open map APIs. Such a large amount of user-contributed materials constitute a luxuriant spatial database that can provide immense mining opportunities. Effective location detecting technique has significant commercial potential and can assist the search engine in classifying and indexing the web resources to improve the relevance of the returned results. It also plays an important role in providing the customers with local personalized services. Existing work tackles this problem by mining and extracting phrases that contain geographical context in web documents or with the aid of hyperlink structures and query logs when the geographical context is not clear. The ambiguities on the location names are eliminated via NLP or IR technique to assign the correct scope so that terms such as "Washington" appearing in "Denzel Washington" will not be treated as a location name. Despite considerably high accuracy, traditional methods are still faced with new challenges in the Web 2.0 environment:

- Various types of resources exist in the spatial database. Existing search engines pay particular attention to gazetteer terms derived from web documents. However, other multimedia resources, such as photos and videos, are not associated with such terms inherently. Without geographical terms, traditional approaches may not work well.

- Current geographic information systems typically rely on the gazetteer information published by authorized communities. In Web 2.0, users have contributed huge amounts of useful contents collaboratively. A prominent example is Wikipedia, the most widely used online encyclopedia. There are abundant implicit geographic information embedded and they should be fully

exploited.

In Chapter 1, we introduced a uniform data model for travel resources based on tagging. Based on this model, an example blog is shown in Figure 1.5. The blog is associated with user-generated tags and the gazetteer terms in the text body are shown in bold. We can see that “Philippines” and “Disneyworld” are noisy terms and not related to the travel destination. This brings challenges to traditional methods utilizing NLP or IR technique. In this case, we can take advantage of the associated tags which are of higher quality because noisy terms have been filtered by human intelligence. The problem now becomes finding a location in Singapore that best matches the four tags “beach”, “island”, “Palawan” and “wooden bridge”.

Another locating example is shown in Figure 1.6. In this example, the query is an image from Flickr with annotations. In order to detect the location of the image, we can use “bull”, “bronze” and “sculpture” to query the spatial database in New York City and find the best match location. This is similar to locating the travel blog in the above example. Since we have proposed a uniform mapped resource model in Chapter 1, the problem of locating mapped resources is essentially a spatial tag matching problem.

Finding co-locating tags in spatial databases remains an ongoing research problem. Traditional approaches of keyword search in spatial databases [57, 50, 42, 38] are seeking for a mapped resource matching all the query tags. However, the number of tags associated with each object is typically small, making it difficult to find a complete match. On the other hand, these methods ignore the fact that spatially close resources could belong to the same object and are related to each other. For example, news about “New York City airplane river crash” could be marked by users around the crash location in the Hudson River and photos of “Statue of Liberty” are likely to be uploaded around the Liberty Island. Therefore, instead of

looking for one-to-one match, we allow one query object to match multiple spatially correlated objects as long as the union of their tags can match all the query tags.

The search algorithm in Chapter 3 shows good scalability in terms of the number of query keywords. However, the proposed bR*-tree indexing structure requires the storage of auxiliary information for each possible tag and can therefore potentially incur high I/O cost when the tag space $\mathcal{T}$ is large. In this chapter, we present a new and efficient index based on the R*-tree [26] and inverted list. A labelled R*-tree is constructed to provide spatial proximity information and the inverted list is used as a partition of the tag space $\mathcal{T}$. The augmented summary information is not stored but built dynamically during the search process to make the index light-weight. We design a bottom-up search algorithm to utilize the inverted index so that only the related lists will be accessed. Since the I/O cost has been greatly reduced, the new indexing and searching technique can ensure scalability in terms of both the number of query tags m and the data size within a large tag space $\mathcal{T}$.

In addition to the efficiency issue, we also address the issue of semantic relevance by proposing a re-ranking mechanism for co-located tags that are found within the top-k closest scope. To this end, we have to take into account the geographical context in the ranking process. There exist works in [47, 24] that proposed geo-ranking mechanism using local popularity of web resources measured by citations. However, the hyperlink structure among the resources is not available in our data model. In the recent works of [42, 38], Cong et al. proposed to retrieve the most relevant spatial web objects by considering both the distance proximity and text relevance in the ranking function. However, the text relevance is still between query keywords and each single spatial web document. In this chapter, we present a more general ranking method that takes nearby resources into account as well. We extend the widely used *tf-idf* and propose a new ranking strategy, named *geo-tf-idf*,

to measure how the tags are related to the area that they are located in.

In summary, the main contributions of the work include:

- We propose to use tags to build a general data model. Based on the model, we describe a system framework to support co-location searches on various types of resources in Web 2.0 applications.

- We develop an efficient indexing and searching strategy, which is scalable in terms of both the number of query tags and the data size of resources, to answer the queries of co-located tag matching.

- We extend the widely accepted *tf-idf* method for the geographical context, called the *geo-tf-idf* ranking method, to measure the relevance of the geo-tags with respect to the area in which they are located.

- We conduct extensive experiments using synthetic and real life data sets. The results confirm the effectiveness and practicability of our proposal in the context of Web 2.0 applications.

The remaining of this chapter is organized as follows. Section 4.2 introduces the improved index and search strategy. Section 4.3 proposes the *geo-tf-idf* ranking mechanism. Section 4.4 presents our performance study on the synthetic and real life data sets. Finally, the chapter is concluded in Section 4.5.

4.2 Spatial Index and Search Algorithm

4.2.1 Light-weight Index Structure

In Web 2.0 context, users can continuously contribute new resources to the map. On one hand, there will be increasing number of locations. On the other hand, in each

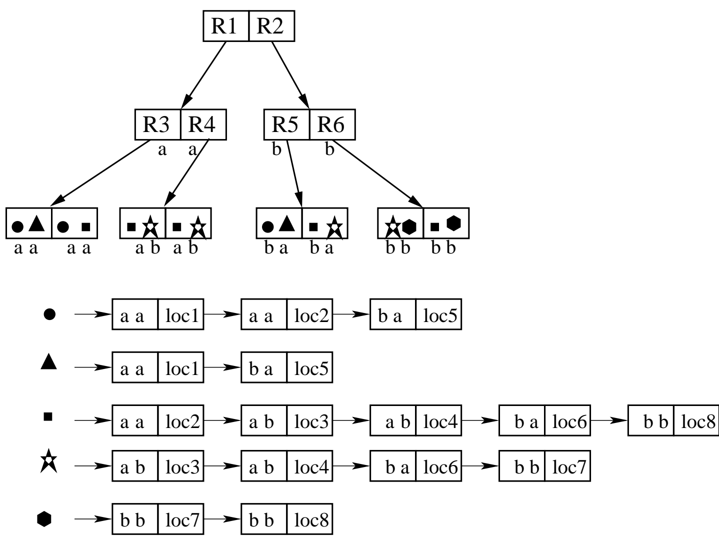


Figure 4.1: The index of R*-tree and inverted index

location, the number of associated tags will increase as well. Thus, it is essential for the proposed index structure to be scalable enough to handle large number of locations and tags. To achieve this goal, we do not maintain the additional summary information. Moreover, we do not integrate all the locations and tags into one tree structure. Instead, we split them into two components: a spatial index and an inverted index, as shown in Figure 4.1. This also ensures that the proposed index could be grafted into existing commercial systems easily.

The R*-tree is used to index all the spatial locations associated with tags. It is constructed in the same manner as described in [26] except that each node is assigned a label indicating the path to the root node. In our example, node R_3 and R_4 are both labelled as "a" because they share the same path to the root node's first entry. Similarly, R_5 and R_6 are labelled with "b", which represents the second entry in the root node. Given a node label, we can judge where the node is located in the R*-tree without accessing the tree. Two locations close to each other probably have the same prefix of node label. If they lie in the same internal node, they will be assigned the same label. Thus, the label can be used to approximate

the spatial distance between the data points.

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 node label derived from the construction of R*-tree and its actual location. Note that the list is ordered by the location label so that the data points close to each other in the geographical space are probably still close in the inverted lists.

Such an index is scalable in terms of both the number of locations and tags. Each time a new location is marked in the map, it is inserted in the labelled R*-tree. The function *ChooseSubtree* in [26] is first invoked to find a leaf node to accommodate the location. If there exist empty entries in the node, the location point is inserted and assigned with the node's label. The inverted lists of the associated tags can also be updated at a small cost as the elements have been ordered. Otherwise, split occurs in the overflowing node and the changes are propagated upward the tree. Besides the MBR adjustment, we need to update the label of nodes as well as the elements in the inverted lists assigned with the old label. Suppose the propagation stopped at node N labelled $l_1 l_2 ... l_t$, all the descendent nodes of N will be re-labelled. Meanwhile, the elements in the affected inverted index whose labels start with $l_1 l_2 ... l_t$ are also updated. Since the lists have been ordered, it is convenient to retrieve the list segment with this prefix. However, if the insertion occurs frequently, ensuring correct label will be computationally expensive. As we do not require accurate labelling in our search algorithm, we can adopt a lazy approach in which we will delay the update of the labels. The location's label is buffered before the split operation. The affected part of R*-tree will be updated in a batch manner using the buffered information to ensure an acceptable cost. The case of inserting a new tag is much simpler. A new inverted list is created for the new tag and then its location and label can be inserted into the list.

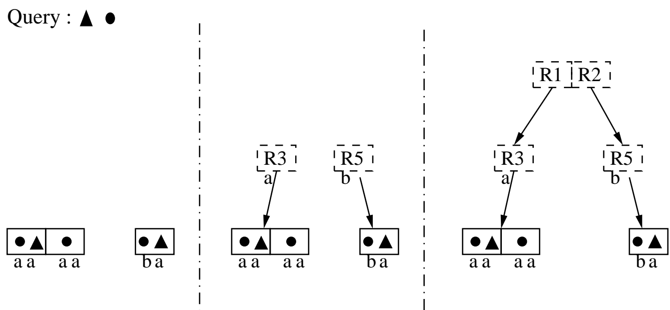


Figure 4.2: Bottom-up construction of virtual bR^*-tree

In contrast to the index in Chapter 3, such a light-weight index saves a large amount of I/O cost compared to indexing all the tags and locations into one bR*-tree. Given a set of query tags, only the relevant location lists are retrieved. The following subsection will introduce how mCK query can be answered on top of inverted lists without even accessing the spatial index. The index can also be utilized to answer queries in which the user specifies a bounding region. We traverse down the R*-tree as much as possible while ensuring that the node MBR bounds the query region completely. Using the label of such a node, we filter off all data points that are not prefixed with this label. The retrieval cost is small as the labels in the lists have been ordered. Then, a further check is performed to see if the data point is within the query region so as to obtain the correct result.

4.2.2 Bottom-Up Search Algorithm

Given m query tags, we retrieve m lists of data points that match the tags from the inverted index. A naive solution to this problem is to exhaustively examine all possible sets of m tuples from different lists. This is prohibitively expensive when the number of objects and/or m is large. To take advantage of the spatial information embedded in the label, we instead propose an elegant solution by constructing a virtual bR*-tree using the label and location of the data points.

The virtual bR*-tree is built level by level in a bottom-up manner as illustrated in Figure 4.2. At first, m inverted lists corresponding to the query tags are retrieved and merged into one list ordered by the node label. The cost of this merge sort is linear to list size. We traverse the sorted list and fetch all the data points with the same label. These points are used to construct a new virtual node which has a counterpart in the original R*-tree. Note that the MBR of the virtual node is much smaller than its counterpart as it is built on the points relevant to the query. For each virtual node, we maintain the additional information: the keyword bitmap and the keyword MBR to summarize the keywords and their distribution inside the virtual node. Compared to the bR*-tree proposed in Chapter 3, the node size has been greatly reduced to save the I/O cost and allows virtual bR*-tree to handle a database with a massive number of possible tags. In addition, the a priori-based search strategy can still be applied in the virtual bR*-tree.

Algorithm 6 Bottom-Up Search Strategy

Input: m query tags, inverted index
Output: Distance and location of m closest keywords

```
1.  Retrieve m inverted lists for each query tag
2.  Merge the lists into one list L ordered by the label
3.  Initialize a virtual node cur_node
4.  while L.level < tree.height do
5.     for each element vnode in L do
6.        if vnode has the same label with its previous element then
7.           add vnode into cur_node
8.        else
9.           SubsetSearch(cur_node)
10.          add cur_node to List L'
11.    move L' to L
```

The detailed search algorithm is shown in Algorithm 6. Each time a virtual node is constructed, it will be treated as a subtree and the pruning algorithm `SubsetSearch` proposed in Chapter 3 could be applied in this virtual node. The difference is that the search space will exclude the single child node that matches

all the query tags. As our search strategy is bottom up, the space within the node must have been explored. Fig 4.3 shows the order of $NodeSet$ candidates checked in the top-down and bottom-up search algorithm respectively. The bottom-up strategy can access the leaf nodes earlier than top-down method and get a smaller δ^* first. In overall summary, the bottom-up search demonstrates better scalability because it only accesses a small virtual bR*-tree and in the meanwhile preserves the effective pruning strategy.

Order	1	2	3	4	5	6
Top-down	R_1	R_3	R_2	R_5	R_1R_2	R_3R_5
Bottom-up	R_3	R_5	R_1R_2	R_3R_5		

Figure 4.3: Order of $NodeSet$ candidates checked

4.3 Ranking

The results returned by mCK search algorithm only consider the spatial closeness while ignoring the geographical relevance. In this section, we propose a new ranking mechanism, namely *geo-tf-idf*, which extends the classic *tf-idf* ranking to be applied in a geographical context.

tf-idf [25, 102, 78] has been widely adopted in search engines to measure the importance of a keyword with respect to a document in a collection or corpus. Intuitively, $score(k, D)$ will be assigned a higher value if keyword k occurs frequently in document D and infrequently in other documents. Formulae 4.1-4.5 shows the ranking mechanism with normalization of document length and frequency taken into account:

$$score(Q,D) = \sum_{k \in Q} weight(k,Q) * score(k,D) \quad (4.1)$$

$$score(k,D) = \frac{tf(k,D)}{dl} * idf \quad (4.2)$$

$$tf(k,D) = 1 + ln(1 + ln(freq(k,D))) \quad (4.3)$$

$$dl = (1-s) + s * \frac{dl(D)}{avgdl} \quad (4.4)$$

$$idf = ln\frac{N}{df+1} \quad (4.5)$$

The term weight of k with respect to Q is usually measured by the raw term frequency in Q. The documents with higher ranking scores will be considered as more relevant to the keywords. Similarly, we can define our score function of a geographical area R with respect to query Q, as shown in Formula 4.6.

$$score(Q,R) = \sum_{k \in Q} weight(k,Q) * score(k,R) \quad (4.6)$$

The problem becomes how to measure the importance of a tag k with respect to an area R. Inspired by the intuition behind *tf-idf*, we propose an extended ranking mechanism used in geographical context. A higher score will be assigned to $score(k,R)$ if the tag t and area R satisfy the following properties:

1. The tag t appears frequently around the area of R. For example, tourist travelling in Beijing will probably take a visit to Forbidden City. There will be many photos and blogs tagged with "Forbidden City" uploaded in the map. Thus, this tag is closely related to Beijing city.

2. The tag t is not frequently mentioned in areas other than R. Although

"Forbidden City" may appear in other travel blogs not located in Beijing, such cases are typically rare. Beijing will be assigned with a high score with respect to "Forbidden City".

In IR systems, the information unit is a document. While in our uniform data model, the concept of document is vague. Each location point is associated with a set of tags. Distinct resources located at the same point can constitute a large virtual document with the tags merged. If there are no resources in the nearby area, we can apply traditional *tf-idf* to measure the weight between keyword k and location p:

$$inw(k, p) = score(k, D) \tag{4.7}$$

,where D is the merged tag "document" located at point p. However, in real Web 2.0 applications, the resources are contributed and uploaded by users. The resources on the same topic will gather around the actual location. The score value on a point location p can not completely capture the geographical context. We need to take into account nearby resources.

