Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

USER-CENTRIC QUERY OPTIMIZATION OVER WEB DATA SERVICES

YU LI

Ph.D

The Hong Kong Polytechnic University
2015

The Hong Kong Polytechnic University

Department of Computing

User-Centric Query Optimization over Web Data Services

Yu Li

A thesis submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

April 2015

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Yu Li

April 2015

Abstract

Web-based data services have become more and more popular. Users from different fields are interested in different web-based data services. In this thesis, we consider three application scenarios with different queries and objectives. We propose effective methods to process and optimize users' queries over web data services.

In the first application, users are interested in datasets provided in Cloud Data Market (e.g., Windows Azure Data Market), which is an emerging cloud service that enables data owners to sell their datasets in a public cloud. Users (i.e., buyers) can access their interested data in data market via a RESTful API. Accessing data in the data market may not be free. We present PayLess, a system that helps users to process and optimize their SQL queries such that they pay less.

In the second application, mobile users of location-based services (LBS) issue range/K-NN queries over points-of-interest (e.g., restaurants, cafes), and they require accurate query results with up-to-date travel times. Lacking the monitoring infrastructure for road traffic, the LBS may obtain live travel times of routes from online route APIs (e.g., Google Directions API and Bing Maps API) in order to offer accurate results. Our goal is to reduce the number of external requests issued by the LBS significantly while preserving accurate query results.

In the third application, emerging spatial crowdsourcing web services enable

the users (i.e., crowdsourcing workers) to complete spatial crowdsourcing tasks (like taking photos, conducting citizen journalism) that are tagged with both time and location features. We study the problem of online recommending an optimal route for a crowdsourcing worker, such that he can (i) reach his destination on time and (ii) receive the maximum reward for tasks along the route. We show that no algorithms can achieve a non-zero competitive ratio in this problem. Therefore, we propose several heuristics, and powerful pruning rules to speed up the methods.

For each application scenario above, we evaluate the performance of our solutions on both synthetic data and real data. Our experimental results show that our solutions are effective and scalable.

Acknowledgements

I would like to thank all the people who give me tremendous support and help to make this thesis happen.

First of all, I would like to express my deepest sense of gratitude to my supervisor, Dr. Man Lung Yiu, for his generous time and devotion on supervision and guidance in the past four years. His passion on research and life, his patience in guiding students, and his diligence set up a great example for me. I always feel very lucky to have Dr. Man Lung Yiu as my supervisor, for all strong supports and earnest encouragements.

Next, I would like to thank Dr. Eric Lo for giving me so many good advices. His bright ideas, his keen insights and vision, his way to thinking and problem solving always inspire me.

I also sincerely thank all my research colleagues and friends during my Ph.D. study. Many thanks to Xiao Li, Zhian He, Duncan Yung, Jianguo Wang, Petrie Wong, Bo Tong, Capital Li, Qiang Zhang, Ziqiang Feng, Chuanfei Xu, Jeppe Thomsen, Ran Bai, Zhizhao Feng, Victor Liang, Edison Chan. for their tremendous help and support. Xiao Li gives me a lot of positive energy and great

encouragements when I feel disappointed and confused.

Finally and above all, I am deeply grateful to my parents and my husband. I want to thank my parents for their endless and unreserved love. They are my constant source of support and power. I want to thank my husband, Wenjian, who has been together with me for about eight years. Without him, I would not come to the academic research world, and I would not become the one who I am today. His love, patience, support and endless care enable me to carry through my Ph.D. study to the end.

Contents

Declaration **i**

Abstract **iii**

Acknowledgements **v**

Contents **vii**

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**

2 Literature Review **9**

2.1 Query Optimization over Remote Data Sources 9

2.1.1 Query Optimization with Limited Access Patterns 10

2.1.2 Query Optimization over Multiple Web Services 11
2.1.3 Distributed Query Optimization 12
2.2 Semantic Caching 12
2.3 Location-dependent Queries 14

3 Query Optimization over Cloud Data Market 17

3.1 Preliminaries 23
3.1.1 Data Market 23
3.1.2 Queries over Data Market 24
3.2 System Overview 26
3.3 Query Optimization 28
3.3.1 Plan Space 28
3.3.2 Semantic Query Rewriting 36
3.3.3 Discussion 48
3.4 Experimental Evaluation 49
3.5 Related Work 60
3.6 Chapter Summary 61

4 Query Processing using Route APIs 63

4.1 Related Work 68

4.1.1 Query Processing on Road Networks 68
4.1.2 Querying on Online Route APIs 69
4.2 Problem Statement 71
4.3 Query Processing 75
4.3.1 Maintenance of Structures at LBS 75
4.3.2 Exact Travel Times and Their Bounds 77
4.3.3 Range Query Algorithm 84
4.3.4 KNN Query Algorithm 89
4.3.5 Applicability of Techniques without Map 91
4.4 Parallelized Route Requests 92
4.5 Experimental Evaluation 96
4.5.1 Experimental Setting 97
4.5.2 Accuracy on Real Traffic Data 98
4.5.3 Performance and Scalability Study 100
4.5.4 Experiments on Google Directions API 110
4.6 Chapter Summary 111

5 Route Recommendation for Spatial Crowdsourcing Workers 113

5.1 Problem Statement 116
5.2 Online Route Recommendation 118

5.2.1 Competitive Analysis 118
5.2.2 Greedy Task Approach 120
5.2.3 Complete Search for Route Approach 123
5.3 Optimization for SnapshotRR 125
5.3.1 Forward Search and Backward Search 126
5.3.2 Route Join 131
5.4 Discussion 133
5.5 Experiment 135
5.5.1 Experimental Setting 135
5.5.2 An Experiment on Real Datasets 137
5.5.3 Scalability Experiments on Synthetic Datasets 138
5.6 Related Work 141
5.7 Chapter Summary 143

6 Conclusion and Future Work 145

6.1 Contributions 145
6.2 Future Research Directions 146

Bibliography 147

List of Figures

1.1 Position of our work 2

1.2 Example dataset WHW in Microsoft Azure Data Marketplace . . 4

1.3 Example of route recommendation for workers: each task p_i with [release time - deadline] 7

2.1 Example of using semantic caching 13

2.2 Example of range and KNN queries 16

3.1 Query processing in Data Market 21

3.2 Setting of PayLess 25

3.3 System architecture 27

3.4 Bushy tree v.s. Left-deep tree 29

3.5 Illustration figures for Theorem 1 32

3.6 Generation of remainder queries 37

3.7 Generation of remainder queries for data market 40

3.8 Generation of remainder queries with a categorical attribute A_2 . 44

3.9 Example for 2D-Bind 46

3.10 Overall effectiveness 51

3.11 Varying the number of results t per transaction 55

3.12 Varying the number of query instances (q) per template 56

3.13 Varying data size 56

3.14 Effectiveness of search space reduction techniques 58

3.15 Effectiveness of bounding box pruning rules 59

4.1 A restaurant rating website: data and queries 64

4.2 Measurement of live travel times 65

4.3 System architecture 73

4.4 Example route log $\mathcal{L}$ and time-tagged road network G, with expiry time $\delta = 2$; solid edges have valid travel times 78

4.5 Data stored at the LBS, at $t_{now} = 9$ ($\delta = 5$); each edge is tagged with $(c\omega^{-}(e), \underline{\omega(e)})$ 79

4.6 Effect of parallelization on schedules 94

4.7 Result accuracy on real traffic data [in color] 100

4.8 Time of route requests on roadmaps 101

4.9 Temporal behavior, expiry time $\delta = 10$ 103

4.10 Effectiveness of techniques 104

4.11 Effect of expiry-time δ 106

4.12 Effect of various parameters [y-axis: route requests] 107

4.13 Effect of the number of threads m 110

4.14 Temporal behavior vs. timeline, $\delta = 10$ minutes, Manhattan region (in New York), on Google Directions API 111

5.1 Route recommendation for the worker: each task p_i with [release time - deadline] 114

5.2 System architecture 118

5.3 At time $m = 3$, adversaries release tasks $p_{2 \leq i \leq n+1}$ with [release time - deadline] 120

5.4 Example of query $q = (s, d, [0, 10])$ in OnlineRR (using Manhattan Distance) 122

5.5 Feasible candidates search space for Euclidean distance metric . . 123

5.6 Example query $q = (s, d, [0, 10])$ for SnapshotRR problem (using Manhattan distance) 128

5.7 Performance on real datasets 138

5.8 Effect of task distribution 139

5.9 Effect of the total number of tasks 140

5.10 Effect of the query period $t_q^+ - t_q^-$ 140

5.11 Effect of pruning rules on Re-Route (E for ENABLE, D for DISABLE) 141

List of Tables

1.1 Differences of query optimizers for different queries 3

1.2 Example Google Directions API 5

3.1 Query result accuracy options 49

3.2 Query templates on real DataSets 50

4.1 Example Google Directions API 69

4.2 Example travel time information (for user q), $t_{now} = 9$ 81

4.3 Range/KNN query example for Route-Saver, $T = 40$ 87

4.4 Experiment parameters 98

4.5 Effect of candidate ordering 104

4.6 Effect of roadmap on range and KNN queries 108

4.7 Effect of query distribution 109

5.1 Forward space search 130

5.2 Route join 132

5.3 Experiment parameters 137

5.4 Reward on the optimal route 139

Chapter 1

Introduction

With the advent of web, web data services become more and more popular. Web data services enable mashup, reuse, and sharing data from sources in different categories. Such data include structured data (such as relational tables), semi-structured information (such as XML documents) and unstructured information (such as content from web applications). The consumers of web data services can be location-based services, enterprises, research organizations, social media, social networks and so on. To access web data services, the consumers need to follow the API interfaces provided by these data services. For example:

- Google Directions API [16] requires the consumers to issue HTTP requests to retrieve live traffic information.

- Microsoft Azure Marketplace [4] requires the consumers to follow the limited access patterns defined in the marketplace to access data.

Consumers use information from web data services for different purposes. For instance:

- Companies or organization can use data from Microsoft Azure Data Marketplace [4] to conduct business or research analytics.
- Location based services can use live traffic information from Google Directions API [16] to solve spatial queries accurately.
- Crowdsourcing workers can use tasks published in Amazon Mechanical Turk [3] to make money through completing those tasks.

Queries issued by consumers which involve data from web data services are called queries over web data services.

Queries in various fields can have various objectives. From a user's perspective, obtaining an accurate/optimal result for a query in a short response time is always preferred. Thus, as a hot topic in database field, query optimization is of great importance for query processing over web data services.