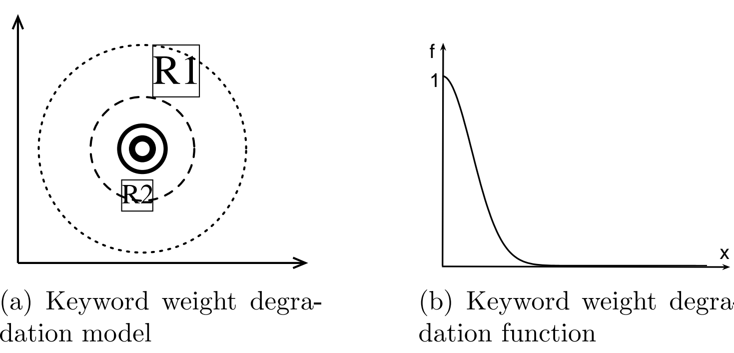
In order to measure the effect of tag t in location p to its nearby areas, we build a degradation model as shown in Figure 4.4(a). The keyword will affect mainly the nearby regions. The regions far away are considered irrelevant to the tag. We may use Gaussian function in Figure 4.4(b) to describe such degradation. Suppose the d-dimensional space has been normalized into $[0, 1]^d$, the relevance of keyword k located at p with respect to area R can be measured via the following formula:

$$score(k, p, R) = \frac{\int_{q \in R} \{inw(k, p) * f(dist(p, q))\}}{1 + ln(area(R))} \tag{4.8}$$

, where f is the degradation function. The intuition behind the definition is that if the area is small and close to the keyword, a higher score will be assigned. If

the area is large, the effect of the keyword on the whole region will be scaled down accordingly. Thus, R_2 is more relevant to k than R_1 in example of Figure 4.4(a). If there are multiple occurrences of keyword k in the spatial database SD, the final score will sum up all the effect of k on the area of R:

$$score(k, R) = \sum_{p \in SD} score(k, p, R) \tag{4.9}$$



(a) Keyword weight degradation model

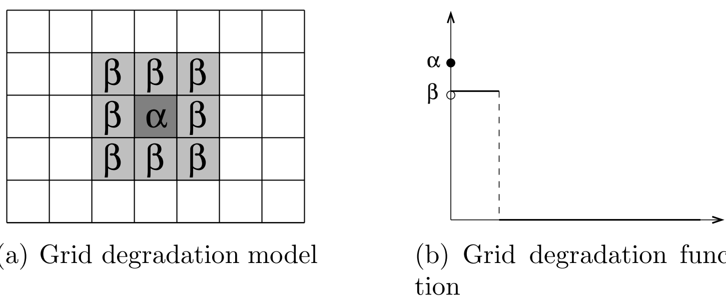
(b) Keyword weight degradation function

Figure 4.4: Degradation of keyword spatial importance

4.3.1 Approximate Ranking Mechanism

In real applications, it is expensive to calculate the exact weight of a region with respect to a keyword. To save the computation cost, we propose an approximate scoring mechanism.

In our approximation model, the space is split into grid cells and region R is approximated by a minimum set of cells that can bound it. The degradation function no longer decreases continuously to 0. Instead, as shown in Figure 4.5(b), we use the grid cell as basic unit. The weight of the grid that holds point p is assigned with constant α and the neighboring grids are assigned with constant $\beta(1 \geq \alpha > \beta > 0)$. The remaining grids are considered not relevant to the keyword. The weight of keyword k with respect the cell C becomes:



(a) Grid degradation model

(b) Grid degradation function

Figure 4.5: Approximate model for degradation of keyword spatial importance

$$score(k, C) = \frac{\alpha * \sum_{p \in C} inw(k, p) + \beta * \sum_{C' \in NC} \sum_{p \in C'} inw(k, p)}{3^d} \tag{4.10}$$

,where NC are the neighboring cells of C. Such a definition will assign higher value to $score(k, C)$ if k frequently appear in C as well as its neighboring cells. Finally, a normalization process similar to *idf* is proposed to reduce the effect of the general tag that occurs all over the grid:

$$igf(k) = ln \frac{|G|}{dg(k) + 1} \tag{4.11}$$

,where $|G|$ is the total number of d-dimensional grid cells and $dg(k)$ is the number of grids containing the keyword. Given all these formulae, the ranking score of k with respect to its location can be approximately defined as :

$$score(k, R) \approx \frac{\sum_{C \in R} score(k, C) * igf(k)}{|C|} \tag{4.12}$$

,where R is approximated by a set of cells C. Such an approximation takes the nearby documents into account. In our experiments, we will provide further analysis of the ranking mechanism.

4.4 Experiment Study

This section provides an extensive performance study on both synthetic and real data sets in order to evaluate the scalability of our query processing strategy as well as its practical utility.

We incrementally generate large synthetic data sets to simulate those from Web 2.0 applications. Meanwhile, real data sets extracted from online photo sharing applications are used to testify the practical utility of mCK query in local services and resource locating. All of our experiments are conducted on a server with Quad-Core AMD Opteron (tm) Processor 8356, 128GB memory, running RHEL 4.7AS.

4.4.1 Experiments on Synthetic Data Sets

Synthetic data sets are generated to simulate real-life applications in two aspects. First, in successful commercial applications, there are millions of users who create large amounts of resources. Hence, the data size we generate must be large scale. Second, the database expands continuously as new resources are added in by different users. Thus, scalability in terms of the number of locations and their associated tags becomes an essential issue. Our experiments are performed on synthetic data sets with millions of locations and thousands of tags. All the spatial data points are generated in a d-dimensional space $[0, 1]^d$ in a random manner. For most mapping applications, d is usually set as two. Each data point is randomly assigned with a fixed number of tags.

We compare our new virtual bR*-tree against the one proposed in Chapter 3 and MWSJ [89] in answering mCK query. These two algorithms retain the same settings as before. The average response time (ART) is used as our performance metric. In the following experiments, we compare the scalability in terms of m, the

number of locations and the associated tags.

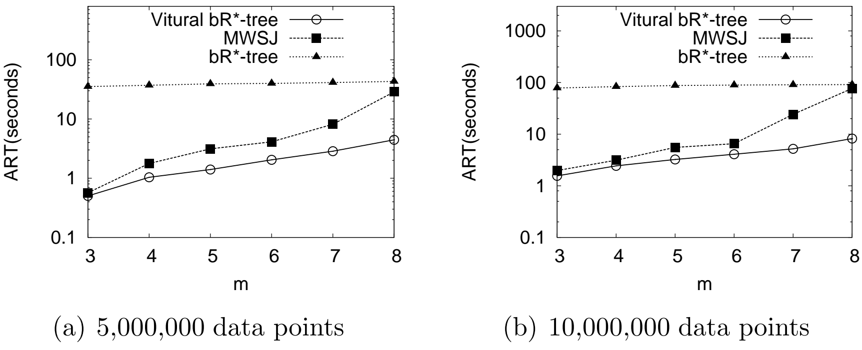
Scalability in terms of m

In this experiment, we randomly generate two data sets with 5,000,000 and 10,000,000 location points respectively. There are in total $5,000$ tags in the database and each point is associated with 5 random tags. The number of query tags submitted is varied from 3 to 8.

Figure 4.6 illustrates the average running time of the three algorithms. The bR*-tree proposed in Chapter 3 shows reasonable scalability as m increases. However, since the tree maintains auxiliary information in each node, this leads to extremely high I/O cost. The index occupies more than 4GB disk storage while our inverted index only takes up 527MB for the data set with $5,000,000$ points. The performance of MWSJ shows the same pattern as in Chapter 3. When m is small, the query can be answered efficiently. When m is increased to larger values, the performance starts to degrade due to the exponential expansion of the search space. Our virtual bR*-tree demonstrates advantage over the other two algorithms. This is due to the improved index structure as well as the efficient pruning strategy. In the following two experiments, we only compare the performance with MWSJ as the original bR*-tree is not suitable for handling data set with a massive number of tags.

Scalability in terms of the number of locations

In order to simulate the real applications in which new resources are continuously marked in the map, we generate the synthetic data sets with the size increasing steadily from $5,000,000$ to $10,000,000$ data points. The number of tags associated with each point is fixed as 5. Figure 4.7 shows the performance trend as the data



(a) 5,000,000 data points (b) 10,000,000 data points

Figure 4.6: Scalability in terms of m

size increases.

When m is small, both algorithms demonstrate similar growth rate in running time. The spatial index does take effect to suppress the expansion of search space. However, as m increases, the performance of MWSJ becomes sensitive to the growth of data size. The reason is that the search space grows substantially with the data size. A small amount of increase in data size can lead to remarkable expansion of the search space. In contrast, our virtual bR*-tree is able to scale in a stable manner.

Scalability in terms of the number of tags

In real applications, new tags can be added to describe a particular resource. In this experiment, synthetic data sets are generated to simulate the growth in the number of tags. Here, we increase the number of tags associated with each location from 1 to 9. The spatial database contains $5,000,000$ data points and $5,000$ different tags in total.

The two algorithms in Figure 4.8 present similar growth rate to that in Figure 4.7 in terms of response time. Their performance is good when m is small. However, MWSJ suffers from serious degradation in handling large numbers of query tags because it does not inherently support effective summarization of tag locations.

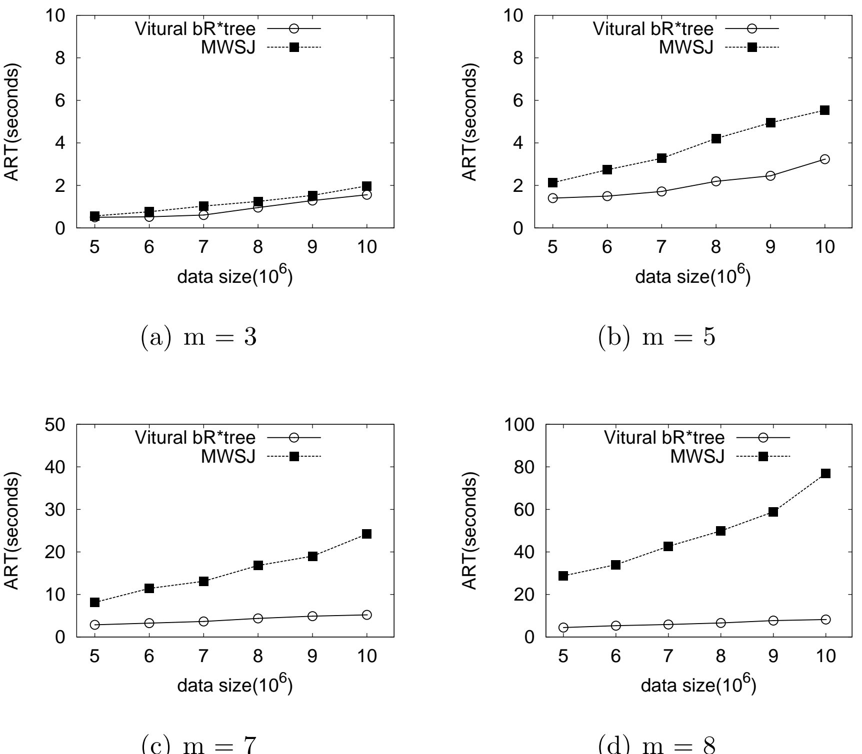


Figure 4.7: Scalability in terms of the number of locations

Our virtual bR*-tree, on the other hand, demonstrates good scalability.

Note that the virtual bR*-tree performs slightly better with respect to the growth in the number of tags compared to the growth in number of locations. The reason is that the virtual node in the bR*-tree maintains a bitmap indicating the query tags within. The insertion of a tag into an existing location will only trigger the setting of the bit in the bitmap. However, the insertion of new locations leads to a larger labelled R*-tree. Therefore, more virtual nodes will need to be created during the search process.

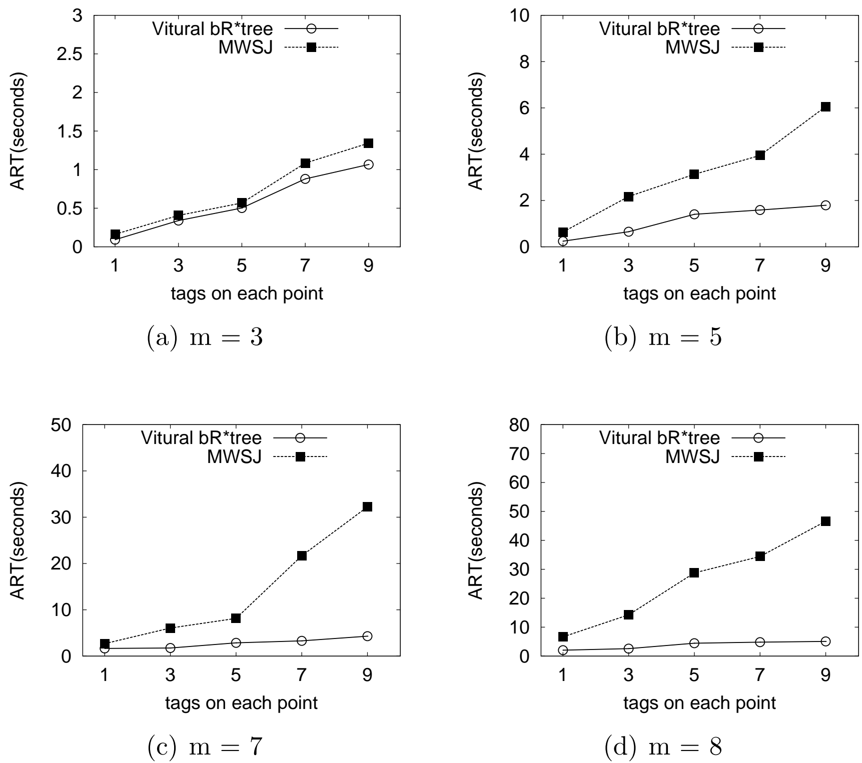


Figure 4.8: Scalability in terms of the number of tags

4.4.2 Experiments On Real Data Sets

Our real data set is generated via the Flickr service API[1] and Picasa Web Albums Data API[2]. We extract all photos in New York that are tagged and geo-marked. Resources located at the same coordinates are merged into the same tag list. After removing the infrequent tags, the database consistes of 74,774 location points from Flickr and 6,729 points from Picasa Web. There are 12,636 tags in total.

Based on our observation, the geo-tag data set is of acceptable quality. Most of the photos are assigned with relevant tags and are correctly marked in the map. The geo-tags are usually distributed in the form of spatial clusters. These clusters can be utilized to identify the locations of popular resources and events because related tags will emerge around that area. For instance, as shown in Figure 4.9,

[1]http://www.flickr.com/services/api/
[2]http://code.google.com/apis/picasaweb/overview.html

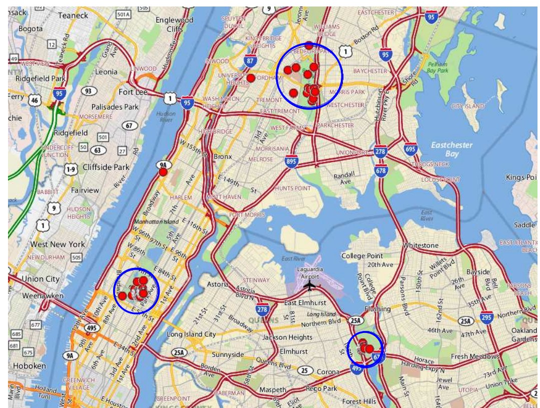


Figure 4.9: Distribution of tag “zoo”

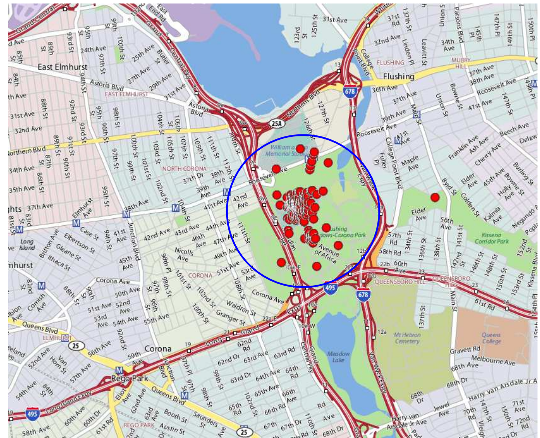


Figure 4.10: Distribution of tag “USOPEN”

the tag “zoo” is mainly distributed in three spatial clusters corresponding to Bronx Zoo, Central Park Zoo and Queens Zoo Wildlife Center respectively. Similarly, in Figure 4.10 there are a large number of “USOPEN” tags gathering around the Arthur Ashe Stadium where the tennis match is held. This phenomenon provides us with new opportunities to locate resources in a more precise geographical scale.

In the following experiments with real data set, we design a set of queries using tags or photos as the input. These queries are mainly for locating local services.

Tag Query

In this experiment, users propose query tags from the perspective of finding local service, including restaurant, museum, shopping, recreation center, viewing site, local news and so on. As shown in Figure 4.11, fifteen example queries are listed and the results are compared against Google Maps. These local service queries can be divided into three categories:

Type	**Tag Query**	**Google**	***m*CK**
I	*liberty statue*	√	√
	marc jacobs store	√	×
II	*restaurant seafood sashimi*	√	×
	bar cocktail jazz player	×	√
	museum dinosaur fossil	√	√
	tennis court Williams champion	×	√
	weapon factory	×	×
	river airplane crush	√	√
	campus Barack Obama	×	√
	park river fishing	×	√
	recreation bowling billiard	√	×
III	*square fountain roller skating*	×	√
	applestore subway	√	√
	supermarket gas station	√	×
	hotel church catholic historic	×	√

Figure 4.11: Example tag queries

- The first type of queries is landmark query, such as “Statue of Liberty” and “Marc Jacobs Store” in our examples. Google Maps can answer this kind of queries effectively as an enormous number of landmarks have been correctly maintained in the database. Each time a query is submitted, it will first look for gazetteer terms so as to reduce the search space. Our mCK query strategy can correctly give the result for “Statue of Liberty” because this is a famous viewing site in New York City and many photos have been tagged and marked around the statue. However, no result is returned for “Marc Jacobs

Store" as no geo-tagged photos about "Marc Jacobs" exist in our data sets.

- The second type of query is seeking for a subject associated with constraint features, persons or events that users are interested in. If the association is common and straightforward, such as a restaurant with seafood and sashimi, a museum with dinosaur fossil and recreation center with bowling and billiard, Google Maps is a good choice. However, this search engine is not suitable for locating subjects with complicated features like "tennis court where Williams won the champion" and "bar with cocktail and jazz player". In contrast, our mCK query can answer most of the queries as long as the target location is well tagged. For the infrequent tags in our data set, such as "sashimi", "recreation", "billiard", it is difficult for them to occur simultaneously with other query tags in the same location. Thus, no related results are returned. The other problem with mCK query is that it can not guarantee that the results returned are related to the same subject. In the query "weapon factory", Knitting Factory, a music club and concert house, is returned. The reason is that there are no available weapon factories in the database and it happens that there is a poster about weapon on the door of the music club so that the two tags "weapon" and "factory" become connected.

- The third type of query differs in that spatial constraint is embedded. For example, the query "applestore subway" aims to find the apple retail stores near the subway. Similarly, the query "supermarket gas station" intends to find a supermarket and gas station close to each other. Google Maps is able to answer these two queries because the result returned happens to contain all the query tags. It can not capture the spatial constraint so as to answer queries like "find a historic catholic church with hotels nearby". Our mCK in essence is proposed to answer this type of query. The quality of the search

result relies on the quantity of tags contributed by users.

Figure 4.12 illustrates some results of example queries returned by mCK and Google Maps. We can observe that Google Maps pays more attention to tags with geographical context, such as college and church. The other keywords used as features or spatial constraints are ignored. Thus, its results can not capture the correct query subject or a user's intention. Our mCK query takes all the tags into account and find a location matching all of them. Such a query mechanism is able to capture the complete meaning and return satisfactory results.

Query Tags	**mCK**	**Google Maps**
BarackObama campus	Columbia University	**Yeshiva University Main Campus** **Campus** Description: 12 acre Yeshiva **College** **campus** in uptown Manhattan Stern **College** **campus** in midtown Manhattan Sy Syms School of Business at both campuses... a2zcolleges.com
hotel church catholic historic	St Patrick's Cathedral	**Old St Patricks Cathedral School** Latin **Catholic** - 2 NYC Cathedrals The people usually known as "Roman Catholics" - the **historic** **Church** of the West under the immediate supervision of the Pope ...fordham.edu

Figure 4.12: Example results returned by mCK and Google Maps

Ranking Mechanism

In this part, we provide more in-depth analysis of our ranking strategy. As mentioned, we assign higher value to $score(k, R)$ if keyword k appears frequently around R and infrequently in other locations. Such keywords usually refer to the distinguishing and prominent entities. To achieve this goal, the primary issue is to determine the size of the grid cell. The setting of this parameter is closely related to the specific services being provided. Precise location at the level of a shop or

sculpture desires a small cell size. Otherwise, we can allow larger cells to save maintenance cost. Note that hierarchy grid structure can be designed to support locating services at different geographical scales. In our experiment on the New York data set, the grid is split into 500 × 500 cells.

Fig 4.13 shows the ranking scores of query tags with respect to the detected location. The bold tags "crash", "catholic" are important and prominent subjects or features in that location. Although tags like "river" and "hotel" also appear frequently, they are not distinguishing enough. These tags spread round the city and the *igf* (inverse grid frequency) takes effect to assign lower scores to them. In addition, we can tell from the figure that most feature tags are assigned with moderate scores.

t_8	airplane 141.221	**crash** 217.448	river 118.497	
t_{15}	hotel 114.849	church 214.979	**catholic** 685.173	historic 45.2946

Figure 4.13: Ranking score for tag query

The last essential issue about ranking is the weights of the query terms. In default, the query tags are assigned with equal weights. When the results returned are not satisfactory, users are allowed to adjust the weights to highlight the important terms to better identify their intention. For example, given query tags "fountain, square", a famous square with a fountain may be returned as the square may be frequently tagged, leading to a dominating score. If the user is actually searching an ornate fountain in a square, he can increase the weight of "fountain". If the fountain is the prominent scene, it is likely to be more frequently tagged than the square where it is located and the famous fountains will be returned.

Accuracy

To further test the accuracy of resource locating, we invite a group of volunteers to generate local service queries for us and verify whether the returned results are satisfactory or not. Since our data set is still too small to meet with daily needs, we extract frequent tags and ask the volunteers to create the query from the combination of these tags. All the 50 queries are shown in Fig 4.14.

sunrise hotel	hospital plaza	dinner movie	island waterfall	weapons factory
wedding church	gold sunset beach	rockband beer	waterfall ferry	weapon museum
orange lamp library	historic architechture	historic museum	pizza plaza	sunset lake
waterfalls hotel	fireworks square	bridge train	river fishing	dogs pet shop
garden cafe	monument square	movie theatre	kids playground	dinner plaza chicken
baseball stadium	sunset skyline	sunset boat sea	tennis usopen	iphone gallery
beach guitar moon	dance rock band	fish market	fashion jewelry shop	architecture museum
sexy girls	flower exhibition	bronze statue	towers lamp	ice cafe pizza
island moon	bowling beer dinner	girls gold shoes shop	911 monument	japanese restaurants
vocation island	ice stadium	park band bass	historic movie theater	museum pizza

Figure 4.14: Local service queries

In this experiment, we first use the Flickr data set and later add in the Picasa Web data set to examine whether the idea of mashup works. The accuracy results are shown in Fig 4.15. When there is only one data set, mCK query only performs slightly better than Google Maps. The reason is that mCK seeks for a location matching the query tags but these tags are possible to be associated with different unrelated subjects. If multiple data sources from other applications are combined, it is more likely for the related query tags to appear in the same location and their distance becomes 0. As such, we obtain an improvement in accuracy when the Picasa Web data set is incorporated.

Photo Query

In this experiment, we discuss how to locate a photo using mCK query. Given a photo, the semantic objects as well as their features can be extracted via human intelligence and represented as query tags. These tags are close to each other in

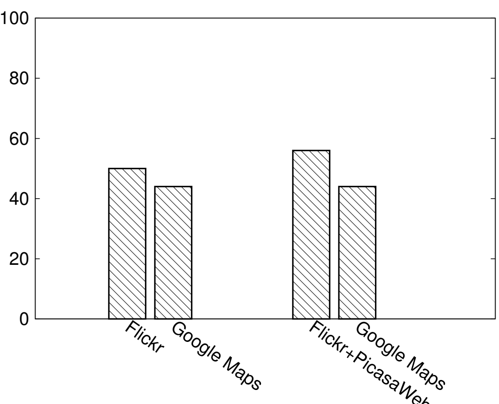


Figure 4.15: Accuracy result

the physical space. Google Maps is not suitable to handle such queries because it is unable to capture the spatial constraint. Thus, we try to solve the problem by submitting an mCK query using the extracted tags.

As different query tags extracted from the same photo may result in different matching locations, the selection of extracted tags becomes an important issue. Based on our experiments of the query photos in Fig 4.16, we observe that:

- Distinguishing tags are preferred as they can help to reduce the search space. For example, in the second query, “skyscraper” is a frequent tag that appears around the downtown of New York City and leads to many candidates. However, “mast” is a distinguishing keyword as it is found in limited locations. The search space can be further reduced through the spatial constraint that the skyscraper is near the mast.

- The number of candidate locations can be reduced by adding new query tags. When there are no distinguishing features embedded in the photo, users can provide more tags from the image to eliminate false positive. For instance, in the first photo query, all the four query tags are widely distributed in New York City. Their spatial constraint assists us in detecting the correct location. Missing any of the query tags could lead to a false result.

Other types of resources, such as blogs, news and videos, can also be located in a similar manner. As long as the resource is concerned with a local area, their associated tags are likely to spread around that area. Therefore, mCK is a useful query in detecting geographical context of mapped resources.

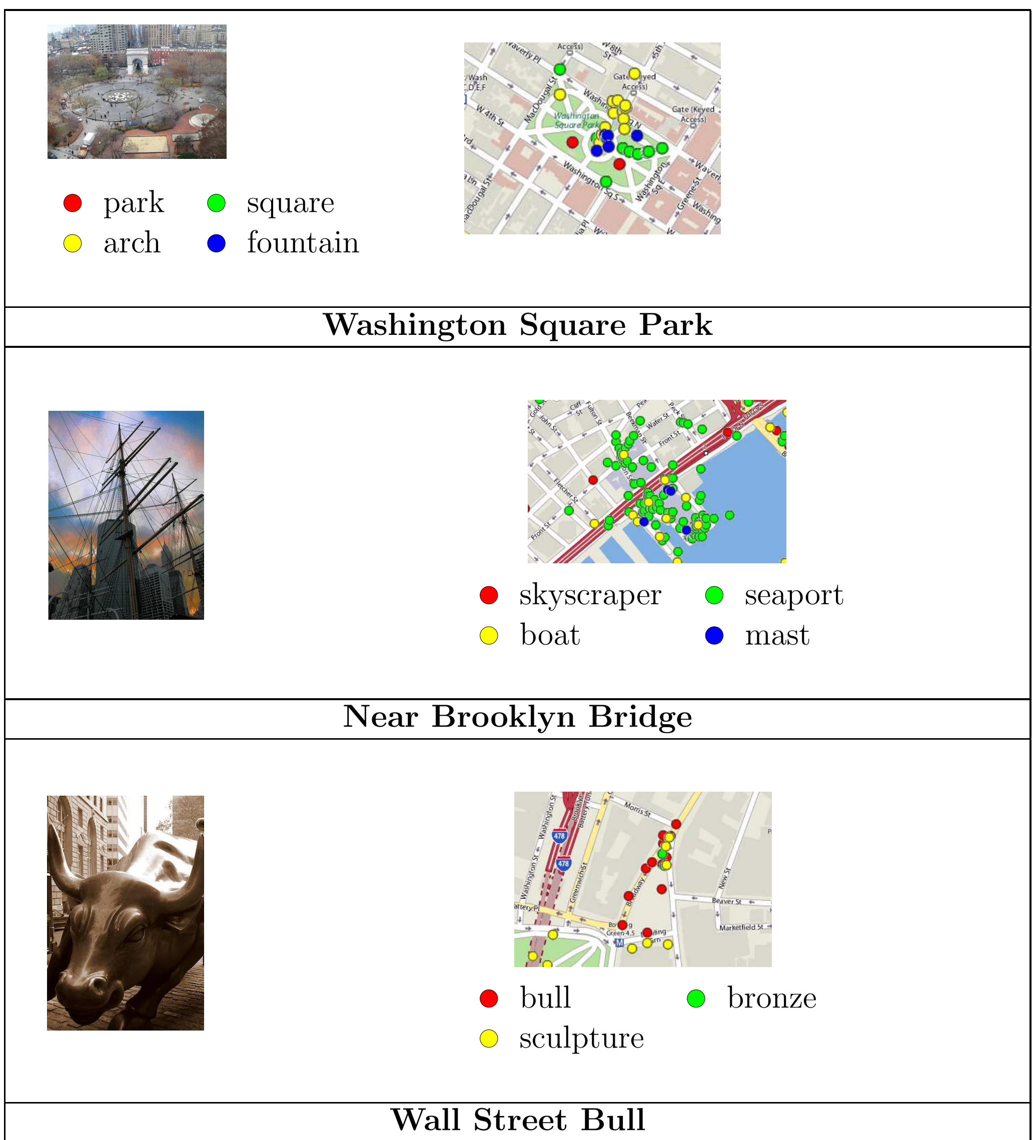


Figure 4.16: Example photo queries

4.5 Summary

In this chapter, we addressed the new emerging problem of locating mapped resources in Web 2.0. We proposed that we can use tags to build a general data and query model to support co-location searches by tag matching. The data resources from different applications can be combined and integrated into the labelled R*-tree and inverted index. Efficient search strategies are developed to effectively answer the tag matching query. We have also proposed a new *geo-tf-idf* ranking mechanism to measure the geographical relevance. Extensive experiments using both synthetic and real life data sets confirm the feasibility and efficiency of our proposed design in the Web 2.0 environment.

CHAPTER 5

LANDMARK RECOGNITION USING HASHFILE

Content based matching method is normally used to detect the geographical context of web resources when tag is not available. In this chapter, we focus our problem on detecting geographical context of images or in other words, landmark recognition. With the increasing popularity of personal digital photography and online photo sharing, landmark recognition becomes an important interface to the vast collection of landmark photos. Typically, the problem is solved by comparing the input photo with all the photos in the database, which is essentially a nearest neighbor query.

In this chapter, we propose a novel index structure, named **HashFile**, for efficient retrieval of multimedia objects. It combines the advantages of random projection and linear scan. Unlike the LSH family in which each bucket is associated with a concatenation of m hash values, we only recursively partition the dense buckets and organize them as a tree structure. Given a query point q, the search algorithm

explores the buckets near the query object in a top-down manner. The candidate buckets in each node are stored sequentially in increasing order of the hash value and can be efficiently loaded into memory for linear scan. HashFile can support both exact and approximate NN queries. Experimental results show that HashFile performs better than existing indexes in answering both types of NN queries.

5.1 Introduction

Due to the proliferation of Web 2.0 social and community systems, a large number of multimedia objects are publicly available. Efficient access to such multimedia objects needs to be supported in order for the web users to benefit from such data. In this chapter, we study the problem of nearest neighbor (NN) search, which is an essential query in many multimedia retrieval applications. We investigate processing strategies for both exact and approximate NN queries. The former has wide applications in similarity search, pattern recognition, clustering and classification. The latter is particularly suitable for efficient retrieval in a large scale database at the risk of certain loss in quality.

The most common and straightforward method for solving exact NN problem is based on hierarchical space partitioning, resulting in various kinds of tree structure indexes [94, 64, 29, 77]. The multi-dimensional feature space is split into smaller partitions and organized as a tree structure. Data close to each other are grouped in the same node so that they can be pruned together without accessing each individual point inside. However, the pruning power of these indexes decreases as dimensionality grows and most of the tree nodes will be accessed, taking considerable CPU and I/O cost. In this case, the performance of existing index structures degrades rapidly and even becomes worse than a simple sequential scan of the

data [110, 30]. Due to this curse of dimensionality, it is difficult to build indexes to efficiently answer exact NN queries.

To provide efficient similarity search, the research community has focused on approximate NN search in recent years. Among various efforts, locality sensitive hashing (LSH) [60, 53] and its variants have received considerable attention. The LSH family adopts hash functions that preserve the distance in the Euclidean space so that similar objects have a high probability of colliding in the same bucket. If there are l hash tables and each table H_i is associated with m hash functions G_{ij}, an object o will be hashed to $H(o) = [h_1, h_2, ..., h_l]$, where $h_i = G_{i_1}(o)G_{i_2}(o)...G_{i_m}(o)$. Given a query object q, the search space includes the buckets in the l hash tables where q is located. All the objects in these buckets are scanned to return the approximate NN result. As m increases, the bucket size becomes smaller and more false positives are removed. Precision increases but recall degrades. Similarly, as l increases, more buckets are examined. Recall is improved but precision may become worse. Thus, the main challenge of LSH is to tune a good tradeoff between precision and recall. To achieve a high search accuracy, hundreds of hash tables are normally used [53] and require a large amount of memory space. In [83], multi-probe LSH was proposed to reduce the number of hash tables and obtain the same search quality. Since multi-probe LSH is adhoc and without theoretical guarantee, Tao et al. have recently proposed the locality sensitive B-tree (LSB-tree) [105] to ensure both quality and efficiency. The drawback of LSB tree is that it uses random I/O access, which requires a considerable number of disk accesses when the database is large.