As shown in Figure 1.1, our query optimizer is located in the client side, delegating our user to interact with web data services. In this thesis, we investi-

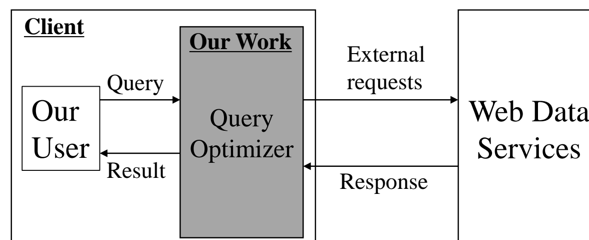


Figure 1.1. Position of our work

gate three different types of queries over web data services: (i) SQL queries over

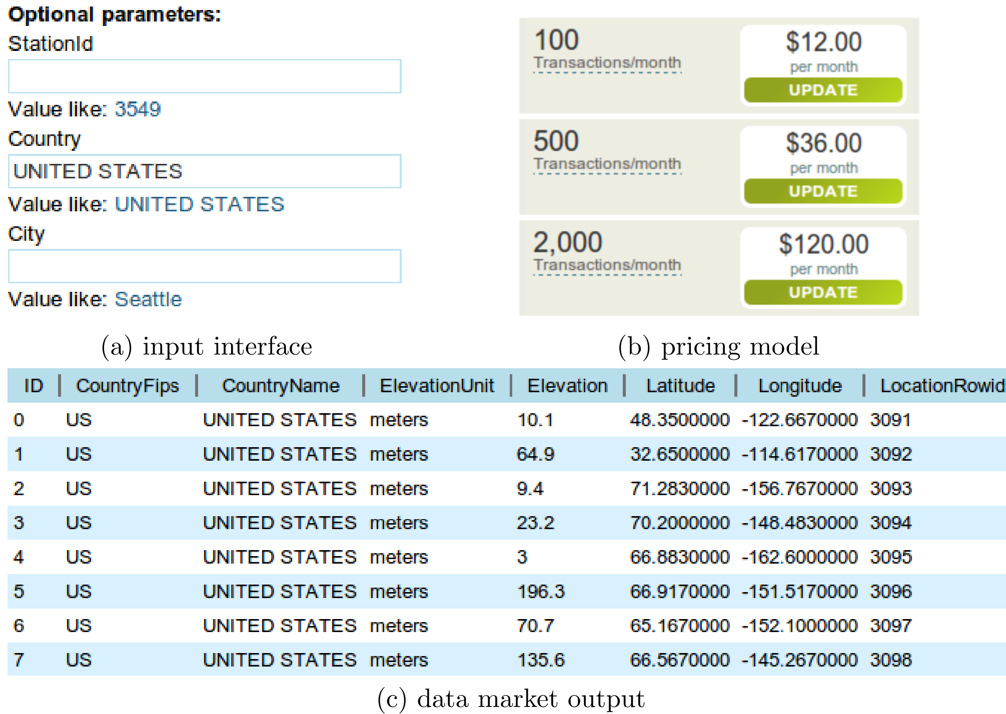
Table 1.1. Differences of query optimizers for different queries

Query	Our User	Data Services	Optimization Goal
SQL queries	data buyers (e.g., companies, organizations)	cloud data market	paying less to data sellers
Spatial range and KNN queries	location-based services	online route APIs	issuing fewer requests to online route APIs
Route recommendation queries	spatial crowdsourcing workers	spatial crowdsourcing servers	earning more rewards by conducting crowdsourcing tasks

cloud data markets, (ii) spatial queries (e.g., spatial range and K-NN queries) over online route APIs, and (iii) route recommendation queries for crowdsourcing workers over crowdsourcing task marketplaces. We summarize the differences among the query optimizers for these three kinds of queries in Table 1.1.

SQL queries over cloud data market. Cloud data market is an emerging type of cloud services that enables a data owner to host and sell datasets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. Figure 1.2 shows an example of interfaces for the GetStation table in Worldwide Historical Weather (WHW) dataset [26] in Windows Azure Marketplace. Given a range or a value for input attributes in Figure 1.2a, the data market returns values for the output attributes in Figure 1.2c. For instance, the Azure Marketplace may take a country name as an input, and return a set of tuples, each details the elevation, latitude, longitude of each weather station in that country. Accessing data in the data market may not be free. As shown in Figure 1.2b, it costs USD $12 to grant access to every 100 "transactions" to the WHW data, where a transaction is a unit of result size which refers to at most 100 records. There is an increasing trend of selling

valuable datasets in data market. Correspondingly, we envision that there is an increasing demand from end users (data buyers) to carry out analytics that involve those datasets. Thus, we study how to optimize the queries for data buyers such that they can get query results by paying less to data sellers.



(a) input interface (b) pricing model

ID	CountryFips	CountryName	ElevationUnit	Elevation	Latitude	Longitude	LocationRowid
0	US	UNITED STATES	meters	10.1	48.3500000	-122.6670000	3091
1	US	UNITED STATES	meters	64.9	32.6500000	-114.6170000	3092
2	US	UNITED STATES	meters	9.4	71.2830000	-156.7670000	3093
3	US	UNITED STATES	meters	23.2	70.2000000	-148.4830000	3094
4	US	UNITED STATES	meters	3	66.8830000	-162.6000000	3095
5	US	UNITED STATES	meters	196.3	66.9170000	-151.5170000	3096
6	US	UNITED STATES	meters	70.7	65.1670000	-152.1000000	3097
7	US	UNITED STATES	meters	135.6	66.5670000	-145.2670000	3098

(c) data market output

Figure 1.2. Example dataset WHW in Microsoft Azure Data Marketplace

Spatial queries over online route APIs. Location-based services (LBS) enable mobile users to issue spatial queries (i.e., range or K-NN queries) over points-of-interest (e.g., restaurants, cafes) on various features (e.g., price, quality, variety, distance to travel). Users require accurate query results with up-to-date travel times within short response time. Lacking the monitoring infrastructure for road traffic, the LBS may obtain live travel times of routes from online route APIs in order to offer accurate results. Take Google Directions API [16] for

instance, as shown in Table 1.2, the request is an HTTP query string, whose parameters contain the origin and destination locations in latitude, longitude, as well as the travel mode. The response is an XML/JSON document that stores a sequence of route segments from the origin to the destination. Using online route APIs raises challenges for the LBS in meeting the response time requirement of the users because a route request to route APIs incurs considerable time (0.1s-0.3s) which is high compared to CPU time in LBS. Also, in order to solve end-users' queries, LBS may need to pay for their issued requests to the online route APIs. Thus, it is important to optimize the end-users' queries in the LBS such that LBS incurs fewer requests to online route APIs.

Table 1.2. Example Google Directions API

HTTP request

```
http://maps.googleapis.com/maps/api/directions/xml?
origin=44.94033,-93.22294&destination=44.94198,-93.23722
mode=driving
```

XML response

```
<step>
   <start_location>
      <lat>44.9403300</lat> <lng>-93.2229400</lng>
   </start_location>
   <end_location>
      <lat>44.9395900</lat> <lng>-93.2229500</lng>
   </end_location>
   <duration> <value>8</value> </duration>
</step>
      ...... remaining steps ......
```

Route recommendation queries over spatial crowdsourcing tasks. Spatial crowdsourcing platforms may publicly publish crowdsourcing tasks that are associated with rewards and tagged with spatial / temporal attributes (e.g., location, release time and deadline). To complete a task, a worker must reach the location of the task before its deadline. Popular tasks include taking photos, reporting activities/accidents, and verifying data on-site, etc. The spatial crowdsourcing approach in worker-centric mode [55] enables the workers to choose tasks

autonomously and protects their location privacy. [55] returns a route that covers the maximum number of tasks (in a worker's specified region, e.g., his city). Compared to [55], two extra requirements should be supported: ($\mathcal{R}1$) update the worker's route online with respect to newly released tasks and ($\mathcal{R}2$) align with the worker's trip, i.e., reaching a destination before deadline. It is important to support requirement $\mathcal{R}1$ in order to assign a worker as many tasks as possible. New spatial crowdsourcing tasks are indeed being released continuously in real systems [1.1]. Requirement $\mathcal{R}2$ is also important as the worker may have planned his own activities, e.g., reaching a specified destination by an expected time [90]. Such worker is willing to take crowdsourcing tasks along his trip provided that he can arrive at his destination on time. As a result, such workers may issue queries to retrieve an optimal route such that he can (i) reach his destination on time and (ii) receive the maximum reward for tasks along the route. For example, as shown in Figure 1.3, the worker starts from s at time 0 and plans to arrive at home $(5, 0)$ at time 8. At time 0, tasks p_1, p_2 are known, and the worker is recommended to take the task p_2. When new tasks are released (e.g., p_3, p_4), the worker is recommended to take them. In the end, the complete recommended route is $s \to p_2 \to p_3 \to p_4 \to d$, which covers 3 tasks and reaches the destination d on time. Therefore, we study the online route recommendation problem for spatial crowdsourcing workers, by taking requirements $\mathcal{R}1$ and $\mathcal{R}2$ into consideration.

The main contributions of this thesis can be summarized as follows:

– We design a system to help data buyers solve SQL queries over cloud data

[1.1] www.clickworker.com/en/clickworkerjob
www.lionbridge.com

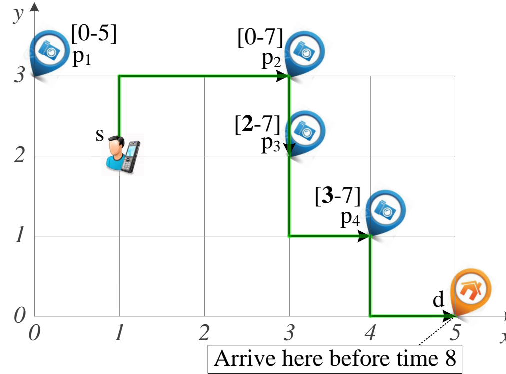


Figure 1.3. Example of route recommendation for workers: each task p_i with [release time - deadline]

markets by paying less to data sellers.

- We develop reusing algorithms and pruning rules to support efficient and accurate spatial range and KNN search on LBS using online route APIs.
- We present algorithms for recommending routes for spatial crowdsourcing workers.

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the existing work related to query optimization over web data sources.

Chapter 3 (based on [91]) studies how to optimize SQL queries which involve datasets in data markets. A system PayLess is built to optimize a SQL query, which takes into consideration the limited access pattern in the data market and its unique pricing model. We design a bottom-up cost-based dynamic programming approach to find the optimal execution plan for a SQL query. Besides, we adapt a semantic query rewriting approach [53] to reuse the records retrieved from data market for previous queries to save money further. We conduct ex-

tensive performance evaluations on both real data (i.e., datasets from Microsoft Azure Data Marketplace) and synthetic data (i.e., TPC-H dataset) to evaluate our PayLess system.

Chapter 4 (based on [92]) studies how to help LBS calculate accurate answers for end-users' spatial queries within a short response time. For the requirement of providing accurate answers, we use live traffic information retrieved from online route APIs to help LBS solve spatial queries. For the efficiency requirement, we reuse the retrieved route information for previous queries to reduce the number of requests issued to online route APIs. We derive tight lower/upper shortest path distance bounds between the end-user and points-of-interest (POIs), which can be used to prune un-promising POIs to improve the efficiency. Moreover, we design parallel algorithm to reduce the response time on the LBS further. We conduct experiments on both real and synthetic data.

Chapter 5 (based on [93]) studies how to recommend routes for spatial crowdsourcing workers, such that the worker can arrive his destination on time as well as earn most reward by completing spatial crwodsourcing tasks on the route. The route recommendation problem is an online problem as tasks are continuously published by the crowdsourcing servers. We first prove that, there is no optimal online algorithm for this problem, thus, we propose two categories of heuristics to solve this problem. Then we utilize the spatial property of this problem to design powerful pruning rules to reduce the route searching space to speed up the algorithm. Extensive experiments over real and synthetic datasets are conducted.

Chapter 6 concludes the thesis and discusses some future research directions.

Chapter 2

Literature Review

In this chapter, we present an overview of query optimization techniques over remote data sources, semantic caching over web queries, and query processing of location dependent queries.

2.1 Query Optimization over Remote Data Sources

Remote data sources (e.g., web services) have recently gained tremendous momentum for sharing data and functionality among loosely-coupled, heterogeneous systems. Companies/organizations prefer to hide their databases by providing an interoperable, function-call like interface for interacting with their data [117]. Given a query over remote data sources, query optimization techniques aim at finding an optimal execution plan with respect to different cost models. In this section, we review those optimization techniques in the literature.

2.1.1 Query Optimization with Limited Access Patterns

The access pattern of remote data sources is usually restricted to $\mathcal{X} \rightarrow \mathcal{Y}$ style: by specifying values of attributes in $\mathcal{X}$, values of attributes in $\mathcal{Y}$ are retrieved. This requirement is commonly referred to as *access limitations (or binding patterns)* [43]. In such contexts, traditional query optimization techniques may not work; instead, the target query plan should incur the lowest cost and comply with the access limitations at the same time. To evaluate a Select-Project-Join (SPJ) query, the so-called *maximal answer* (i.e., the set of answer tuples can be disclosed for the query) needs to be retrieved while minimizing accesses to remote sources. Next, we elaborate query optimization techniques applicable at query definition time (i.e., static optimization) and at run-time, respectively.

To begin with, determining which sources are *relevant* to a given query is of great importance, as it saves query execution time by excluding irrelevant sources. Existing solutions [42, 88] of determining relevance sources at query definition time are limited to SPJ queries over relations with exactly one access pattern. After ruling out irrelevant data sources from a query plan, Calì and Martinenghi [42] further proposed techniques to avoid accesses (to relevant relations) that are unnecessary for obtaining the maximal answer.

By taking full advantage of intermediate data extracted from the remote sources, query plans can further be optimized at run-time. Particularly, when remote databases enforce certain integrity constraints, some accesses to sources planned statically may turn out to be useless for the computation of new answer tuples. In such cases, we can avoid those accesses at run-time [41, 56, 98, 99].

For instance, Calì et al. [41] proposed a dynamic optimization for relations with integrity constraints including functional dependencies and inclusion dependencies.

2.1.2 Query Optimization over Multiple Web Services

The ability of efficiently evaluate queries over multiple Web services is necessary, in light of the expensive communication cost over the network. Srivastava [117] proposed to optimize queries over *a collection of* Web services with a Web Service Management System (WSMS). In their system, each Web service $WS(\mathcal{X}, \mathcal{Y})$ is modeled as a virtual table with $\mathcal{X} \rightarrow \mathcal{Y}$ style data access pattern. The optimization algorithms in the system take the Select-Project-Join (SPJ) queries as input, and return a pipelined execution plan over multiple Web services with minimum total running time. Qi and Athman [130] took a further step to take both the response time and the quality of Web services into consideration in the query optimization. Braga et al. [38] proposed the so-called multi-domain queries that can be answered by combining knowledge from multiple domains, e.g., *Where can I attend an interesting database workshop close to a sunny beach?* To evaluate such queries, they designed a branch-and-bound search strategy to find the best query plan with regard to a scheduling of (possibly parallel) service invocation. As the cost for data transfer is the main bottleneck for the evaluation of queries involving multiple web services, Anastasios et al. [67] proposed a robust control solution to tune the block size for data transfer at run-time. Finally, we remark that the concept of utilizing the results from one web service to query another is essentially the same as the *Dependent Join* [47, 95].

2.1.3 Distributed Query Optimization

In above setting of query processing over remote data sources, only *data shipping* is allowed, that is, data is transferred between client-side and the remote sources, and remote sources process the data according to their pre-defined functionality. In contrast, traditional distributed query processing and optimization has been addressed extensively in the literature [78,100], where *code shipping* is also allowed apart from data shipping, that is, assigning each machine to execute portions of code over portions of data. To reduce the communication cost in this distributed query processing environment, a multitude of techniques are proposed in previous work [78], including semi-join, fragment-replicate join, and double-pipelined hash join.

2.2 Semantic Caching

Semantic caching is a client-side caching technique in client-server systems. The data cached at the client side contains both semantic descriptions and results of previous queries.

When a new query comes, we check whether the cached results of a previously computed query can be used for a new query, or whether the client needs to request additional data from the server. A cache located at a client can only serve queries from the client itself, not from other clients. Thus, with semantic cache in client-side, less cost will be spent to communicate with servers.

With semantic caching [53, 105], a new query will be split into two disjoint pieces: (i) a probe query that extracts the relevant portion of the query result

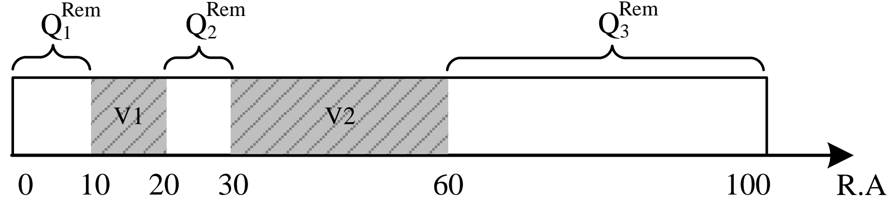


Figure 2.1. Example of using semantic caching

in the local client cache, and (2) a remainder query that accesses the server to retrieve any missing tuples in the query result. To illustrate, consider the example in Figure 2.1. The example assumes that the results of two queries V_1 and V_2 on table R have been stored in the semantic cache. Both V_1 and V_2 are range queries on an integer attribute A whose domain is $[0, 100]$. V_1 and V_2 respectively cover the ranges $[10, 20)$ and $[30, 60)$ on attribute A. Now, with V_1 and V_2, we assume the following query Q is posed to retrieve from R all tuples with $0 \leq A \leq 100$:

$$Q: \quad R(A[0, 100])$$

Then the remainder query $Q^{Rem} = Q_1^{Rem} \vee Q_2^{Rem} \vee Q_3^{Rem}$ is generated as:

$$Q^{Rem}: \quad R(A[0, 10) \vee [20, 30) \vee [60, 100])$$

Semantic caching has been studied for web queries [32,33,50,84]. Chidlovskii et al. [84] introduce a semantic caching scheme for conjunctive keyword-based web queries. The results of previous queries are stored as semantic regions [53] in the cache and reused to reduce the response time and network traffic of a new query. Here, to quickly process a comparison of an input query against the semantic views, binary signature method is used. DBProxy [32] is a semantic data cache at the edge server. DBProxy decides dynamically to add new views and

discards others on the fly to save space and execution time. Consistency maintenance of the semantic cache is considered in DBProxy. Results of SQL queries are stored as materialized views in the cache, independent from the original database schema. The list of queries whose results are currently stored is kept in a cache index. The system uses template based query containment checking algorithms to answer a new query using the cached results of previous queries [33].

Semantic caching has also been studied for mobile services [70, 85, 133]. In [133], the Voronoi cell of a spatial object is defined as its semantic region, which can be used to solve spatial NN queries for a moving user. Semantic caching of tree nodes in an R-tree is studied in [70]. Lee et al. [85] support generic spatial queries by building generic semantic regions for spatial objects.

2.3 Location-dependent Queries

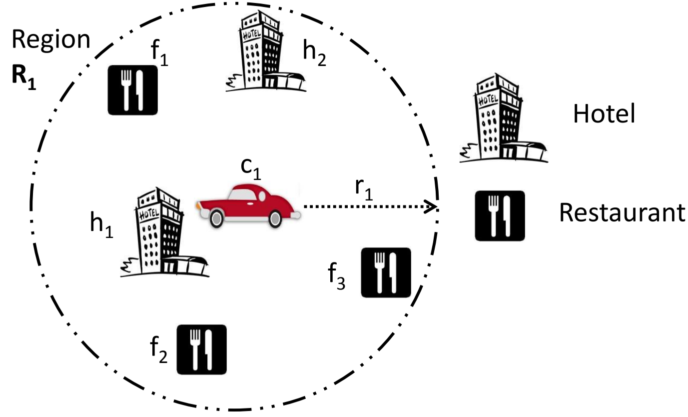
Location-dependent query [71] is a query whose answer depends on the locations of the query and points-of-interest (POIs). The location of a POI determines whether or not it can be chosen as part of the answer. According to their objectives, the location-dependent queries can be divided into different types.

Range queries. Range queries find the POIs located within a certain range [101, 124]. According to whether the query's location is fixed or moving, range queries can be divided into static range or moving range queries. When the range of query is a rectangle window [120], the range queries are called as window queries; when the range is a circular range centered on a certain location, the range queries are called as within-distance queries [123]. An example of range query is illustrated

in Figure 2.2. A car c_1 issues a range query to retrieve the restaurants inside the region R_1, which is a circle centered at c_1 with radius r_1. According to Figure 2.2a, three restaurants $\{f_1, f_2, f_3\}$ fall in the region R_1. Thus, the result is $\{f_1, f_2, f_3\}$.