To support efficient NN query processing, we propose a novel index structure, named *HashFile*, based on the following three observations:

1. In LSH, the hash function is likely to place most of data objects into the

buckets near the mean hash value, resulting in a skewed distribution of bucket size. When m increases, the dense buckets can be recursively partitioned to reduce the number of false positives. However, other buckets containing fewer points will be partitioned as well. Since the data points inside these buckets are well separated from others, further partitioning will generate a large number of very small buckets and the quality of the approximate results may degrade.

2. Expanding the search space by inspecting neighboring buckets can significantly improve the result quality [83, 71] because some missing nearest neighbors can be retrieved in this way without building a new hash table. Although such a method is adhoc and without theoretical guarantee, we argue that the theoretical bound obtained by previous works [60, 105] is too loose to be applied in practice. For example, LSB-tree only guarantees 4-approximate NN with at least constant probability. In other words, the distance of NN result returned by LSB-tree can be guaranteed to be within 4 times of the real best distance.

3. Disk page access method plays an important role in the cost of the index lookup [27], especially for high dimensional data. The time of random access is higher than that of sequential access by many times.

In our implementation, the entire data set is first hashed into a set of buckets like LSH. The dense buckets are fetched and re-hashed so as to further separate the objects inside to remove the false positives. The remaining buckets only contain a small number of points and will not be further partitioned. The process will be repeated until there is no dense bucket. In this manner, the distribution of the bucket size is much more balanced than that generated by LSH. We organize

HashFile as a tree structure and each node is associated with a unique hash function as well as a data file to store the buckets that can not be further partitioned. These buckets are stored in increasing order of the hash value for linear scanning. We propose a dynamic bucket allocation strategy to guarantee a minimum storage utilization of 50%. Given a query point q, the search algorithm explores the tree in a top-down manner and only retrieves those pages around the query object. The candidate pages in each file are retrieved using sequential scan, with a single disk seek operation followed by the data transfer.

We choose random projection as our hash function. This is because it is simple, database-friendly [15] and yields comparable results to conventional dimensionality reduction, such as PCA, for both image and text data [32]. Furthermore, using random projection, we can extend the search algorithm to support exact NN search in L_1 norm, which is found to be effective for multimedia data [16]. Experiment results on image data sets show that performance is improved as random projection is useful to filter away the data points that are far away and the remaining candidates are processed efficiently using a linear scan. In summary, we propose HashFile, a novel index structure for processing nearest neighbor queries efficiently over multimedia databases. HashFile has the following desirable features:

1. It supports approximate NN search in the Euclidean space as well as exact NN search in L_1 norm.

2. It has a linear space complexity of $O(2N + N/B)$ where N is the data size and B is the page size.

3. Experiment results show that HashFile outperforms state-of-the-art techniques for processing both types of NN queries.

The rest of the chapter is organized as follows. First, in Section 5.2, we review

the necessary background to provide the underlying ideas and the intuition behind HashFile for a better understanding. The detailed algorithm for inserting, updating and deleting data objects is presented in Section 5.3. The query processing strategies for exact NN and approximate NN are proposed in Sections 5.4 and 5.5, respectively. In Section 5.6, we analyze the complexity of data operations and query processing. Extensive experiments on the real image data sets are conducted in Section 5.7 to establish the superiority of the proposed methods over current state-of-the-art NN processing techniques. Section 5.8 concludes the chapter.

5.2 The Preliminaries

In this section, we briefly review the theoretical background on random projection and motivate the intuition for the notion of HashFile.

5.2.1 Random Projection

Random projection is a powerful method for dimensionality reduction. It is computationally efficient and sufficiently accurate. Given a d dimensional data set P with n points and a random matrix $R_{d\times k}$, the projection is computed as follows:

$$P'_{n\times k} = P_{n\times d} \times R_{d\times k},$$

which results in a k dimensional data set P' with n points. Random projection can preserve the Euclidean distance in the lower dimensional space provided the constraints specified by the Johnson-Lindenstrauss Lemma [62] hold.

Lemma 5.1 (Johnson-Lindenstrauss Lemma). *Given $\epsilon > 0$ and an integer n, let k be a positive integer such that $k \geq k_0 = O(\epsilon^{-2} logn)$. For every set P of n points*

in $\mathbb{R}^d$*, there exists* $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ *such that for all* $u, v \in P$*, we have*

$$(1-\epsilon) \parallel u - v \parallel^2 \leq \parallel f(u) - f(v) \parallel^2 \leq (1+\epsilon) \parallel u - v \parallel^2$$

The choice of random matrix is a key step and it is often Gaussian distributed with mean 0 and variance 1 to solve the ϵ-approximate nearest neighbor problem [69, 72, 60, 83, 105]. In [15], Achlioptas showed that the computational efficiency can be further improved by replacing Gaussian distribution with a much simpler random distribution:

Lemma 5.2. *Given* $\epsilon, \beta > 0$*, let* $k_0 = \frac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} \log n$*. For integer* $k \geq k_0$*, let* R *be a* $d \times k$ *random matrix where the elements are independent random variables from either one of the following probability distributions:*

$$r_{ij} = \begin{cases} +1 & \textit{with probability } 1/2 \\ -1 & \textit{with probability } 1/2 \end{cases}$$

or

$$r_{ij} = \sqrt{3} \begin{cases} +1 & \textit{with probability } 1/6 \\ 0 & \textit{with probability } 2/3 \\ -1 & \textit{with probability } 1/6 \end{cases}$$

With probability at least $1 - n^{-\beta}$*, for all* $u, v \in P$*,*

$$(1-\epsilon) \parallel u - v \parallel^2 \leq \parallel f(u) - f(v) \parallel^2 \leq (1+\epsilon) \parallel u - v \parallel^2$$

In practice, we can select either of the two projection functions for dimension-

ality reduction. In our implementation, we pick r_{ij} randomly from $\{-1, 1\}$.

5.2.2 Distance Constraint for Exact NN Query Using L_1

Suppose $\mathcal{H}$ is a hash function derived from any row of $R_{d\times k}$ such that it hashes an object o with d dimensions into one dimensional value:

$$\mathcal{H}(o) = \lfloor \sum_{i=1}^{d} h_i \cdot o_i \rfloor$$

Since each element h_i in $\mathcal{H}$ is from $\{-1, 1\}$, we can easily achieve a lower bound for the exact L_1 distance.

Lemma 5.3. *Given two d dimensional data points x and y and a hash function $\mathcal{H} : \mathbb{R}^d \rightarrow \mathbb{R}^1$ with $h_i \in \{-1, 1\}$, we have*

$$\| x - y \|_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)| - 1$$

Proof.

$$\begin{aligned} \| x - y \|_{L_1} &= \sum_{i=1}^{d} |x_i - y_i| \\ &\geq \sum_{i=1}^{d} |h_i(x_i - y_i)| \\ &\geq |\sum_{i=1}^{d} h_i(x_i - y_i)| \end{aligned}$$

Since for two real values a and b, $|a - b| \geq |\lfloor a \rfloor - \lfloor b \rfloor| - 1$, we have

$$\| x - y \|_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)| - 1$$

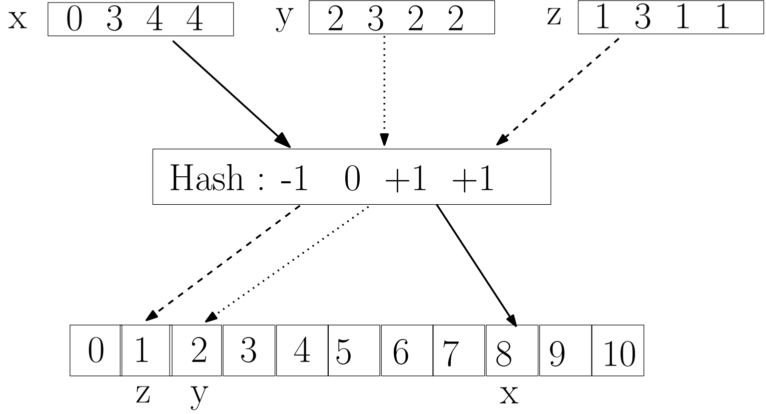


Figure 5.1: A random projection example

□

Example. Figure 5.1 shows an illustrative example. Given three 4-dimensional data points x, y and z, a random hash function is generated to hash each point into a 1-dimensional value. It is obvious that the distance of the hashed value is the lower bound of their real distance. Since the elements are integers, we have $|a-b| = ||a| - \lfloor b \rfloor|$ for two integers a and b. The factor "-1" can be dropped from the RHS and therefore we get: $\| x - y \|_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)|$.

$$\| x - y \|_{L_1} = 6 \geq 6 = |\mathcal{H}(x) - \mathcal{H}(y)|$$
$$\| x - z \|_{L_1} = 7 \geq 7 = |\mathcal{H}(x) - \mathcal{H}(z)|$$
$$\| y - z \|_{L_1} = 3 \geq 1 = |\mathcal{H}(y) - \mathcal{H}(z)|$$

Inspired by this, we adopt random projection to partition the large volume of high dimensional data points into buckets in 1-dimensional space. Each bucket is associated with a hash value and occupies one disk page. The pages are stored

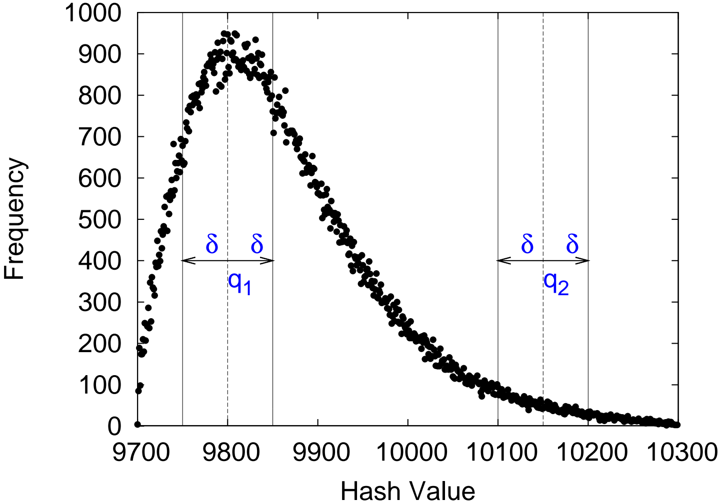


Figure 5.2: Hash value frequency of a color histogram dataset

sequentially in increasing order of hash value. Given a query point q, suppose it is hashed into bucket h_q. We denote δ as the distance to the nearest neighbor ever found. According to Lemma 5.3, only the pages with hash value in $[h_q - \delta, h_q + \delta]$ need to be accessed. The other data points outside the range can be safely pruned. Figure 5.2 illustrates an example of frequency distribution of hash value derived from a color histogram dataset with $200,000$ points corresponding to 200K images, which looks similar to a normal distribution. Most of the data points are hashed into the buckets around the mean hash value μ. For buckets far away from μ, the number of points inside the bucket drops dramatically. Therefore, if h_q is lucky to be far from μ, e.g., q_2 in the figure, the search space only includes a set of buckets with relatively few points inside the buckets. Since the pages with hash value in $[h_q - \delta, h_q + \delta]$ are stored sequentially, we can use linear scan to process the data without resorting to random I/O accesses. However, if $h_q = q_1$ as shown in the figure, we need to access the majority of the data points to answer exact NN query.

To solve this problem, we need to further partition the dense buckets. A new

hash function is created for each dense bucket to further partition the data points inside. We expect that some amount of data far away from q can still be pruned away via the re-hashing. However, as each bucket is associated with only a value of $\mathcal{H}(\cdot)$, the bucket size is relatively small. As shown in Figure 5.2, even the densest bucket contains less than $1,000$ data points. The pruning power of re-hashing in such a small partition is very limited. Each time we access the new partition, only very few points can be pruned. This results in high I/O cost to perform sequential scan on these small files. Therefore, we adopt the popular hash function in LSH to increase the number of points hashed to each bucket:

$$\mathcal{H}(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + b}{W} \rfloor$$

In our case, a_i is selected the same with r_{ij}. b_i is not required and simply set to 0. W is the hash window size. The original hash space is split into intervals with length W and the data points hashed to the same interval will be stored in the same bucket. Obviously, the larger W is, the more data points will be hashed to the same bucket. Figure 5.3 shows the new frequency distribution in the larger buckets. W is set to 450 and there are now only 15 partitions in the hash space. The densest bucket contains around $35,000$ data points. Given a fixed page size, all the buckets whose size exceeds the page size will be re-partitioned to gain additional pruning power. When q is located in the dense area, we do not need to access all the data points in the nearby buckets as in Figure 5.2. Instead, points far away from q can still be pruned to save the CPU cost. Finally, we prove that we can still achieve a lower bound of distance constraint using the new hash function.

Lemma 5.4. *Given two d dimensional data points x and y and a hash function*

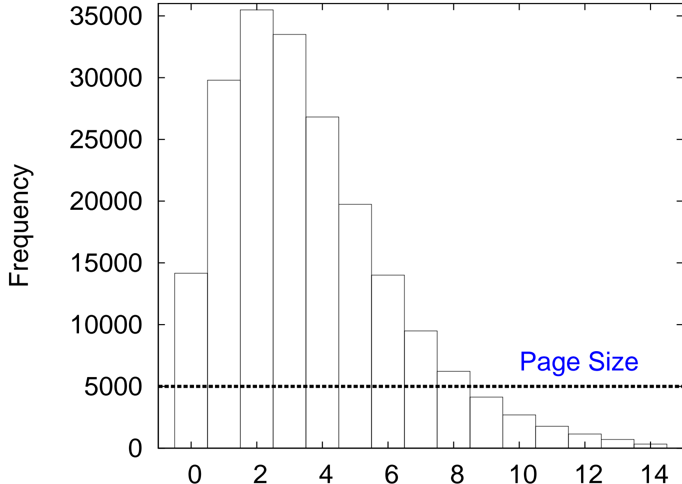


Figure 5.3: New frequency distribution of window based hashing

$\mathcal{H}_w(o) = \lfloor \frac{h_i \cdot o_i}{W} \rfloor$, *we have*

$$\| x - y \|_{L_1} \geq W(|\mathcal{H}_w(x) - \mathcal{H}_w(y)| - 1)$$

Proof.

$$\begin{aligned} \| x - y \|_{L_1} &\geq |\sum_{i=1}^{d} h_i(x_i - y_i)| \\ &= W|\sum_{i=1}^{d} \frac{h_i x_i}{W} - \sum_{i=1}^{d} \frac{h_i y_i}{W}| \\ &\geq W(|\mathcal{H}_w(x) - \mathcal{H}_w(y)| - 1) \end{aligned}$$

□

In the extreme case, when W is set to $+\infty$, HashFile degrades to linear scan without any pruning power. In our experiment setup, we will study how the per-

formance varies with this parameter.

5.3 HashFile Index Structure

In the section, we first present the overall design of the HashFile index structure then describe the algorithms for data insertion, update and deletion in HashFile. The notations used in the presentation are summarized in Table 5.1.

Table 5.1: Notation table

d	the data dimension
o	an object with d dimensions
W	the hash window size
$\mathcal{H}_w$	$\mathcal{H}_w(o) = \lfloor \frac{\sum_{i=1}^{d} h_i \cdot o_i}{W} \rfloor$
$[l_i, h_i]$	the hash interval of a page
B	the page size to host B objects
δ	the best NN distance ever found
μ	the storage utilization rate
κ	the tree height of HashFile

5.3.1 HashFile Overview

We build HashFile in a top-down manner based on the distribution of the data set. In the beginning, we randomly generate a hash function in the root node and create a disk file to store the data. Since we are unable to predict the hash range, we cannot allocate a collection of fixed-size pages in advance. Moreover, such a static allocation strategy wastes a lot of storage resource because the buckets far from the mean hash value usually contain a small number of data points, resulting in a low storage utilization. Hence, we propose a dynamic allocation mechanism which guarantees that the page utilization is at least 50%.