Nearest neighbor (NN) queries. Nearest neighbor query [119] is to retrieve the POI with certain features, which is the closest to the query location in terms of the spatial distance. When k POIs must be returned, the problem becomes KNN queries. Static NN queries are executed against a static database whereas in dynamic NN queries, the target location can move. [101] studies how to process range queries and KNN queries over POIs, with respect to shortest path distances on a road network. To speed up the KNN query processing over road network, landmark [82, 83, 103] and distance oracle [109] can be applied to estimate shortest path distance bounds between two locations, which can be used to prune irrelevant objects and early detect results. For example, a car c_1 in Figure 2.2a searches for its $k = 1$ nearest hotels. All k nearest hotels $\{h_1\}$ will be returned as the query result (i.e., last line in Figure 2.2b).

Navigation queries. Navigation query is to recommend the best route for a mobile client to arrive at his destination, based on the underlying road network, current traffic conditions, and his specific purposes [115]. Such navigation query can be issued by tourists for guidance to travel in a region [127]. Based on tourists' interest profile, up-to-date POI information and trip information, an optimal and feasible selection of POIs and a route between them can be suggested. Traveling salesman problem (TSP) [114] is another example of navigation queries. Selective traveling salesman problem (STSP) [60] is a variant of TSP



(a) Example map and regions

Query type	Query parameters			Answer
	Query point	Region / K	POI Type	
Range	Car c_1	Region R_1 (radius r_1)	Restaurant	$\{f_1, f_2, f_3\}$
KNN	Car c_1	$k = 1$	Hotel	$\{h_1\}$

(b) range and KNN query results

Figure 2.2. Example of range and KNN queries

problem, where the user has flexibility to reject part of the POIs on the map. When all POIs are known in advance, the navigation problem is a static trip planning problem [60, 66, 90, 113, 114, 126]. On the other hand, when the POIs are known progressively, the navigation problem becomes online path selection problems [34, 37, 72, 128].

Chapter 3

Query Optimization over Cloud Data Market

Data market [4, 35, 110] is an emerging type of cloud service that enables a data owner to host and sell their datasets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. The REST based API has function-call like interface $\mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes: given a range or a value for an attribute in $\mathcal{X}$, the data market returns values for the attributes in $\mathcal{Y}$ (if no values are specified for $\mathcal{X}$, the whole table is returned). For example, the Worldwide Historical Weather (WHW) dataset [26] in Windows Azure Marketplace [4] may take a country name and a date, and return a set of tuples, each details the temperature, precipitation, dew point, sea level pressure, windspeed, and wind gust recorded by each weather station in that country on that date.

Accessing data in the data market may not be free. For example, it costs

USD 12 to grant access to every 100 “transactions” to the WHW data, where a transaction is a unit of result size (e.g., a query result of 4400 records costs 44 transactions in Windows Azure Marketplace, which confines one transaction to 100 records). There is an increasing trend of selling valuable datasets in data market [80]. Correspondingly, we envision that there is an increasing demand from end users (data buyers) to carry out analytics that involve those datasets. To this end, in this chapter, we present PayLess, a system that helps users to optimize their queries so that they can obtain the query results by *paying less* to the data sellers.

Query optimization is never trivial. First, from a data buyer’s (the company or the organization) perspective, it is hard to know in advance how many queries will be posed by their end users eventually. Otherwise, downloading the whole dataset would become a viable plan when the foreknowledge tells that the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set. Second, query optimization would never work well without rich data statistics. Unfortunately, datasets in data market are rarely tagged with rich statistics (e.g., no value distribution), although basic information like the size (cardinality) of each table and the domain size of the attribute is usually available.

Tackling the above two challenges sounds not difficult, especially that we can build a *learning* optimizer like LEO [118] so that it begins with little statistic and introduces a feedback loop to correct the statistics when more queries are issued. The evil, however, lies in the detail of adopting the learning approach to data market query optimization.

First, learning-based optimizers like LEO [118] and POP [96] are originally designed for traditional databases that have full access to the data. In contrast, the access pattern of data market is restricted to only $\mathcal{X} \rightarrow \mathcal{Y}$ style. When a data source has limited access patterns, (a) operations might become complicated and (b) specialized access paths may shine. An example of (a) is that a query that asks `Country = 'Canada' OR Country = 'Germany'` has to decompose into two queries, one asks for `Country = 'Canada'` and another asks for `Country = 'Germany'`. An example of (b) is *bind joins* (other names include theta semi-join, dependent join) [63]. To explain, consider the real access patterns of Worldwide Historical Weather (WHW) dataset in Windows Azure Marketplace listed in Figure 3.1a.[3.1] The access patterns are specified using a notation of binding patterns extended from [63]. We write $R^{\alpha}(A_1, A_2, A_3)$ to denote a table R in the data market with three attributes A_1, A_2, and A_3 and *binding pattern* α. We write $\alpha = R(A_1^b, A_2^f)$ to denote a binding pattern that in any query accessing R, the value of attribute A_1 must be *bound* (given/specified). In contrast, the value of attribute A_2 is *free* to be specified or not specified in any query. If an attribute is not included in the binding pattern (e.g., A_3), it is solely served as an output attribute in a query result. In other words, if an access pattern of a table has only free attributes, then we can download the whole table by not specifying any value to any attribute.

Now, consider the following SQL query that asks the WHW dataset for the daily temperature of Seattle in June 2014:

[3.1] The attribute names here are renamed for better exposition.

```
SELECT Temperature        -------// Query Q1
FROM Station, Weather
WHERE City = 'Seattle' AND
      Country = 'United States' AND
      Date >= 20140601 AND Date <= 20140630 AND
      Station.StationID = Weather.StationID
```

Figure 3.1b shows an execution plan P_1 for this SQL. It first submits two RESTful GET calls C_1 and C_2, where C_1 gets the StationID of Seattle from Station table, and C_2 gets the weather records for all stations in the United States on June 2014 from Weather table. The final query result is obtained by carrying out a *local join* (i.e., regular join) operation at the end user (data buyer) side because joins cannot be done at the data market [4]. In plan P_1, a total of 238 transactions were incurred – one was spent on RESTful call C_1 and 237 were spent on RESTful call C_2 (there are 788 weather stations in the US and each station contributes 30 days records, resulting in $\lceil 788 \times 30/100 \rceil = 237$ transactions). Figure 3.1c shows an alternate execution plan P_2. It first gets the list of StationIDs of Seattle (call C_1). Then, it carries out a bind join ($\overrightarrow{\bowtie}$) operation that binds each StationID (e.g., 3817) to an individual RESTful call to Weather. Finally, the weather records for each station in Seattle are collectively retrieved and returned. In this case, plan P_2 incurs only two transactions: call C_1 costs one transaction and call C_3, which returns 30 days of weather records for the only one weather station in Seattle, costs also one transaction.

Second, although there are optimizers designed for queries over remote data sources with limited access patterns (e.g., [42, 59, 63, 86–89, 104, 117]), they focus on minimizing the number of calls to the remote data sources so as to reduce the

Data Set	Schema and Access Pattern α	Size
WHW	Station$^{\alpha_1}$(Country, StationID, City, State$\cdots$) α_1=Station(Country^f, StationID^f, City^f)	3962
	Weather$^{\alpha_2}$(Country, StationID, Date, Temperature$\cdots$) α_2=Weather(Country^f, StationID^f, Date^f)	19549140
EHR	Pollution$^{\alpha_3}$(ZipCode, Rank, Latitude, Longitude$\cdots$) α_3=Pollution(ZipCode^f, Rank^f)	44210
local	ZipMap (ZipCode, City)	

(a)

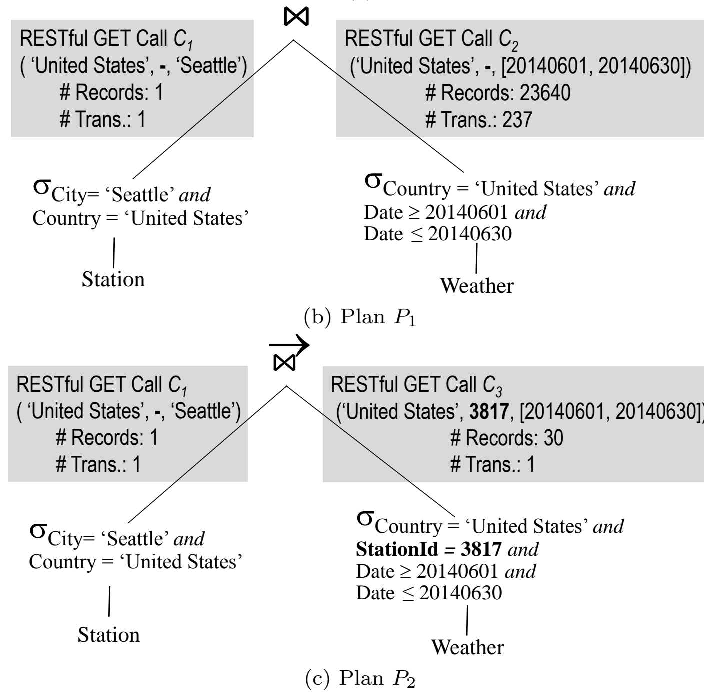


Figure 3.1. Query processing in Data Market

overall execution time. As an example, assume that there are 15 weather stations in Seattle, those optimizers will pick plan P_1 because it incurs only two RESTful calls (C_1 and C_2). In data market, although P_2 needs to bind each Seattle's weather station id, resulting in $1 + 15 = 16$ RESTful calls and 16 transactions (each transaction returns 30 days of records for each weather station), it is still more economical than P_1, which requires 238 transactions. On the other hand, if we further assume that there are only 20 weather stations in the United States

and 15 of them are in Seattle. Then, plan P_1 will cost only $1 + \lceil 20 \times 30/100 \rceil = 7$ transactions. In contrast, plan P_2 still costs 16 transactions. In this case, P_1 is better than P_2.

Summing up the above, we need a (i) learning-based optimizer that (ii) includes bind join as an access path with the goal of (iii) minimizing the amount of (intermediate) retrieved data measured in terms of data market pricing units. Traditional learning-based optimizers satisfy (i) and partially satisfy (iii)[3.2] but not (ii). Optimizers for queries over remote data sources satisfy only (ii). Therefore, the principal contributions of this chapter are centered around the issues of building an optimizer for PayLess that satisfies all (i), (ii), and (iii) above. Those include:

– Defining the cost model and search space for data market query optimization.

– Devising effective techniques to reduce the amount of intermediate retrieved data (e.g., by adapting semantic query rewriting methods) and integrating those techniques into our optimizer.

– Implementing a prototype and evaluating its performance through extensive experiments over synthetic data and real data.

The remainder of this chapter is organized as follows. Section 3.1 gives more background about the data market. Section 3.2 presents the architecture of PayLess. Section 3.3 describes the details of PayLess's optimizer. Section 3.4

[3.2] Traditional optimizers also aim to generate plans that minimize intermediate result size of each operation (e.g., push down selection).

reports the results of the evaluation. Section 3.5 discusses the related work and Section 3.6 concludes.

3.1 Preliminaries

According to a recent survey [8], the three most established data marketplaces are Factual [14], Microsoft Windows Azure Data Marketplace [4], and DataMarket [10]. Factual [14] and DataMarket [10] are specialized data markets that sell datasets in a very specific domain (e.g., Factual sells mainly geographical data and DataMarket sells mainly economic indicators). Microsoft Windows Azure Data Marketplace offers datasets in all kinds and many popular data resellers in smaller size like Wolfram Alpha [24], ESRI [13], World Bank [25], data.gov [9], Xignite [27] also provide their data in the Windows Azure Data Marketplace [4]. After Infochimps [19], one of the early data market entrants, gradually leaves the data market business [11, 20], Microsoft Windows Azure Data Marketplace is becoming the de facto data market [8]. Therefore, in this chapter, we base our setting on Windows Azure Data Marketplace.

3.1.1 Data Market

A data market hosts and sells multiple datasets. Each dataset's access/binding pattern is defined by the data owner on per table basis. For numeric attributes, the input can be bound with a single value or a range like $[150, 200)$. Datasets in data market are tagged with very basic statistics, normally the do-

main of each attribute and the number of records (cardinality).[3.3] Datasets in a data market are *append-only* because they are released for analytic purposes. New data could be added periodically (e.g., every month). The price of accessing data is mainly based on the number of tuples retrieved. A *transaction* represents a page of t tuples (e.g., 100 tuples) and it is the smallest pricing unit. Let p be the price per transaction for a particular dataset. Then, the total price of a RESTful call is:

$$p \cdot \lceil \frac{\text{number of resulting records}}{\text{number of tuples per transaction } (t)} \rceil \tag{3.1}$$

For easy exposition, in the subsequent discussion, we assume $p = \$1$ and a transaction page size is $t = 100$ tuples.

3.1.2 Queries over Data Market

Figure 3.2 shows the target setting of PayLess. An organization is interested in carrying out certain analytics that involve datasets hosted in a data market. The organization thus registers with the data market to obtain the authentication access keys of the datasets. The access keys are stored in PayLess, which constructs RESTful calls to the data market when necessary. PayLess encapsulates the details of interacting with the data market and exposes a SQL query interface for client query processing. A SQL query to PayLess can query against both tables in a local DBMS and tables in data market. The following is an example PayLess query that aims to retrieve the average temperature for each

[3.3] If not publicly available, the data sellers would release the basic statistic to data buyers upon email requests [4].

city in a country whose environmental pollution rank is lower than a threshold within a period:

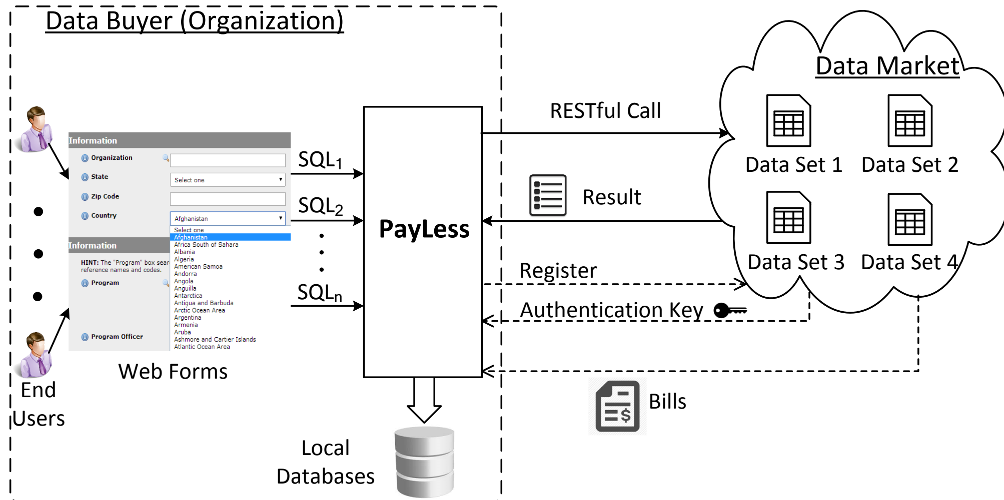


Figure 3.2. Setting of PayLess

```
SELECT City, AVG(Temperature)
FROM Pollution, Station, Weather, ZipMap
WHERE Station.Country = Weather.Country = ? AND
      Weather.Date >= ? AND Weather.Date <= ? AND
      Pollution.Rank <= ? AND
      Pollution.ZipCode = ZipMap.ZipCode AND
      ZipMap.City = Station.City AND
      Station.StationID = Weather.StationID
GROUP BY City
```

This query involves joining four tables: the Station and Weather tables from the aforementioned Worldwide Historical Weather (WHW) [26] dataset, another Data Market table, the Pollution table from the Environmental Hazard Ranking (EHR) [12] dataset, and a local table that maps Zip codes to a city name. The

access patterns of these tables are shown in Figure 3.1a. We expect SQL queries to PayLess are parameterized queries embedded in certain application so that users (e.g., data scientists) issue the queries by specifying the parameter values via a web interface. We do not expect the organization restricts her users the number of queries to the data market because that is counter-productive.

3.2 System Overview

Figure 3.3 shows the architecture of PayLess. It is designed to be lightweight and offloads most query processing to a DBMS query engine. It accepts and parses a SQL query (with parameter values instantiated) ①. The parser differentiates local tables and tables from the data market using the information (e.g., the table name) obtained when registering with the data market (see Figure 3.2). Then, the optimizer of PayLess optimizes the query ② by consulting the statistics of local and data market data ③. The optimized query is then passed to an execution engine ④. A query, after optimization, may be able to skip some or the entire access to the data market. When it is necessary to access the data market, the execution engine will pass the access requests to the data market connector ⑤ and let the connector interact with the data market (5.1) (5.2). PayLess stores all the data market access requests and their returned data in a semantic store (5.3). Whenever new data is retrieved from the data market, PayLess will update its statistics (5.4). In our implementation, we implement our updatable statistics using ISOMER [116]. After this step, all data required by a query should be ready and stored in the DBMS and the execution engine of PayLess instructs the DBMS query engine ⑥ to process the query ⑦. In the end, the execution engine

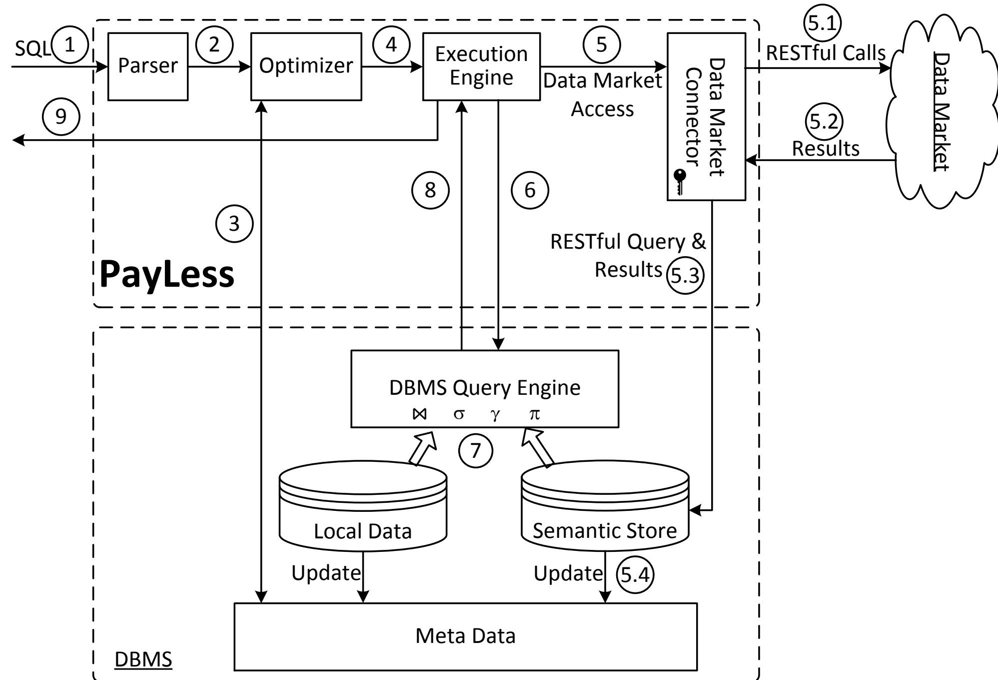


Figure 3.3. System architecture

of PayLess retrieves the query result from the DBMS ⑧ and then returns it to the front end ⑨.

PayLess is supposed to be installed by each data buyer and serves all the end users from the same data buyer. As a data buyer would not be interested in all datasets available in the data market, the storage space (for the DBMS) is not a problem here. Cache management is out of PayLess's interest because we deliberately use cheap storage space to store all intermediate results (i.e., no eviction) in order to eschew retrieving redundant data from the data market. Besides, PayLess is indeed amenable for any updatable statistic. As our focus of this chapter is to give a proof-of-concept first solution, we will test other updatable statistics (e.g., [61]) in place of ISOMER in the next version of PayLess.