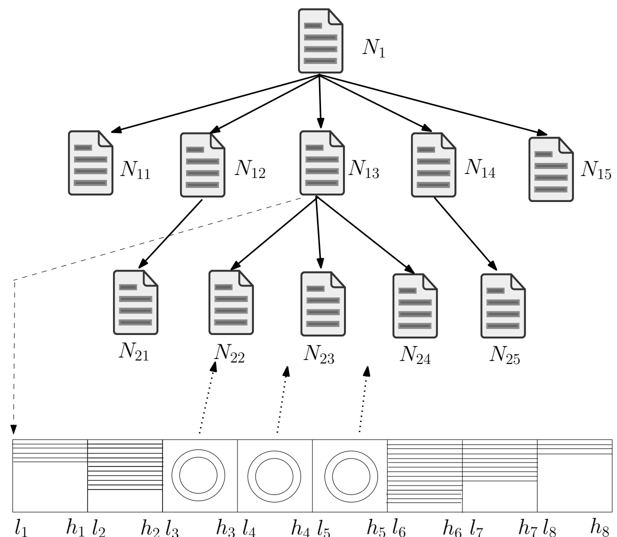


Figure 5.4: The structure of HashFile and HashNode

Initially, there is only one page allocated in the disk file with a hash interval $(-\infty, \infty)$. The size of the page is fixed as B, indicating that it can accommodate B data points. The first B objects can be successfully inserted into the page. When the $(B+1)$-th object arrives, the page is full and we select a hash value h_s and split the page into two new pages. Now, the object can be inserted into the new page based on its hash value. As new objects are inserted continuously, more and more split operations occur and the hash intervals of the page become smaller and smaller. Finally, an object may be hashed to a full bucket in which all the data points inside this page are associated with the same hash value and can not be split any more using the node's hash function. To insert this new object, we need to create a child node with a new random projection function. The data points in that page are extracted and re-hashed into the child node's data file. At this time, we have vacant space in the child node to accommodate the new object.

Figure 5.4 illustrates an example of HashFile with the tree structure and the logical view of one internal tree node. The tree is built by continuous insertion of

data points. When a bucket in the parent node can not be split any more, child nodes are created to further partition the dense bucket. Therefore, each internal node contains a list of child nodes and pages as shown in the figure. Each cell represents one page with a hash interval $[l_i, h_i]$. The pages are stored in increasing order of the hash value, i.e., $l_i \leq h_i \leq l_{i+1}$ and $\bigcup [l_i, h_i] = (-\infty, \infty)$. The pages with circles inside are buckets with only one hash value and the data inside have been extracted and re-hashed into the child node. In this example, we have $l_3 = h_3$, $l_4 = h_4$ and $l_5 = h_5$. Therefore, node N_{13} has three child nodes N_{22}, N_{23} and N_{24}. Note that in the physical file, these three buckets do not occupy any page as their content has been extracted and stored in the child nodes' disk files. In this way, we ensure that the remaining disk pages are still sequentially stored.

5.3.2 Data Insertion

Algorithm 7 `InsertData`(o) **: Insert an object into HashFile**

Input: Data object o

1. $p_i =$ `FindPage`(o)
2. **if** p_i is not full **then**
3. append o in p_i
4. **else**
5. **if** $h_i > l_i$ **then**
6. select a hash value h_s
7. split the pages into two pages p_{i_1} and p_{i_2} with intervals $[l_i, h_s]$ and $[h'_s, h_i]$ respectively
8. leave data points of p_{i_1} in the page p_i
9. create a new page for p_{i_2} and insert p_{i_2} next to p_{i_1}
10. **if** $h_o \leq h_s$ **then**
11. append o in p_{i_1}
12. **else**
13. append o in p_{i_2}
14. **else**
15. create a new hash function and a new disk file as the child node
16. extract p_i and hash the points to the new file
17. move p_j to p_{j-1} for $j > i$ in the parent node
18. `InsertData`(o)

Algorithm 8 `FindPage(o)` **: Find the disk page o is hashed into**

Input: Data object o
Output: The disk page that o is hashed into

1. init *node* as the root node
2. **while** 1 **do**
3. $h_o = node.hash(o)$
4. **if** h_o is the hash value of a child node *child* **then**
5. $node = child$
6. **else**
7. find disk page p_i so that $h_o \in [l_i, h_i]$
8. return page p_i

Given a data object o to be inserted into HashFile, we need to find the disk page in the tree node where o is hashed into. Since the hash value range of each tree node is $(-\infty, +\infty)$, we can ensure that the hash value of o must be located either in a disk page or in one of the child nodes. Thus, we start from the root node and compare the hash value of the intervals of each page. If we find a page p_i with interval $[l_i, h_i]$ such that $h_o \in [l_i, h_i]$, o will be inserted into this page. Otherwise, o must be hashed into one of the child nodes. We compare the hash value with the list of child nodes until we find a match. The child node is visited in a similar way to the root node. Finally, we must be able to find a disk page p_i in the tree node to host the object o.

If the found page p_i has vacant space, we directly append o into the page. Otherwise, we need to check the hash range of p_i to determine whether we split the page or create a new child node. If objects inside p_i are associated with multiple hash values, we can find the median hash value h_s so that we can split p_i into two new pages p_{i_1} and p_{i_2}. h_s is selected to make sure the page is split into two even parts to guarantee the storage utilization is no less than 50%. The hash intervals of the p_{i_1} and p_{i_2} become $[l_i, h_s]$ and $[h'_s, h_i]$ respectively. Note that the objects hashed to h_s could appear in both new pages. In this case, $h_s = h'_s$. Otherwise, $h'_s = h_s + 1$.

We leave the objects hashed into $[l_i, h_s]$ in the original page p_i and create a new physical page for p_{i_2}. To ensure that the hash intervals are in increasing order for sequential scan, we insert the page p_{i_2} right next to p_i in the file.

If p_i is full and all the objects inside p_i are hashed to the same value, we can not split the page any more. Thus, we create a new hash function as the child node for further partition of p_i. All the contents in p_i are extracted and hashed into the child node. p_i is deleted in the parent node and we move the block of pages $\{p_j | j > i\}$ ahead to fill the gap. The object o can now be inserted into the child node with the new hash function. The detailed algorithm of data insertion is shown in Algorithm 7.

5.3.3 Data Deletion

The delete operation is simpler than the insert operation. Given an object o to delete, we first find the disk page where o is located in using Algorithm 8. Then, we sequentially scan the data inside it and delete the object. A simpler implementation is to maintain a bitmap with B bits for each page. Each bit indicates whether the corresponding slot is free or in use. To delete an object, we only need to set the status flag as free. In this case, to insert an object, we only need to find any free slot for the data.

If the page becomes empty after the delete operation, we do not delete it. The reason is that the overhead caused by the empty page in the query processing stage is negligible. The pages are loaded into the memory in blocks and do not require additional CPU cost when scanning the data. Moreover, in the Web 2.0 applications, insert operations are much more frequent than delete operations. There would be new objects inserted into the page in the future. Thus, we can save the overhead of removing the page cost by simply leaving the page there and reusing

it later when new objects are inserted.

5.3.4 Data Update

Given an object o to update, since its hash value could be updated as well, we can not simply update it in the data file. Instead, we treat an update operation as a deletion followed by an insertion.

5.4 Exact NN Query Processing

In this section, we present the exact NN query processing algorithm using the proposed HashFile. We adopt the lower bound distance constraint in Lemma 5.4 to prune the search space. Since we have organized the disk pages in the file in increasing order of the hash value, the candidates can be retrieved efficiently.

Algorithm 9 `ExactNN` **: Exact NN Search in HashFile**

Input: Query object q

Output: The distance δ from q to its nearest neighbor

1. init δ
2. $\delta =$ `ExactNNInNode` $(q, root, \delta)$
3. return δ

Algorithm 9 and Algorithm 10 show how exact NN search is performed in HashFile. Given a query point q, we start from the root node and recursively search the child nodes in the depth-first order. Suppose q is hashed to h_q in the current tree node, the candidate hash space is $[h_q - \frac{\delta}{W} - 1, h_q + \frac{\delta}{W} + 1]$. We check the list of child nodes whether their hash values are located in this range. The candidate child nodes are put in the heap in increasing order of their distance to h_q so that the most promising child node can be accessed first in the best first search manner. Meanwhile, we check the the disk pages to find the start offset and end offset whose hash intervals intersect with our candidate search space. The block of

pages are loaded into memory using random access and scanned sequentially. If a better result is found, we update δ accordingly. Finally, the algorithm terminates until all the candidate nodes are explored.

Algorithm 10 `ExactNNInNode` **: Exact NN Search in the HashNode**

Input: Query object q, a tree node *node* and δ
Output: The new δ

1. $h_q = node.hash(q)$
2. **for** each child node *child* **do**
3. $cdist = W(|child.hashvalue - h_q| - 1)$
4. **if** $cdist < \delta$ **then**
5. add *child* to the heap ordered by *cdist*
6. **for** each node *candiddate* in the heap **do**
7. $\delta =$ `ExactNNInNode` $(q, candidate, \delta)$
8. **for** each disk page p_i in increasing order **do**
9. find the list of pages from p_{start} to p_{end} so that their intervals intersect with $[h_q - \frac{\delta}{W} - 1, h_q + \frac{\delta}{W} + 1]$
10. load the block of pages into memory
11. **for** each object o in the block **do**
12. $dist = \| o - q \|_{L_1}$
13. **if** $dist < \delta$ **then**
14. $\delta = dist$
15. return δ

5.5 Approximate NN Query Processing

In LSH, each hash table is associated with m hash functions. Intuitively, we can consider all the buckets are organized in a height-balanced tree in which the path length from the root to the leaf nodes is always m. When m increases, the bucket size becomes smaller and it is more likely for the objects close to each other to be hashed into neighboring buckets and missed by the search algorithm. Thus, expanding the search space by inspecting neighboring buckets can significantly improve the result quality [83, 71].

Inspired by this, we propose a flexible and effective method to process approx-

imate NN query. Users can specify a parameter λ to determine the search space. If q is hashed to h_q, we only explore the neighboring buckets with hash value interval intersecting with $[h_q - \lambda, h_q + \lambda]$. Note that λ specifies the query range in the hash space instead of the number of neighboring buckets. Figure 11 shows an example in which q is hashed to B_5 with hash value 9. When $\lambda = 1$, only three buckets $\{B_4, B_5, B_6\}$ are in the search space. All the points in B_6 are scanned sequentially. Since B_4 and B_5 correspond to child nodes, they will be processed in a similar way to their parent node. When $\lambda = 3$, the search space expands to $\{B_2, B_3, B_4, B_5, B_6, B_7\}$. Since buckets $\{B_2, B_6, B_7\}$ are stored sequentially in the data file, we can load them into memory for linear scan.

Algorithm 11 and 12 illustrate how to answer an approximate NN query. The search process starts from the root node in a top-down manner and recursively explore the neighboring buckets within the search range. These candidate buckets are retrieved using sequential scan. When λ is 0, we only examine the points located within the same bucket with the query object. This is the same with the basic LSH.

Algorithm 11 `ApproximateNN` **: Approximate NN Search in HashFile**

Input: Query object q and query range λ
Output: The distance δ from q to its approximate nearest neighbor

1. init δ
2. $\delta =$ `ApproximateNNInNode` $(q, root, \delta, \lambda)$
3. return δ

5.6 Complexity and Cost Analysis

In this section, we analyze the storage utilization as well as the query time cost in answering both exact and approximate NN queries. Since the split operation partitions one page into two even parts, we have the first lemma on the storage

Algorithm 12 `ApproximateNNInNode` **: Approximate NN Search in the HashNode**

Input: Query object q, a tree node *node*, distance δ, and query range λ
Output: A new δ

1. $h_q = node.hash(q)$
2. **for** each child node *child* **do**
3. $\quad wdist = |child.hashvalue - h_q|$
4. $\quad$ **if** $wdist < \lambda$ **then**
5. $\quad\quad$ add *child* to the heap ordered by *wdist*
6. **for** each disk page p_i in increasing order **do**
7. $\quad$ find the list of pages from p_{start} to p_{end} so that their intervals intersect with $[h_q - \lambda, h_q + \lambda]$
8. load the block of pages into memory
9. **for** each object o in the block **do**
10. $\quad dist = \| o - q \|_{L_2}$
11. $\quad$ **if** $dist < \delta$ **then**
12. $\quad\quad \delta = dist$
13. return δ

utilization:

Lemma 5.5. *Given a data set P with N points, where $N \gg B$, after inserting all the data points into HashFile, we have the storage utilization rate for each page $\mu \geq 50\%$.*

Proof. We prove that $\mu \geq 50\%$ is the loop invariant during the batch insertion:

Initialization : After the first $\frac{B}{2}$ data points are inserted, there is only one

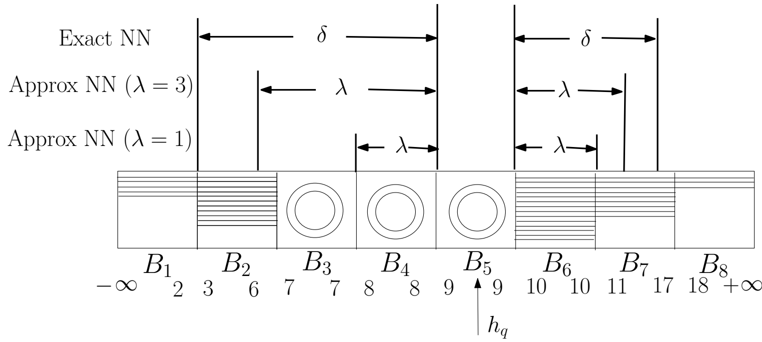


Figure 5.5: Approximate search in the tree node

page and $\mu = 50\%$.

Maintenance : When a new data is inserted, there are three cases.

1. The page has free space to host the data. The number of points in that page will increase by 1. Thus, $\mu' > \mu \geq 50\%$.

2. If the page is full and a split occurs, since the objects in the page can be evenly partitioned, we have $\mu \geq 50\%$ for the two new generated pages[1].

3. If the page is full and can not be split, a new child node is created and this page is moved to the child node's data file. Similar with case 2), when the new object is inserted to this page, a split occurs at the child node and $\mu \geq 50\%$ still holds.

Termination : When the insertion terminates, $\mu \geq 50\%$ still holds. □

5.6.1 Storage Cost

Since the storage utilization rate for each page is no less than 50%, we only need at most $2N$ space to store the data points. Also, there would be at most N/B tree nodes, the storage cost for HashFile is $O(2N + N/B)$.

5.6.2 Exact NN Query

In the worst case, the search space includes the entire HashFile when no points can be pruned. Since $N \gg B$, each node contains at least two disk pages using the even split approach, there are at most N/B tree nodes in HashFile. The exact query processing algorithm visits all the data points and the tree nodes. Hence, the cost is $O(N(1 + \frac{1}{B}))$.

[1] Counting in the new data to be inserted, we actually split $B+1$ objects into two pages and $\frac{B+1}{2} \geq \lceil \frac{B}{2} \rceil$

5.6.3 Approximate NN Query

In the approximate NN query, we usually select a small W so that the densest bucket contains a small fraction of the data size, say ϵN. Since we only visit at most $2\lambda + 1$ buckets in the root node, the total number of data accessed would be $O((2\lambda+1)\epsilon N)$, spread in $O(\frac{2(2\lambda+1)\epsilon N}{B})$ buckets. As each node contains at least two buckets, we need to access $O(\frac{(2\lambda+1)\epsilon N}{B})$ node. The cost of approximate NN query becomes $O((1+\frac{1}{B})(2\lambda+1)\epsilon N)$.

5.7 Experiments

In this section, we study the performance of HashFile and compare it with state-of-the-art approaches using real image data sets. Both exact and approximate NN query processing algorithms are evaluated. All the experiments are conducted on a server with Quad-Core AMD Opteron (tm) Processor 8356, 128GB memory, running Centos 5.4.

5.7.1 Data Set and Query

We use **NUS-WIDE** [39] as the image data set, which contains $269,648$ web images from Flickr. Two types of image features widely used in the image retrieval applications are extracted:

1. **Color Histogram.** In image processing and photography, a color histogram summarizes the distribution of colors in an image. The color space is quantized into a set of bins and the value of each bin represents the number of projected pixels by color. In practice, LAB color space [104] is normally selected as the candidate because its space is linear and suitable for quantiza-

tion. In our experiment, we extract 64-dimensional color histograms in LAB space from the image data set.

2. **SIFT.** The SIFT [81] descriptor has been widely used in image retrieval and object recognition due to its invariance with respect to translation, scaling, rotation and small distortions. Each feature is a 128-dimension vector extracted from $4 * 4$ subregions around the key point and each subregion is approximated by an 8-bin histogram of the image gradients.