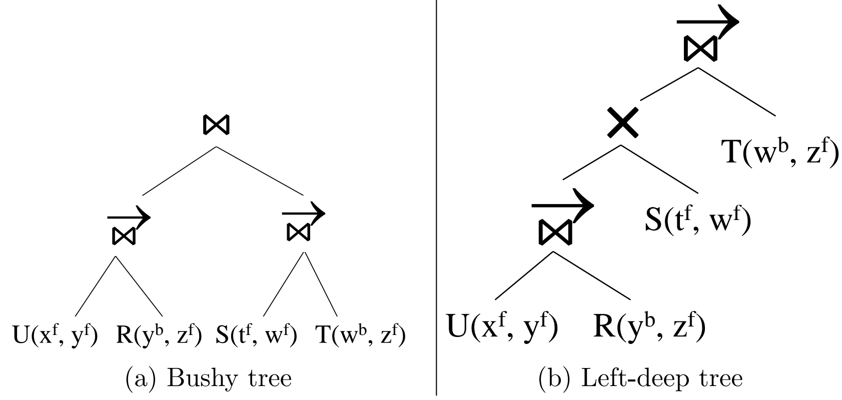
3.3 Query Optimization

PayLess's optimizer follows the typical bottom-up, cost-based, and dynamic programming approach [65]. That is, it first considers the best plan for single relations, then the best plan for joining two relations, and then for three relations, so on. On top of that, PayLess's optimizer considers bind joins $\vec{\bowtie}$ as an access path in addition to the regular join $\bowtie$. The key feature of PayLess's optimizer is that it carries out *semantic query rewriting* to optimize its queries using the query results stored in the semantic store. Semantic query rewriting [53] is not new, but later we will explain why it is not included in limited access query optimizer (e.g., [42,63,117]) and why it is helpful to us here. We will also explain the limitations of current semantic query rewriting techniques in our setting and our solutions to unlock their potential and integrate them into our optimizer.

This section describes how to derive the optimal execution plan after parsing a SQL query. We first propose several techniques to reduce the plan search space and prove their correctness (see Section 3.3.1). After that, we illustrate the semantic query rewriting method used in PayLess (see Section 3.3.2). In the end, we end with some discussions about our query optimization approach (see Section 3.3.3).

3.3.1 Plan Space

When optimizing queries for limited access pattern data sources, *bushy trees* are included in the plan space to avoid plans with Cartesian products [63]. For example, consider a query that joins four relations U, R, S and T with access



(a) Bushy tree

(b) Left-deep tree

Figure 3.4. Bushy tree v.s. Left-deep tree

patterns: $U(x^f, y^f)$, $R(y^b, z^f)$, $S(t^f, w^f)$, $T(w^b, z^f)$. Since R has a bind attribute y, it must require values for attribute y to retrieve tuples. In the example, the only choice is thus to carry out a bind join $U \overrightarrow{\bowtie} R$. Similarly, since T has a bind attribute w, it must require values for attribute w to retrieve tuples. In the example, the only choice is thus to carry out a bind join $S \overrightarrow{\bowtie} T$. After that, the only way is to join them together by using a local join, resulting in a bushy tree like Figure 3.4a. So, if only left-deep plans are allowed, a "logical" cross product must be used to *logically connect* the relations like Figure 3.4b[3.4].

Including bushy trees would significantly enlarge the search space. In our problem setting, as our primary goal is to minimize the money-to-pay, we exclude bushy trees in our plan space because:

THEOREM 1. *Given any plan P, we can transform it to a left-deep plan P' such that $\phi(P) \geq \phi(P')$, where $\phi(\cdot)$ denotes the total price of a plan. In other words, the optimal plan must be one of the left-deep plans.*

[3.4]The cross product is just logically connecting intermediate results U$\overrightarrow{\bowtie}$R and S$\overrightarrow{\bowtie}$T. Physically, (U$\overrightarrow{\bowtie}$R) joins (S$\overrightarrow{\bowtie}$T) is done by the DBMS, using any equi-join implementation like hash-join.

Proof. In what follows, we use the terms RESTful call, leaf node, and relation/table interchangeably.

First, we re-iterate a very important fact:

Fact *Only leaf nodes in P contribute to the price $\phi(P)$ because they represent RESTful calls to the data market. Therefore, $\phi(P)$ equals to the sum of prices of leaf nodes in P.*

Without loss of generality, we name the leaf nodes (RESTful calls) in P from left-to-right as: $C_1, C_2, \cdots, C_n$.

We write $P^{(k)}$ to denote that, for all leaf nodes of P, if named from left-to-right, the first k leaf nodes form a left-deep subtree. So, given a plan P with n leaf nodes, if we write $P^{(n)}$, we mean P is a complete left-deep tree. As an example, for the bushy tree P in Figure 3.4a. $P^{(1)}$ and $P^{(2)}$ hold. As another example, let P be the plan in Figure 3.4b, then we see that $P^{(1)}$, $P^{(2)}$, $P^{(3)}$ and $P^{(4)}$ all hold.

Now, we proceed to prove $\phi(P) = \phi(P^{(1)}) \geq \phi(P^{(2)}) \geq \cdots \geq \phi(P^{(n)})$. In the following, we first prove $\phi(P^{(1)}) = \phi(P)$ and then prove that for a given $1 \leq k \leq n-1$, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$.

Base case: $k = 1$ $P^{(1)}$ simply means we just look at the left-most leaf nodes of P without moving any nodes, so the cost of the whole plan P is unchanged: $\phi(P^{(1)}) = \phi(P)$.

General case: $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$

When C_{k+1} is C_k's uncle: Figure 3.5a illustrates this case. In this case, the left-most $k+1$ leaf nodes form a left-deep subtree. So, P^{k+1} holds. Note

that we did not move any leaf node yet, so the plan cost would not change: $\phi(P^{(k+1)}) = \phi(P^{(k)})$.

When C_{k+1} is not C_k's uncle: Figure 3.5b illustrates this case. In this case, the uncle node of C_k, say U, must be a non-leaf node and its subtree contains C_{k+1}. Let T_F be the left-deep subtree rooted at F, the father of C_k. Further, we let G be the grandfather of C_k. Finally, we let T_{UL}, T_{UR} be the left and right subtrees rooted at U, respectively.

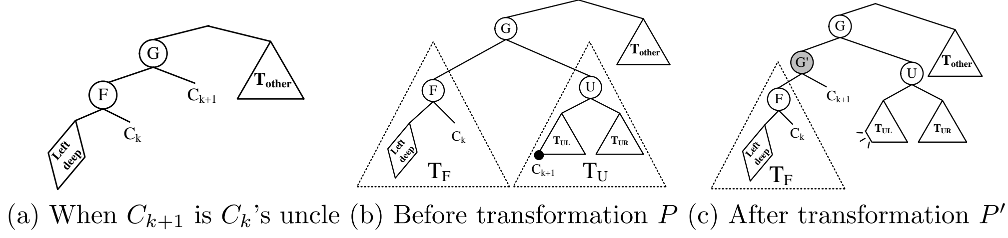
We now explain that making $P^{(k+1)}$ holds by joining T_F with C_{k+1} through a new node G' would not increase the overall plan cost. Figure 3.5c illustrates the resulting plan P' with $P'^{(k+1)}$ holds.

First, we see that the price of subtree T_F is the same among P and P'.

Second, the price of C_{k+1} is the same in both P and P' because C_{k+1} takes the same join result from T_F no matter G or G' is a bind join or a regular (local) join.

Now, we consider the price for each node (other than C_{k+1}) in T_{UL} and T_{UR} in P and P'. Let C_u be such a node. First, if C_u does not require any binding from C_{k+1}, then the price of C_u in P' is unchanged. Second, if C_u requires binding values from C_{k+1}, then the price of C_u depends on the number of distinct binding values from C_{k+1}. Note that in P', C_{k+1} has been joined with the others earlier than P, that causes the number of binding values to C_u possibly decreases. So, the price for C_u would not increase.

Finally, we look at the subtree T_{other}. As the result of the left operand of T_{other} remains the same, the price of T_{other} is unchanged.



(a) When C_{k+1} is C_k's uncle (b) Before transformation P (c) After transformation P'

Figure 3.5. Illustration figures for Theorem 1

As the price of any C_i in P would not increase, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$.

□

Traditional optimizers include only left-deep plans as a heuristic to improve the efficiency of the plan search. In PayLess, with Theorem 1, enumerating only left-deep plans is not a heuristic but with a guarantee that the optimal plan is not lost. Furthermore, in PayLess, including Cartesian product is not a problem because that would not contribute any extra data market transaction.

In addition to enumerating left-deep plans only (Theorem 1), PayLess's optimizer further trims the search space by first joining all relations that incur zero price to the data market. Those relations can either be local relations or relations whose required tuples can be found in the semantic store. In the following, we show that such zero-price-relations-join-first idea retains the optimal plan in the plan space:

THEOREM 2. *Let $P = \langle C_1, C_2, \ldots, C_n \rangle$ be a left-deep plan with a leaf node (RESTful call) C_i whose price $\phi(C_i) = 0$. Then, the plan $P' = \langle C_i, C_1, \ldots, C_n \rangle$ has $\phi(P') \leq \phi(P)$.*

Proof. We divide the other calls into two groups: (1) RESTful calls that executed

before C_i, i.e. C_1 to C_{i-1}, and (2) RESTful calls that executed after C_i, i.e. C_{i+1} to C_n.

If we move C_i to the left-deepest node of P:

- $\phi(C_i)$ is unchanged and remains 0.
- $\phi(C_j)$ for $j > i$ is unchanged because the join results before executing C_j and the possible binding values for C_j are the same.
- $\phi(C_j)$ for $j < i$ cannot increase. If C_j does not use any binding attributes, then moving C_i before C_j would not increase $\phi(C_j)$. If C_j uses binding values from a bind join, then moving C_i before C_j would not increase (but may decrease) the number of bind join values for C_j, and that would not increase $\phi(C_j)$.

□

PayLess's optimizer applies Theorem 2 repeatedly and moves all zero price calls to the leftmost subtree of P. That way, the search space of PayLess's optimizer is further reduced.

Lastly, PayLess's optimizer would prune some candidate subplans during plan enumeration:

THEOREM 3. *When searching for the best plan for a set $\mathcal{C}$ of relations $C_1, C_2, \ldots, C_n$, if $\mathcal{C}$ can be partitioned into disjoint subsets $\mathcal{C}_1 \ldots \mathcal{C}_j$, where relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$ (unless using Cartesian product $\times$). Then the best plan for $\mathcal{C}$ is $Best(\mathcal{C}_1) \times Best(\mathcal{C}_2) \times \ldots Best(\mathcal{C}_j)$, where $Best(\mathcal{C}_i)$ denotes the best plan for the set of relations in $\mathcal{C}_i$.*

Proof. The proof is trivial because the relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$, the price of calling $\mathcal{C}_j$ would not be influenced by $\mathcal{C}_i$. So, the best plan for $\mathcal{C}$ becomes simply connecting the best subplans of $\mathcal{C}_1 \ldots \mathcal{C}_j$ using Cartesian product. □

Consider a chain query that joins four relations: $\mathcal{C} = \{U(v,w),\ R(w,x),\ S(x,y),\ T(y,z)\}$. Assuming that the best plans determined for the pairs of relations are:

	$\{U,R\}$	$\{U,T\}$	$\{U,S\}$	$\{R,S\}$	$\{R,T\}$	$\{S,T\}$
Best Plan	$U \overrightarrow{\bowtie} R$	$U \times T$	$U \times S$	$R \overrightarrow{\bowtie} S$	$R \times T$	$S \overrightarrow{\bowtie} T$

So, when determining the best plan for 3-way join, the candidate plans that would be generated are:

	$\{U,R,S\}$	$\{U,R,T\}$	$\{U,S,T\}$	$\{R,S,T\}$
Candidate	$(U \overrightarrow{\bowtie} R) \bowtie S$	$(U \overrightarrow{\bowtie} R) \bowtie T$	...	...
Plans	$(U \overrightarrow{\bowtie} R) \overrightarrow{\bowtie} S$	$(U \times T) \bowtie R$	...	...
	$(U \times S) \overrightarrow{\bowtie} R$	$(U \times T) \overrightarrow{\bowtie} R$	...	...
	...	$(R \times T) \bowtie U$	...	...
	...	$(R \times T) \overrightarrow{\bowtie} U$	...	...

Observe that the set $\{U,R,T\}$ can be partitioned into two disjoint subsets: $\mathcal{C}_1 = \{U,R\}$ and $\mathcal{C}_2 = \{T\}$. So, we can apply Theorem 3 to determine the best plan for the set $\{U,R,T\}$ as $Best(U,R) \times T$, i.e., $(U \overrightarrow{\bowtie} R) \times T$. In other words, Theorem 3 eliminates many candidates (e.g., $(R \times T) \overrightarrow{\bowtie} U$) and eliminates their associated costing steps and semantic rewriting steps.

Let the total number of candidate plans in all levels of the dynamic pro-

gramming approach be the size of the search space. For a chain query with n relations whose attributes are all free. The use of the above theorems can reduce the search space from $\approx 6^n - 5^n$ down to $\approx 2^{n'} + \frac{2}{3} \cdot n'^3$ with the optimal plan retained, where m is the number of zero price relations and $n' = n - m$. Specifically, the original plan space with dynamic programming is:

$$n + \sum_{k=2}^{n} \left(\binom{n}{k} \cdot (\sum_{i=1}^{k-1} \binom{k}{i} \cdot 4^{min\{i,k-i\}}) \right) \approx 6^n - 5^n$$

where k represents the level in dynamic programming (e.g., when $k = 2$, we consider joining two relations). At level k, there are $\binom{n}{k}$ size-k subsets to be examined. For each size k subset, we can form a plan by: (i) choosing a size i subset for the left subtree (and the complementary size $k - i$ subset for the right subtree), and (ii) deciding the binding attributes for the join (at root). For (ii), each call on the right subtree can bind with attributes from at most 2 calls from the left subtree; thus, there are 2·2=4 binding choices per call, and at most 4^{k-i} choices per plan. We can tighten this number to $4^{min\{i,k-i\}}$ when i is small and the left subtree can provide at most 4^i binding choices.

The plan space of PayLess's optimizer is:

$$4n' + \sum_{k=2}^{n'} \left(4 \cdot k \cdot (n' - k + 1) + (\binom{n'}{k} - (n' - k + 1)) \right)$$
$$\approx 2^{n'} + \frac{2}{3} \cdot n'^3$$

where m is the number of zero price relations and $n' = n - m$. Specifically by Theorem 2, we first build a plan with all local m relations. Then, in dynamic

programming, we consider growing the plan by using the remaining $n' = n - m$ relations. At level k, there are $\binom{n'}{k}$ size-k subsets. We can divide them into (i) disconnected subsets (in which some relations must be joined by Cartesian product), and (ii) connected subsets. For the chain query, there are $n' - k + 1$ connected subsets and $\binom{n'}{k} - (n' - k + 1)$ disconnected subsets. For each disconnected subset, we can compute its best plan directly by Theorem 3. For each connected subset, we can obtain it by Theorem 1, i.e., combining a size-$(k-1)$ subset with a new call. There are k choices for the call and at most $2{\cdot}2{=}4$ binding choices for that call.

3.3.2 Semantic Query Rewriting

In PayLess, we store all RESTful queries issued to the data market and their corresponding results in the semantic store. The objective of doing so is to carry out *semantic query rewriting*, i.e., answer the queries using those stored results so as to reduce the amount of data retrieved from the data market. Semantic query rewriting falls into the category of rewriting queries using views [68, 129]. Given a query Q, a set $\mathcal{V}$ of RESTful queries and their corresponding stored results, the key step in semantic query rewriting is to compute the set $\mathcal{R}em(Q, \mathcal{V})$ of *remainder queries* [53]. The set $\mathcal{R}em(Q, \mathcal{V})$ essentially contains the set of RESTful queries that has to be sent to the data market in order to retrieve the tuples required by Q but not covered by $\mathcal{V}$.

Before we delve deeper, we first explain why optimizers for queries over remote data sources like [42,63,117] do not use semantic query rewriting. Consider our example query $Q1$ (page 1), which inquires about the daily temperature of

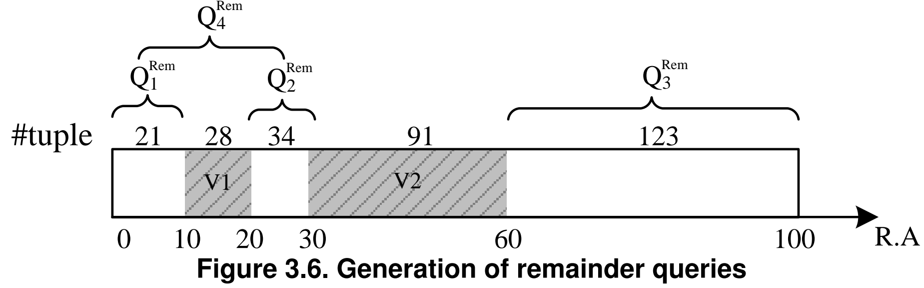


Figure 3.6. Generation of remainder queries

Seattle in June 2014, has been issued, and its 30 resulting tuples (one tuple for each day in June) are stored in the semantic store. Assume that there is another query $Q2$ being issued, with $Q2$ shares the same query template like $Q1$ but the date ranges from May 2014 to July 2014 (3 months). Using semantic query rewriting, $Q2$ will generate two remainder queries: one asks for weather records in May (31 records; 1 transaction), another asks weather records in July (31 records; 1 transaction). The final result is then obtained by union the above with the stored results of $Q1$. The plan of using semantic query rewriting incurs a total of two calls to the external data source. In contrast, only one call to the external data source is required if $Q2$ is sent to the external data source without semantic query rewrite. So, in the context of minimizing the number of calls to external data sources, semantic query rewriting obviously is not a fruitful technique because it decomposes a call to several sub-calls.

Now, we show how we could adapt semantic query rewriting to PayLess's optimizer to yield competitive plans for data market query processing. To illustrate, consider the example in Figure 3.6. The example assumes that the results of two queries V_1 and V_2 have been stored in the semantic store. Both V_1 and V_2 are range queries on an integer attribute A whose domain is $[0, 100]$. V_1 and V_2 respectively cover the ranges $[10, 20)$ and $[30, 60)$ on attribute A and have

retrieved 28 and 91 tuples from table R. In what follows, we write a query Q in the form as

$$Q :- \ R_1(A[s,e], B = \beta, C), R_2(C, ..)$$

which means it joins R_1 and R_2 using C as the join attribute, and tuples in table R_1 have values in numeric attribute A fall between s and e and have values in categorical attribute B equal β.

Now, with V_1 and V_2, we assume the following query Q is posed:

$$Q :- \ R(A[0,100])$$

Using the vanilla semantic query rewriting techniques, it will generate an invalid remainder query $Q^{Rem}_{invalid}$:

$$Q^{Rem}_{invalid} :- \ R(A[0,10) \vee [20,30) \vee [60,100])$$

In data market, $Q^{Rem}_{invalid}$ is invalid because it involves disjunction, which is not supported by the access pattern of data market. Therefore, our first step to adapt semantic query rewriting techniques is to decompose remainder queries that violate the data source access patterns into a set of valid remainder (sub)queries. For the example above, PayLess will generate a set $\mathcal{R}em_1$ of remainder queries:

$Q_1^{Rem} :- \ R(A[0,10))$ //21 tuples; 1 transaction

$Q_2^{Rem} :- \ R(A[20,30))$ //34 tuples; 1 transaction

$Q_3^{Rem} :- \ R(A[60,100])$ //123 tuples; 2 transactions

So, altogether, $\mathcal{R}em_1$ will cost a total 4 transactions.

Note that such straightforward decomposition may not yield the best plan. For example, the following is another possible set of remainder queries $\mathcal{R}em_2$:

$Q_4^{Rem} :- \ R(A[0,30))$ //21+28+34= 83 tuples; 1 transaction

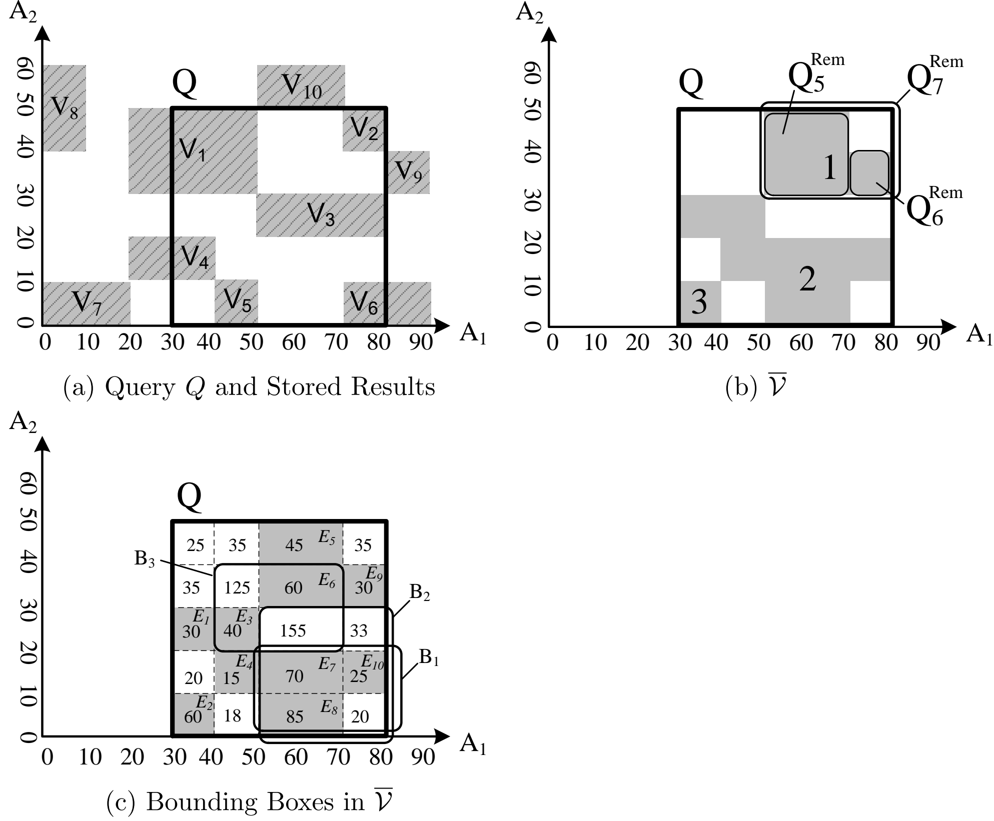
$Q_3^{Rem} :- \ R(A[60,100])$//123 tuples; 2 transactions

The remainder query Q_4^{Rem}, although overlaps with stored query V_1, will still cost $\lceil (21 + 28 + 34)/100 \rceil = 1$ transaction. So, altogether, $\mathcal{R}em_2$ will cost a total 3 transactions only.