We split the color histogram data set into two parts D_1 and Q_1 without intersection. 100 color histograms were randomly selected as the query. The leftover data are used as the underlying database. Similarly, we extract 10 million SIFT features from the Flcikr photos for indexing and another 200 for query.

5.7.2 Performance Measurement

The goal of the experiments is to show that the performance of our index is better than state-of-the-art query processing methods in answering exact and approximate NN queries in the high dimensional vector space. Before we present the experiment results, we first address the performance measurement used in our experiments to judge the superiority of an index.

Due to the curse of dimensionality, the pruning power degrades in the high dimensional space, resulting in a large candidate set for the NN result. We need to load these candidates into memory and calculate their distance to the query object. Both disk I/O cost and CPU computation time play an important role in the query processing stage. Since HashFile takes advantage of sequential scan, it may be unfair to measure the I/O cost simply by the number of page access or disk access. To make a fair competition, we use average response time (ART) to

measure the performance of an index in answering an exact NN query.

$$ART = (1/N_Q)\sum_{i=1}^{N_Q}(T_f - T_i)$$

where T_i is the time at which the query was issued, T_f is the time of query completion and N_Q is the total number of times exact NN query was issued. The ART is equivalent to the elapsed time including both I/O and CPU cost. Besides the ART, we also report the selectivity of different indexes in the query processing. If we assume the time cost of calculating the distance between a candidate to the query object is fixed as t, the CPU cost can be measured by the selectivity.

$$C_{cpu} = N * s * t$$

, where N is the data size and s is the average query selectivity.

In the approximate NN query, the quality of the result is usually represented by the distance ratio between the approximate NN and the exact NN. For top-k approximate NN query, we can adopt the same metric as in [105]:

$$\begin{aligned} R_i(q) &= \frac{\|o_i, q\|}{\|o_i^*, q\|}, 1 \leq i \leq k \\ R(q) &= \frac{\sum_{i=1}^{k} R_i}{k} \end{aligned}$$

where o_i is i-th approximate neighbor and o_i^* is the exact neighbor. Obviously, the smaller the value of $R(q)$, the better is the quality of results retrieved via approximate NN query. If $R(q)$ equals to 1, the query results are exact. However, the distance recall is not enough to measure the performance of an index as there is a trade-off between the access cost and the result quality. If more data are accessed in the buckets, a smaller $R(q)$ can be retrieved. Hence, we need to consider both

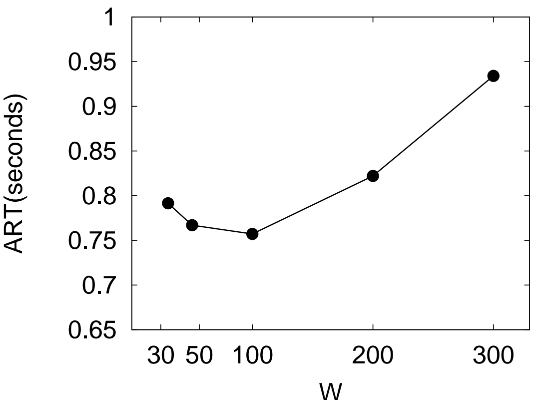


Figure 5.6: Tune parameter W

factors in the performance measurement. Since LSB-tree uses random access and our index takes advantage of sequential scan, we still use ART to measure the access cost. We plot the curve of $R(q)$ as the y axis with respect to the ART as the x axis. The curve drops down as the ART increases, meaning that $R(q)$ is getting close to 1. We use the area under the curve as the measurement. If an index can retrieve an approximate result closer to the exact neighbors in less running time, we consider it to be superior.

5.7.3 Parameter Tuning

In our first experiment, we study how the performance varies with the only parameter W. The page size B is fixed as 100. We randomly select a subset with 1 million SIFT descriptors from D_2 as the data set and test the performance of the exact NN search algorithm.

The result in Figure 5.6 shows that the performance degrades when W is set too large or too small. If W is chosen to be very large, a lot of data points will be hashed into the same bucket. It becomes easy for a bucket to be overloaded and generate new child nodes. Therefore, the tree becomes higher and more random

accesses are needed to retrieve the tree nodes. Also, there are fewer disk pages in each node's file. The advantage of sequential scan is offset in this case. On the other hand, if W is set to be very small, the strategy of further partitioning the dense bucket becomes useless. The number of points in the buckets is small as well and the re-hashing does not take much effect.

5.7.4 Frequent Insertion

HashFile needs to maintain the sequential order of disk pages based on the hash value. When a split operation occurs in a full page p_i, we need to move the pages $\{p_j | j > i\}$ backwards to make room for the new page. Similarly, when p_i contains objects with the same hash value and can not be split using the node's hash function, we need to delete p_i from the file and move forward the pages $\{p_j | j > i\}$ that are behind to fill the gap. We consider the page movement as a sequence of two disk operations: the block of pages $\{p_j | j > i\}$ is first read into the memory and then written to the target location in the disk file. The I/O cost is determined by the size of the block to be moved.

Figure 5.7 shows an example of how the number of disk pages in the root varies when data points are continuously inserted. In this example, we use the color histogram data set and set the page size to 100. From this curve, we can tell that the number of pages in the data file is always in a small scale. The first split occurs when the 101-st point arrived. After that, the pages in the root are split frequently and the number of pages in the file increases dramatically. As more data points are inserted, the buckets near the mean value can not be split anymore. These buckets are extracted from the file and re-hashed into the child nodes. Thus, the number of pages starts to decrease. Finally, this number becomes relatively stable, varying between 3 to 6, because the buckets near the mean hash value have been deleted

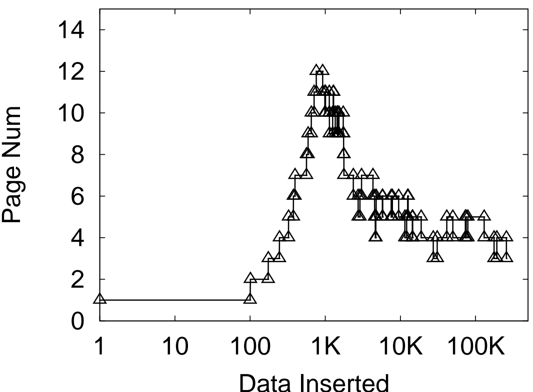


Figure 5.7: The number of pages in the root node

and new objects are more likely to be inserted into the child nodes. Therefore, the file size does not grow too large and cause too much I/O overhead in the page movement.

5.7.5 Exact NN Query

In this experiment, we compare HashFile to linear scan, VA-file, and iDistance. The parameter setting for each type of index is shown in Table 5.2. We tune the bit number for each dimension in the VA-file in the range of $[4, 6]$ and select the best one to build the index. The number of reference points in iDistance is set to $2d$. The window size of W in HashFile is also set differently according to the data set. The exact NN query is executed on both color histogram and the SIFT descriptor. We gradually increase the data set in D_1 from $180,000$ to $260,000$ and in D_2 from 2 million to 10 million.

Table 5.3 shows the disk storage used for each of the three types of index. VA-file takes up the least storage cost. Besides the real data points, it only maintains the bitmap summary which can be stored efficiently. iDistance needs to build an additional B^+-tree and store the real data in the leaf entries. Since the page can not

Table 5.2: Parameter Setting

	VA-File	iDistance	HashFile
D_1	bit=5	ref=128	W=20
D_2	bit=4	ref=256	W=100

be fully utilized, it takes much more storage cost than VA-file. HashFile consumes slightly larger storage than iDistance. But it is still under the bound $O(2N+N/B)$.

Table 5.3: Index storage cost

Color Histogram(MB)					
Data Size	180K	200K	220K	240K	260K
VA-file	49.4	54.9	60.4	65.9	71.4
iDistance	72	80	87	95	104
HashFile	74	82	89	96	104
SIFT Descriptor(GB)					
Data Size	2M	4M	6M	8M	10M
VA-file	1.125	2.25	3.375	4.5	5.625
iDistance	1.6	3.1	4.6	6.1	7.6
HashFile	1.8	3.5	5.3	7.0	9.2

The pruning power of the three types of index in answering exact top-50 NN queries is reported in Table 5.4. It counts the proportion of the real data points accessed in the query processing stage. VA-file has the best selectivity because the data space is split into exponential number of small cells and the real data points are tightly approximated by the cells. After comparing the query point to the bitmap file, only a very small number of real data need to be accessed. The pruning power of iDistance is mainly determined by the selection of the reference points. When we use the cluster center to partition the space into Voronoi cells, iDistance demonstrates a better pruning power than HashFile. The pruning of iDistance is based on real distance while HashFile is based on the lower bound distance constraint in the hash space. All the data points in the same page are

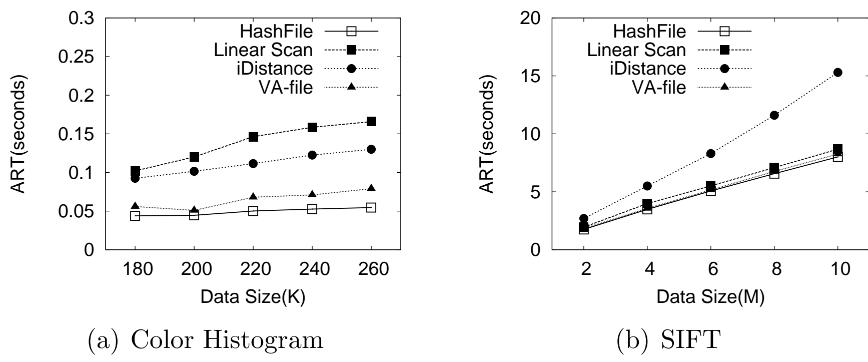
sequentially scanned without any pruning. As d increases from 64 to 128, we can see that the pruning power of VA-file becomes even better. However, iDistance and HashFile need to access the majority of the data set.

Table 5.4: Top-50 NN query selectivity

Color Histogram					
Data Size	180K	200K	220K	240K	260K
VA-file	0.00281	0.00275	0.00269	0.00264	0.00260
iDistance	0.217	0.205	0.2	0.193	0.182
HashFile	0.301	0.291	0.285	0.278	0.268
SIFT Descriptor					
Data Size	2M	4M	6M	8M	10M
VA-file	0.00034	0.00024	0.00021	0.00019	0.00018
iDistance	0.841	0.769	0.696	0.645	0.599
HashFile	0.918	0.911	0.902	0.889	0.867

The average running time to answer the queries is shown in Figure 5.8. Since the color histogram data set is skewed, iDistance shows good pruning power and its performance is better than linear scan. But when it comes to the SIFT data set, which is more uniformly distributed, iDistance needs to access the majority of the pages and performs worst. Such a large amount of random access makes the index I/O bound. VA-File outperforms linear scan in both two data sets. It has extremely low selectivity in the query processing and the operations to calculate the minimum and maximum bound distance from the query point to the cells are optimized in the implementation to greatly save the CPU cost. HashFile adopts random projection to gain the pruning power and takes advantage of sequential scan to reduce the number of random access. It achieves significant superiority over the other index structures in the color histogram data set, which is skewed to black and white colors. In the SIFT data set, the data is more uniformly distributed and HashFile needs to access a large population of the data. Comparing Figure 5.8(b) with

Figure 5.8(b), we can see that the performance of high dimensional index start to degrade to linear scan as the dimensionality increases. This makes approximate NN search an appealing method to handle similarity search in a large scale multimedia database.



(a) Color Histogram (b) SIFT

Figure 5.8: Performance of the exact top-50 NN search

5.7.6 Approximate NN Query

In this experiment, we test the performance of HashFile in answering the approximate NN query. We use the LSB trees [105] as the comparison method. In order to plot the curve of $R(q)$ with respect to the access cost for the LSB trees, we do not terminate the search algorithm based on the two conditions proposed in [105]. Instead, we set a parameter I as the number of iteration to execute. We gradually increase I so that more and more leaf entries will be accessed. With the increasing access cost, the algorithm returns the nearest neighbors with smaller $R(q)$. In this way, we can plot how the quality of the result varies with the running time. As to HashFile, we set a small window and gradually increase the parameter λ. Each time λ is increased, we can access more disk pages to find a better nearest neighbor. The experiment is conducted on D_1 with $260,000$ color histograms and D_2 with two million descriptors.

Table 5.5: Storage cost of HashFile and LSB forest

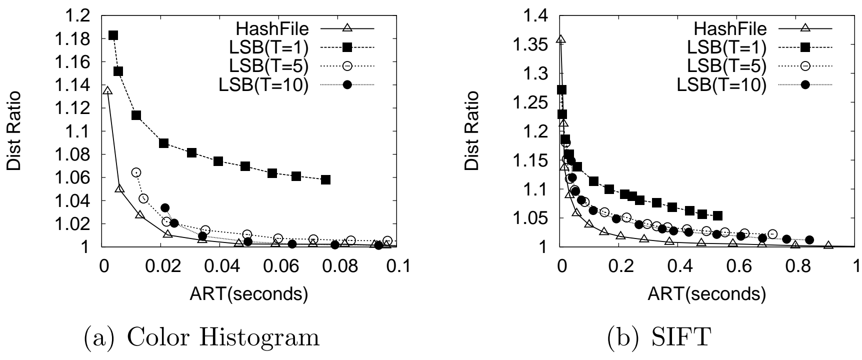
	Color Histogram	SIFT Descriptor
HashFile	110MB	1.6GB
LSB(1 tree)	107MB	1.6GB
LSB(5 trees)	539MB	8GB
LSB(10 trees)	1094MB	16GB

We computed the overall disk storage costs for the LSB forest with varying number of trees. Table 5.5 depicts the storage cost for different variants. The size of HashFile is basically the same with that of one LSB tree. The size of the LSB-forest grows linearly with the number of trees. In particular, when 10 trees are used, the storage cost of LSB is almost 10 times that of HashFile.

Figure 5.9 shows the tradeoff between $R(q)$ and ART on the top-20 nearest neighbors in D_1 and D_2 respectively. As more data are accessed, $R(q)$ declines to be near 1, indicating that better results are found. We can clearly tell from the figure that HashFile has significant superiority than LSB forests. Even though LSB forests consume much more storage cost than HashFile, when the number of trees increases to 10, its performance is still dominated by HashFile. Given a point in the curve of 10 LSB trees, we can always find another point in HashFile curve so that the query is processed with less running time but with better $R(Q)$. This validates the superiority of linear scan to random access.

Finally, we compare the search quality of HashFile with LSH. In HashFile, only the dense buckets are further partitioned and the data points are associated with hash values of variable length. In contrast, the length is fixed as m when LSH is used. We use multi-probe LSH [2] as the comparison method because it is also adhoc and without theoretical guarantee. In this experiment, we set $m = \kappa$, i.e.,

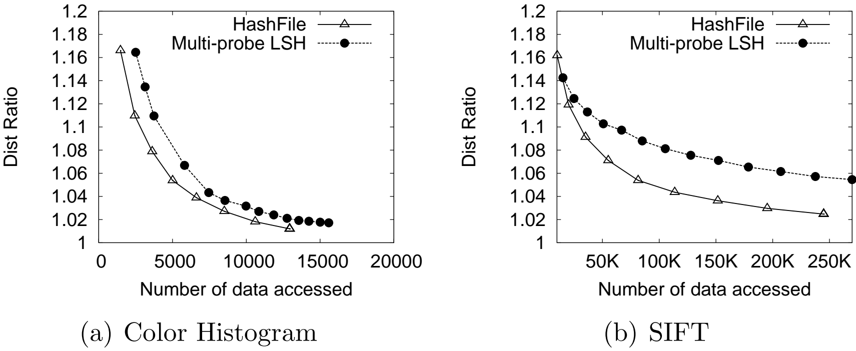
[2] http://lshkit.sourceforge.net/



(a) Color Histogram (b) SIFT

Figure 5.9: Approximate NN query results of LSB-tree and HashFile

the number of hash functions m is equal to the tree height of HashFile. We also tuned the parameter W in these two indexes to generate roughly the same number of buckets. Since multi-probe LSH is an in-memory index, the search quality is measured by the distance ratio with respect to the number of data accessed. We gradually increase the number of probe in multi-proble LSH and λ in HashFile to expand the search space. The experiment results on the color histogram and SIFT data sets are shown in Figure 5.10. Since the distribution of bucket size in LSH is much more skewed than HashFile, many false positives still exist in the dense buckets. Accessing these false positives will reduce the search quality. HashFile has more balanced bucket size and demonstrates better search quality.



(a) Color Histogram (b) SIFT

Figure 5.10: Approximate NN query results of Multi-probe LSH and HashFile

5.8 Summary

In this chapter, we proposed a novel index structure, HashFile, to handle nearest neighbor queries in the high dimensional vector space which are needed to support object retrieval in multimedia databases. The index can support approximate NN search in the Euclidean space and exact NN search in L_1 norm. Users can select the query strategy based on their applications. HashFile is simple in structure and therefore is easy to be implemented into the real system and applications. It provides better efficiency in processing both two types of NN queries. Experiments were conducted on real data sets to establish the superiority of Hashfile over other state-of-the-art index structures.

CHAPTER 6

LANGG : A TRAVEL MASHUP SYSTEM FOR LOCATION-BASED SERVICES

In this chapter, we introduce the framework and design issues of our travel mashup system which provides location-based services. We also demonstrate the system to show how each locating application is implemented. We target our system at travel market in Singapore and crawl relevant travel data mainly from Foursquare[1], Flickr[2] and TripAdvisor[3] to build a spatial database. User feedback shows that our system provides satisfactory search results.

[1]https://foursquare.com/
[2]http://www.flickr.com/
[3]http://www.tripadvisor.com/

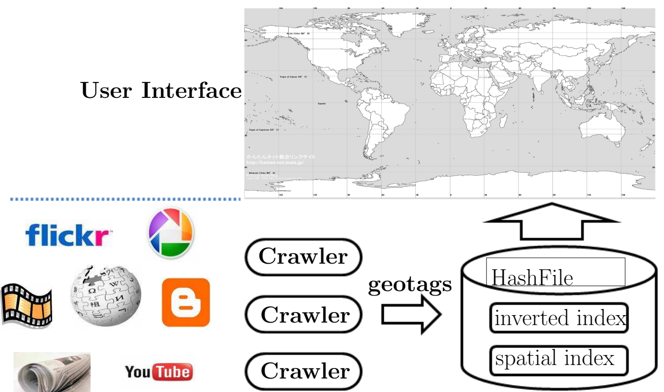


Figure 6.1: The framework of location detecting in Web 2.0 applications

6.1 System Framework

The emergence of Web 2.0 has resulted in the concept of mashup where data objects from multiple external sources are combined to create a new service. In other words, mashup is a hybrid web application built on top of resources from different channels. In this section, we describe a new type of mapping mashup that integrates data sources from various Web 2.0 applications to support travel resource locating. The framework exhibits the following features:

- A wide range of data sources from different Web 2.0 applications are combined.
- All the resources are represented in a uniform data model to facilitate the indexing and searching process.
- A simple, map-based user interface is designed to provide users with satisfactory experience in searching and browsing the geographical resources.

Figure 6.1 shows the map mashup framework. It consists of three components: 1) the index engine to crawl, integrate and index source data. 2) query engine to find the location of the resources. 3) friendly interface to improve user experience.

Index Engine

The index engine aims at supporting efficient tag matching and image locating on top of the combined data resources derived from other applications, such as Wikipedia, Flickr, Picasa Web Albums, and Youtube. In Wikipedia, most of the articles with geographical context have been geo-located. For instance, the page "Forbidden City" is associated with coordinates $39°54'53''N$ and $116°23'26''E$. This location information can be parsed as the spatial attribute of the article. In online photo sharing applications like Flickr and Google Web Picasa, APIs have been published to access the public albums, photos, tags as well as their locations. Similarly, when users share their videos in Youtube or other online video sharing websites, they may mark on the map the location where the video was shot. Hence, we are able to retrieve mapped resources of different types to build a spatial database. In our implementation, we crawl travel services from Foursquare, geo-tags from Flickr and photos of sight attractions from TripAdvisor. After the retrieval step, we need to combine the data sources. We can use the uniform resource mapped resource model to seamlessly integrate the articles, photos and videos. It can provide a transparent access layer and benefit the indexing and searching process. SIFT features are also extracted from the photos and indexed using HashFile.

The Query Engine and User Interface

Based on our uniform resource model, the query interface allows users to submit query in different formats. Our system supports three types of input, including desired travel services, tags and travel photos. Local search engines can benefit greatly from such queries as these can help them to provide better customized service.

The search engine is designed to be simple and friendly. The whole interface is

Figure 6.2: System portal of **LANGG**

map-based. Users can freely explore geo-resources in the area they are interested in. Different search portals are provided for different types of input. All the related resources will be displayed directly on the map for further refinement.

6.2 Demonstration

The main portal of our system is shown in Figure 6.2. The logo is Explore Singapore as our system is targeted at Singapore travel market. There are there buttons indicating the three types of locating services that our system supports. Users can select the locating function they are interested in by clicking the corresponding button. In the following, we will explicitly explain the implementation of each locating application.

6.2.1 Search Closest Travel Services

We crawl 45 popular local services from Foursquare in Singapore to benefit travellers. The services are divided into five categories, including Restaurant, Shop, Night Club, Entertainment and MISC. Since restaurant is a very popular travel service, we cover various types of restaurant in our database. As we can see from

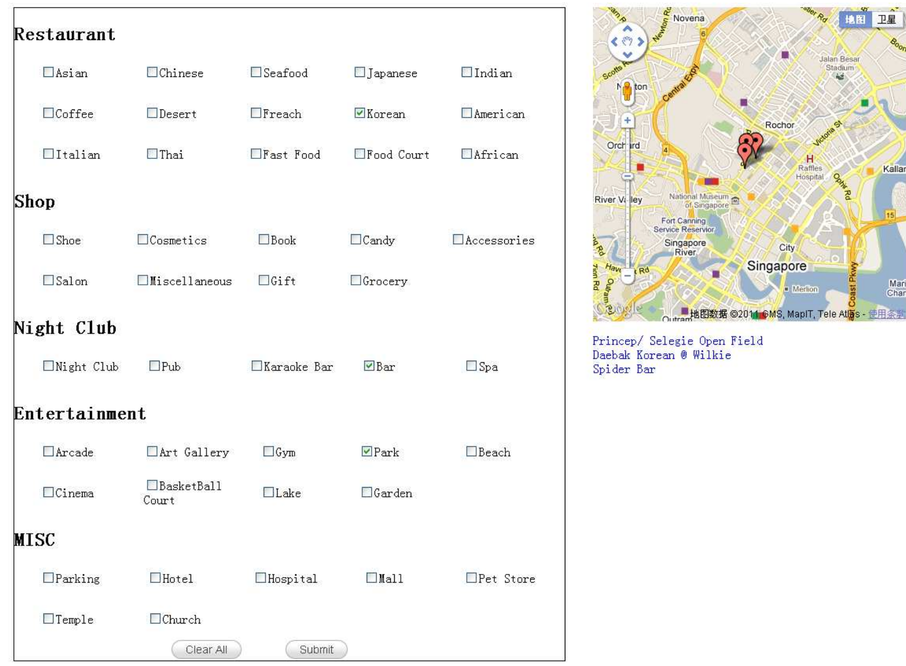


Figure 6.3: Interface of locating closest travel services

Figure 6.3, our system supports search for restaurants from all over the world including Asian, Chinese, Japanese, Indian, French, American, Italian and African. We also provide Fast Food, Coffee and Dessert for special needs. To use the locating function, users just check their desired services from the list and click "Submit" button. The search result will be displayed on the right-hand, as shown in Figure 6.3. The closest services are displayed on Google Maps with markers indicating their locations. Textual description of these entities are also provided under the map block. Users can click them for more detailed information in Foursquare. To start a new search, they can click the "Clear All" and re-check the query services.

6.2.2 Search Location Using Tags

In our system, the geo-tags are derived from Flickr images. When users upload their photos to Flickr, some of them would also mark the shooting location of each

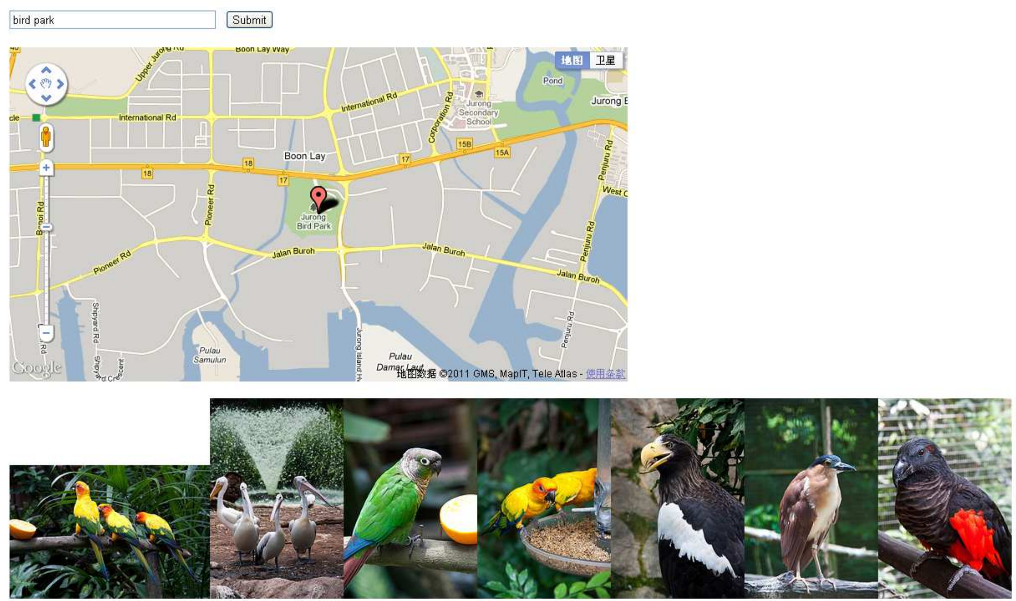


Figure 6.4: Query by “bird park”

photo on the map and annotate it with tags as description or summary. We crawl 4,000 photos with geo-tags as our underlying database. Although the database is not large, it can support simple queries that are relevant to some popular travel resources in Singapore. Figure 6.4 and 6.5 show two query examples of “bird park” and “chicken rice”. The location and related Flickr photos are displayed to show the relevance.

6.2.3 Search Location by Image

We crawl travel photos that are associated with most popular sight attractions from TripAdvisor. There are in total 2,228 photos which generate around 4 million SIFT features. We use HashFile to index these features and adopt approximate nearest neighbor search for image similarity retrieval. In our search interface, users can submit a query photo either from an external url or from their local file system. When the photo is uploaded, we first extract its SIFT features. Then, for each

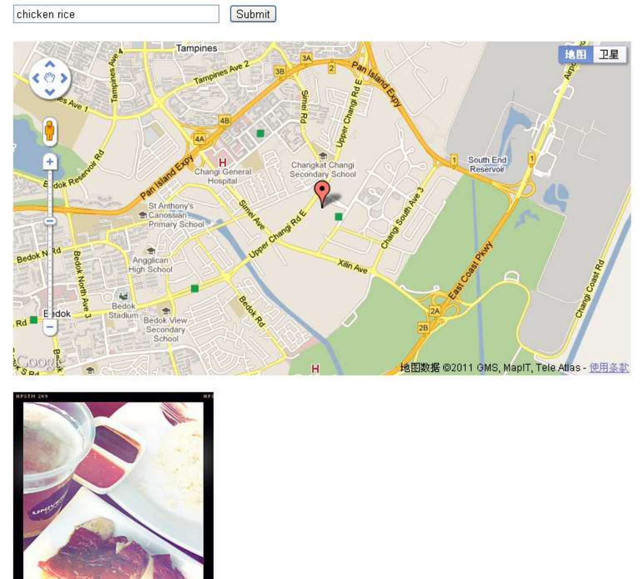
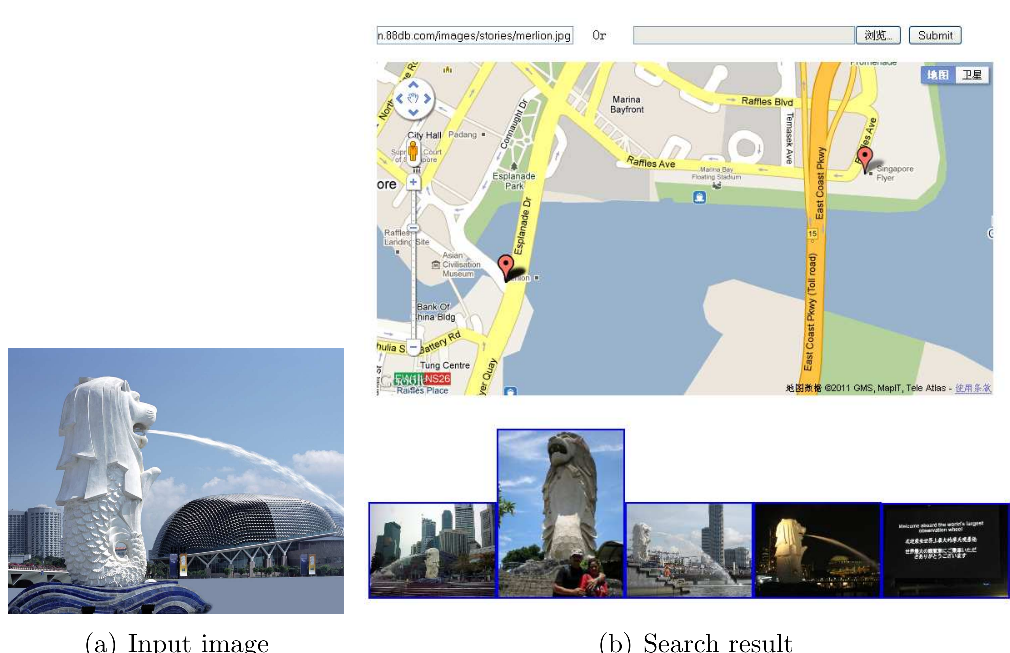


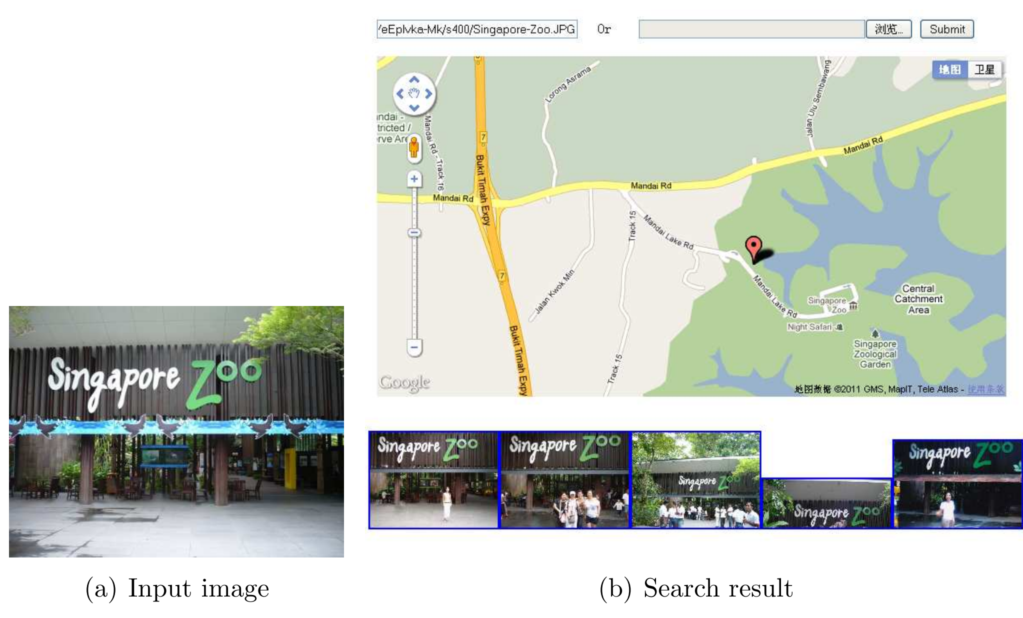
Figure 6.5: Query by "chicken rice"

feature, we check which bucket it is located in our HashFile index. Features in that bucket are retrieved as matching candidates. If an image has more matching features with the query image, we consider it more similar and assign it with higher score. We illustrate two query examples in Figure 6.6(a) and 6.7(a) about Singapore Merlion and Singapore Zoo. The search results for these two queries, including top-5 matching images as well as their location on the map are shown in Figure 6.6(b) and 6.7(b) respectively. From the results, we can see that HashFile provides satisfactory results for famous landmark recognition.



(a) Input image (b) Search result

Figure 6.6: Example image query of Merlion



(a) Input image (b) Search result

Figure 6.7: Example image query of Zoo

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Map mashup is a popular and convenient means to integrate and visualize various types of web resources, including documents, photos and videos. In this thesis, we proposed efficient spatial keyword query processing technique and built a new map mashup system to provide users with location-based services.

First, we addressed a novel query, named mCK query, in Chapter 3 to retrieve the closest set of objects matching m specified query keywords. We used keyword to represent travel service and applied mCK query to find a location on the map where a set of travel services are closest to each other. Such an application is specially useful to save the transportation cost when a traveller only has limited staying time in a city. Since the search space of mCK query is exponential to the number of travel services and the size of database, an efficient solution is typically required. We built a bR*-tree which effectively summarizes keyword locations to

facilitate pruning. We also proposed effective a priori-based search strategies for mCK query processing and discussed two monotone constraints with efficient implementation. Our performance study on both synthetic and real data sets shows that our proposed bR*-tree answers mCK queries efficiently within relatively short query response time. Furthermore, it demonstrates remarkable scalability in terms of the number of query keywords. It significantly outperforms the existing MWSJ approach when m is large.

Besides travel services, we also proposed efficient solutions to detecting the geographical context of web media. If the media is associated with user-generated tags, we developed a new detecting method to support co-location searches by tag matching in Chapter 4. We built a labelled R*-tree and inverted lists to index the spatial and textual attributes of web resources in the underlying database. We developed efficient search strategies to answer the tag matching query. To further improve the matching precision, we proposed a new *geo-tf-idf* ranking mechanism to measure the geographical relevance of query tags with respect to the detected location. Extensive experiments using both synthetic and real life data sets confirm the feasibility and efficiency of our proposed design in the Web 2.0 environment.

If the media is a raw travel blog or photo, content-based matching method can be used. If the input is a textual document, there have been quite a few related works about detecting geographical context of web documents and we could adopt existing approaches to solve the problem. If the input is a travel photo, we used nearest neighbor(NN) search to find the most similar image as the match result. To efficiently support NN query, we proposed a novel index structure, HashFile, in Chapter 5 to handle nearest neighbor queries in the high dimensional vector space. The index can support approximate NN search in the Euclidean space and exact NN search in L_1 norm. Users can select the query strategy based on their applications.

HashFile is simple in structure and therefore is easy to be implemented into the real system and applications. It provides better efficiency in processing both the two types of NN queries. Experiments conducted on real data sets established the superiority of HashFile over other state-of-the-art index structures.

7.2 Future work

Although our system provides efficient solutions to location detecting services, there still exist situations that it can not handle well. The bR*-tree in Chapter 3 integrates all the keyword information inside the tree node. When there exist a large number of keywords in the database, the node size could become very large and the I/O access to tree nodes could turn into the bottleneck of performance. Hence, the performance is not scalable in terms of the number of travel services. Although we proposed a new virtual bR*-tree in Chapter 4 which combines the advantage of R*-tree and inverted index and demonstrates good scalability, it incurs considerable update cost when the insertion frequency of new objects is high. Each time a node in the R*-tree is split after an insertion, the label of its descendants will change and all the related elements in the inverted lists need to be updated as well. Therefore, the design of a scalable and easy-to-maintain spatial index for answering mCK query still remains an interesting area for future work.

There are also two dimensions to extend mCK query to more useful applications. The first one is to add spatial constraint to support mobile applications. For example, users may be more interested to find closest tuples around their locations. The other extension is to support partial mCK query. When the number of keywords is large, the distance of the results could be too large to be meaningful. In this case, we may need to select subset of query keywords in the query processing.

If web resource is associated with noisy tags or the number of query tags is too large, it may not be meaningful to return a location matching all the query tags. In this case, a partial tag match would be a desired solution. The place that matches only a subset of the query tags can be returned. Hence, the design of a new spatial index and ranking strategy for partial tag match remains another interesting area.

Finally, our HashFile is efficient for image match based on color histogram. Although SIFT feature has been widely used due to its invariance with respect to translation, scaling, rotation and small distortions, we can not directly apply our index for SIFT data as each image contains hundreds of SIFT descriptors. How to efficiently support high dimensional set similarity is still an on-going research problem.

BIBLIOGRAPHY

[1] Columbia geosearch. http://geosearch.cs.columbia.edu.

[2] Google local search. http:/www.google.com/local.

[3] http://en.wikipedia.org/wiki/exchangeable_image_file_format.

[4] http://en.wikipedia.org/wiki/mashup_

[5] http://en.wikipedia.org/wiki/web_2.0.

[6] http://geonames.usgs.gov.

[7] http://twitter.com/facebook/status/22372857292005376.

[8] http://www.lkozma.net/wpv/.

[9] http://www.mediawiki.org/wiki/api.

[10] http://www.usatoday.com/tech/news/2006-07-16-youtube-views_x.htm.

[11] http://www.world-gazetteer.com.

[12] United nations department of economic and social affairs. http://www.world-gazetteer.com.

[13] Usps - the united states postal services. http://www.usps.com.

[14] Yahoo regional. http://www.yahoo.com/regional.

[15] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.

[16] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 420–434, London, UK, 2001. Springer-Verlag.

[17] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[18] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[19] Sattam Alsubaiee, Alexander Behm, and Chen Li. Supporting location-based approximate-keyword queries. In *GIS*, pages 61–70, 2010.

[20] Sattam Alsubaiee and Chen Li. Fuzzy keyword search on spatial data. In *DASFAA (2)*, pages 464–467, 2010.

[21] Einat Amitay, Nadav Har'El, Ron Sivan, and Aya Soffer. Web-a-where: geo-tagging web content. In *SIGIR '04: Proceedings of the 27th annual interna-*

tional ACM SIGIR conference on Research and development in information retrieval, pages 273–280, New York, NY, USA, 2004. ACM.

[22] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *SIGMOD Conference*, pages 347–358, 2004.

[23] Saeid Asadi, Guowei Yang, Xiaofang Zhou, Yuan Shi, Boxuan Zhai, and Wendy Wen-Rong Jiang. Pattern-based extraction of addresses from web page content. In *Proceedings of the 10th Asia-Pacific web conference on Progress in WWW research and development*, APWeb'08, pages 407–418, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Saeid Asadi, Xiaofang Zhou, and Guowei Yang. Using local popularity of web resources for geo-ranking of search engine results. *World Wide Web*, 12(2):149–170, 2009.

[25] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[26] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[27] Stefan Berchtold, Christian Böhm, H. V. Jagadish, Hans-Peter Kriegel, J??rg Sander, and J??rg S. Independent quantization: An index compression technique for high-dimensional data spaces. In *In ICDE*, pages 577–588, 2000.

[28] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. In *Proceedings of*

the 1998 ACM SIGMOD international conference on Management of data, SIGMOD '98, pages 142–153, New York, NY, USA, 1998. ACM.

[29] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[30] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, pages 217–235, 1999.

[31] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[32] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, New York, NY, USA, 2001. ACM.

[33] Karla A. V. Borges, Alberto H. F. Laender, Claudia B. Medeiros, and Clodoveu A. Davis, Jr. Discovering geographic locations in web pages using urban addresses. In *Proceedings of the 4th ACM workshop on Geographical information retrieval*, GIR '07, pages 31–36, New York, NY, USA, 2007. ACM.

[34] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 237–246, New York, NY, USA, 1993. ACM.

[35] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

[36] Orkut Buyukkokten, Junghoo Cho, Hector Garcia-Molina, Luis Gravano, and Narayanan Shivakumar. Exploiting geographical location information of web pages. In *WebDB (Informal Proceedings)*, pages 91–96, 1999.

[37] Rui Cai, Chao Zhang, Lei Zhang, and Wei-Ying Ma. Scalable music recommendation by search. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 1065–1074, New York, NY, USA, 2007. ACM.

[38] Xin Cao, Gao Cong, and Christian S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[39] Tat-Seng Chua, Jinhui Tang, Richang Hong, Haojie Li, Zhiping Luo, and Yan-Tao. Zheng. Nus-wide: A real-world web image database from national university of singapore. In *Proc. of ACM Conf. on Image and Video Retrieval (CIVR'09)*, Santorini, Greece., July 8-10, 2009.

[40] Ondrej Chum, James Philbin, and Andrew Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, 2008.

[41] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml, 2003.

[42] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[43] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD Conference*, pages 189–200, 2000.

[44] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, 49(1):67–104, 2004.

[45] David J. Crandall, Lars Backstrom, Daniel Huttenlocher, and Jon Kleinberg. Mapping the world's photos. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 761–770, New York, NY, USA, 2009. ACM.

[46] Silviu Cucerzan. Large-scale named entity disambiguation based on wikipedia data. In *EMNLP-CoNLL*, pages 708–716, 2007.

[47] Junyan Ding, Luis Gravano, and Narayanan Shivakumar. Computing geographical scopes of web resources. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 545–556, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[48] Wei Dong, Zhe Wang, Moses Charikar, and Kai Li. Efficiently matching sets of features with random histograms. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 179–188, New York, NY, USA, 2008. ACM.

[49] Joel L. Fagan. Automatic phrase indexing for document retrieval: An examination of syntactic and non-syntactic methods. In *SIGIR*, pages 91–101, 1987.

[50] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[51] Katerina T. Frantzi. Incorporating context information for the extraction of terms. In *ACL*, pages 501–503, 1997.

[52] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[53] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[54] Richard Göbel, Andreas Henrich, Raik Niemann, and Daniel Blank. A hybrid index structure for geo-textual searches. In *CIKM*, pages 1625–1628, 2009.

[55] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents, 2003.

[56] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[57] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSDBM*, page 16, 2007.

[58] James Hays and Alexei A. Efros. im2gps: estimating geographic information from a single image. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2008.

[59] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[60] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA, 1998. ACM.

[61] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[62] William B. Johnson and Joram Lindenstrauss. Extensions of lipschitz mapping into hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[63] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[64] Norio Katayama and Shin'ichi Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 369–380, New York, NY, USA, 1997. ACM.

[65] Yan Ke, Rahul Sukthankar, and Larry Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 869–876, New York, NY, USA, 2004. ACM.

[66] Yan Ke, Rahul Sukthankar, Larry Huston, Yan Ke, and Rahul Sukthankar. Efficient near-duplicate detection and sub-image retrieval. In *In ACM Multimedia*, pages 869–876, 2004.

[67] Lyndon S. Kennedy and Mor Naaman. Generating diverse and representative image search results for landmarks. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 297–306, New York, NY, USA, 2008. ACM.

[68] Ali Khodaei, Cyrus Shahabi, and Chen Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA (1)*, pages 450–466, 2010.

[69] Jon M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *STOC*, pages 599–608, 1997.

[70] Nick Koudas, Beng Chin Ooi, Heng Tao Shen, and Anthony K. H. Tung. Ldc: Enabling search by partial distance in a hyper-dimensional space, 2004.

[71] Yin-Hsi Kuo, Kuan-Ting Chen, Chien-Hsing Chiang, and Winston H. Hsu. Query expansion for hash-based image object retrieval. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 65–74, New York, NY, USA, 2009. ACM.

[72] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 614–623, New York, NY, USA, 1998. ACM.

[73] Chen Li, Bin Wang, and Xiaochun Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[74] Guoliang Li, Jianhua Feng, Feng Lin, and Lizhu Zhou. Progressive ranking for efficient keyword search over relational databases. In *BNCOD*, pages 193–197, 2008.

[75] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD Conference*, pages 903–914, 2008.

[76] Yunpeng Li, David J. Crandall, and Daniel P. Huttenlocher. Landmark classification in large-scale image collections. *International Conference on Computer Vision*, 2009.

[77] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The tv-tree: An index structure for high-dimensional data. Technical Report 4, 1994.

[78] Fang Liu, Clement T. Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[79] Yiming Liu, Dong Xu, Ivor W. Tsang, and Jiebo Luo. Using large-scale web data to facilitate textual query based retrieval of consumer photos. In *ACM Multimedia*, pages 55–64, 2009.

[80] Ziyang Liu and Yi Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[81] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[82] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[83] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[84] Nikos Mamoulis and Dimitris Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4):424–475, 2001.

[85] Alexander Markowetz, Yen-Yu Chen, Torsten Suel, Xiaohui Long, and Bernhard Seeger. Design and implementation of a geographic search engine. In *WebDB*, pages 19–24, 2005.

[86] Kevin S. McCurley. Geospatial mapping and navigation of the web. In *WWW*, pages 221–229, 2001.

[87] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *PODS*, pages 214–221, New York, NY, USA, 1993. ACM.

[88] Dimitris Papadias and Dinos Arkoumanis. Search algorithms for multiway spatial joins. *International Journal of Geographical Information Science*, 16(7):613–639, 2002.

[89] Dimitris Papadias, Nikos Mamoulis, and Yannis Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *PODS*, pages 44–55, 1999.

[90] Ho-Hyun Park, Guang-Ho Cha, and Chin-Wan Chung. Multi-way spatial joins using r-trees: Methodology and performance evaluation. In *SSD*, pages 229–250, 1999.

[91] Ross S. Purves, Paul Clough, Christopher B. Jones, Avi Arampatzis, Benedicte Bucher, David Finch, Gaihua Fu, Hideo Joho, Awase Khirni Syed, Subodh Vaid, and Bisheng Yang. The design and implementation of spirit: a spatially aware search engine for information retrieval on the internet. *Int. J. Geogr. Inf. Sci.*, 21(7):717–745, 2007.

[92] Arun Qamra and Edward Y. Chang. Scalable landmark recognition using extent. *Multimedia Tools Appl.*, 38(2):187–208, 2008.

[93] Till Quack, Bastian Leibe, and Luc Van Gool. World-scale mining of objects and events from community photo collections. In *CIVR '08: Proceedings of the 2008 international conference on Content-based image and video retrieval*, pages 47–56, New York, NY, USA, 2008. ACM.

[94] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, New York, NY, USA, 1981. ACM.

[95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.

[96] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proceedings of the 26th International Conference on*

Very Large Data Bases, VLDB '00, pages 516–526, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[97] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[98] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[99] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[100] Mehdi Sharifzadeh and Cyrus Shahabi. Processing optimal sequenced route queries using voronoi diagrams. *GeoInformatica*, 12(4):411–433, 2008.

[101] Börkur Sigurbjörnsson and Roelof van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 327–336, New York, NY, USA, 2008. ACM.

[102] Amit Singhal, Chris Buckley, and Mandar Mitra. Pivoted document length normalization. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–29, New York, NY, USA, 1996. ACM.

[103] Benno Stein. Principles of hash-based text retrieval. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 527–534, New York, NY, USA, 2007. ACM.

[104] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[105] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 563–576, New York, NY, USA, 2009. ACM.

[106] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3), 2010.

[107] Takashi Tomokiyo and Matthew Hurst. A language model approach to keyphrase extraction. In *Proceedings of the ACL 2003 workshop on Multiword expressions: analysis, acquisition and treatment - Volume 18*, MWE '03, pages 33–40, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[108] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[109] Chuang Wang, Xing Xie, Lee Wang, Yansheng Lu, and Wei-Ying Ma. Detecting geographic locations from web resources. In *GIR '05: Proceedings of the 2005 workshop on Geographic information retrieval*, pages 17–24, New York, NY, USA, 2005. ACM.

[110] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on*

Very Large Data Bases, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[111] Jeremy Witmer and Jugal Kalita. Extracting geospatial entities from wikipedia. In *Proceedings of the 2009 IEEE International Conference on Semantic Computing*, ICSC '09, pages 450–457, Washington, DC, USA, 2009. IEEE Computer Society.

[112] Ian H. Witten, Gordon W. Paynter, Eibe Frank, Carl Gutwin, and Craig G. Nevill-Manning. Kea: Practical automatic keyphrase extraction. In *ACM DL*, pages 254–255, 1999.

[113] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[114] Yin Yang, Nilesh Bansal, Wisam Dakka, Panagiotis G. Ipeirotis, Nick Koudas, and Dimitris Papadias. Query by document. In *WSDM*, pages 34–43, 2009.

[115] Bin Yao, Feifei Li, Marios Hadjieleftheriou, and Kun Hou. Approximate string search in spatial databases. In *ICDE*, pages 545–556, 2010.

[116] Seiji Yokoji, Katsumi Takahashi, and Nobuyuki Miura. Kokono search: A location based search engine. In *WWW Posters*, 2001.

[117] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[118] Yan-Tao Zheng, Ming Zhao, Yang Song, Hartwig Adam, Ulrich Buddemeier, Alessandro Bissacco, Fernando Brucher, Tat-Seng Chua, Hartmut Neven,

and Jay Yagnik. Tour the world: a technical demonstration of a web-scale landmark recognition engine. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 961–962, New York, NY, USA, 2009. ACM.