The example above illustrates a new and unique issue specific to the generation of remainder queries in data market. Specifically, we see that there are alternate ways to generate valid remainder queries and it is possible that a lower overall price can be achieved even when *a remainder query overlaps with a stored query*.

PayLess obviously does not want to miss the above opportunity when optimizing the queries. So, we have devised a remainder query generation method that leverages the above opportunity to reduce the overall price to access the data market.

We illustrate our idea using a more general example in Figure 3.7a. In the example, the query Q is a 2d-query that inquires table R:



(a) Query Q and Stored Results

(b) $\overline{\mathcal{V}}$

(c) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 3.7. Generation of remainder queries for data market

$$Q :- \quad R(A_1[30, 80], A_2[0, 50])$$

In the example, we assume there are ten RESTful queries $V_1, \ldots, V_{10}$ stored in the semantic store. Figure 3.7b shows the intersection of Q and the complement of $\mathcal{V}$, i.e., the data supposed to be retrieved from the data market. Denoting that space as $\overline{\mathcal{V}}$, there are alternate sets of remainder queries that can retrieve all the missing data. For example, consider the following set of remainder queries $\mathcal{R}em_3$:

$Q_5^{Rem} :- \quad R(A_1[50,70), A_2[30,50])$

$Q_6^{Rem} :- \quad R(A_1[70,80], A_2[30,40])$

$\mathcal{R}em_3$ covers the missing data in region 1. Alternately, the following set of remainder queries $\mathcal{R}em_4$ can also cover data in region 1:

$Q_7^{Rem} :- \quad R(A_1[50,80), A_2[30,50])$

From the above, we see that our goal boils down to finding a set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ using the least number of data market transactions.

To achieve a good solution, we use a two-step approach. The first step aims to generate a set of promising bounding box $\mathcal{B}$ candidates that cover different data regions in $\overline{\mathcal{V}}$. The bounding box candidates may possibly overlap with each other. The second step aims to extract from $\mathcal{B}$ the best set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ in minimum price.

We now elaborate the first step. Specifically, we begin with a decomposition of $\overline{\mathcal{V}}$ into a union $\mathcal{E}$ of disjoint *elementary boxes*. Figure 3.7c shows an example. On each dimension i, we collect a *separator* set S_i from the corners of each elementary box. For example, elementary box E_8 contributes values 50 and 70 to S_1 and contributes values 0 and 10 to S_2. Accounting for all elementary boxes, then we have $S_1 = \{30, 40, 50, 70, 80\}$ and $S_2 = \{0, 10, 20, 30, 40, 50\}$. Then, we exhaustively construct a set $\mathcal{B}$ of *bounding boxes*, where the extent of a bounding box $B \in \mathcal{B}$ on dimension i is picked from any two values in S_i. For example, the bounding box B_1 in Figure 3.7c has extent $[50, 80]$ on dimension A_1 and extent $[0, 20]$ on dimension A_2 when it picks values 50 and 80 from S_1 and values 0 and 20 from S_2. Each resulting bounding box represents a remainder query that

covers certain data to be retrieved from the data market.

Algorithm 3.1 presents the pseudo-code of generating the bounding boxes, with powerful pruning rules to prune unpromising bounding boxes. First, it estimates the number of tuples falling into each elementary box in $\mathcal{E}$ from ISOMER (Lines 2–3). Figure 3.7c shows an illustration with those estimates. We will discuss the case of insufficient/inaccurate statistics in Section 3.3.3. Next, it enumerates a set of bounding boxes from the separator sets $S_1, S_2, ...S_d$, where d is the dimensionality of the query. It applies two pruning rules to discard unpromising bounding boxes.

The first pruning rule (Line 6) prunes a bounding box B if it is not tight. In other words, only *minimum bounding boxes* could stay. Consider the bounding boxes B_1 and B_2 in Figure 3.7c. They both contain the same set of elementary boxes E_7, E_8, E_{10} but B_2 contains B_1. Therefore, B_2 is not a minimum bounding box and is pruned. This makes sense because B_2 has to download an extra $155 + 33$ redundant tuples comparing with B_1.

The second pruning rule (Line 8) prunes a bounding box if its price is not smaller than the price sum of its individual elementary boxes. Consider bounding box B_3 Figure 3.7c. It requires $\lceil (125 + 60 + 40 + 155)/100 \rceil = 4$ transactions. However, if E_3 and E_6 are individually retrieved, they collectively cost only $\lceil 40/100 \rceil + \lceil 60/100 \rceil = 2$ transactions. So, in this case, B_3 is not helpful and is pruned as well.

Algorithm 3.1 would enumerate $\binom{|S_i|}{2}^d$ bounding boxes for a d-dimensional query in the worst case. However, because of the high effectiveness of the pruning rules, the number of (minimum) bounding boxes considered is indeed much fewer

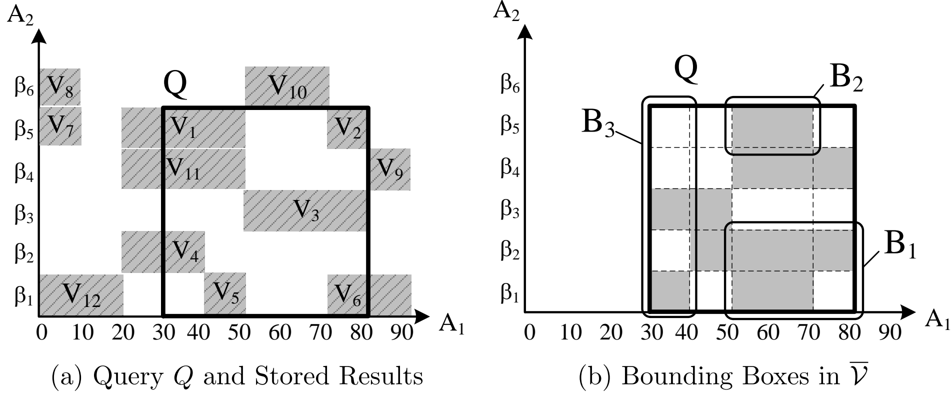
Algorithm 3.1 Generating Candidate Remainder Queries

Input (elementary boxes $\mathcal{E}$, separator sets $\{S_1, S_2, ..., S_n\}$)
Output (A collection of minimum bounding boxes $\mathcal{B}$)
1: initialize $\mathcal{B}$
2: **for** each elementary box E_i in $\mathcal{E}$ **do**
3: E_i.price $\leftarrow$ estimate the price of E_i
4: enumerate every possible bounding box B using the separator sets $S_1, S_2, \ldots, S_n$.
5: **for** each bounding box B **do**
6: **if** B is a minimum bounding box **then** ▷ pruning rule 1
7: estimate the price of B
8: **if** B.price $< \sum_{E_i \in B} E_i$.price **then** ▷ pruning rule 2
9: insert B into $\mathcal{B}$
10: return $\mathcal{B}$

than the worst case in practice.

The second step of our idea is to find the best subset of minimum bounding boxes (generated from Algorithm 3.1) that cover all the elementary boxes (all missing data) in minimum price. This is a weighted set cover problem [51]. Specifically, the weighted set cover problem states that, given (1) a set of elements $\mathcal{E} = \{E_1, E_2, ...\}$ and (2) a family $\mathcal{B}$ of subsets of $\mathcal{E}$, in which each subset in $\mathcal{B}$ is associated with a $cost_i$, find a collection of subsets, namely the cover, $Cover \subseteq \mathcal{B}$, whose union of the elements in $Cover$ is $\mathcal{E}$ and the sum of cost of elements in $Cover$ is the minimum. In our context, we have (1) $\mathcal{E}$ as all elementary boxes and (2) $\mathcal{B}$ as the set of candidate minimum bounding boxes returned by Algorithm 3.1, $cost_i$ is referred as a bounding box 's estimated transactions. To solve this $\mathcal{NP}$-hard problem, we use the greedy algorithm in [51] that runs in $O(|\mathcal{B}| \cdot |\mathcal{E}|)$ time with $(1 + \ln(|\mathcal{B}|))$ approximation ratio.

The generation of bounding boxes for queries with *categorical attributes* is illustrated as follows. Figure 3.8a shows an example similar to the previous one



(a) Query Q and Stored Results (b) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 3.8. Generation of remainder queries with a categorical attribute A_2

but with attribute A_2 now becomes a categorical attribute with the following domain: $\{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$. We remark that there are no stored queries that can span across multiple categorical values because of the limitation of the access interface.

Figure 3.8b shows the corresponding space $\overline{\mathcal{V}}$. Since A_2 is a categorical attribute, the bounding box B_1, which represents the following remainder query, is invalid:

$$:- \quad R(A_1[50, 80), A_2 = \beta_1 \vee A_2 = \beta_2)$$

Therefore, we will only generate bounding boxes that span either one value or the whole domain of a categorical attribute. For example, bounding boxes B_2, which represents the following remainder query, is valid and would be generated:

$$:- \quad R(A_1[50, 70), A_2 = \beta_5)$$

Similarly, bounding boxes B_3, which represents the following remainder query, is also valid and would be generated:

$$:- \ R(A_1[30, 40))$$

The generation of bounding boxes for queries with *bind joins* is illustrated as follows. Consider a relation U with binding pattern $U(A_1^f, A_2^f)$ and a relation S with binding pattern $S(A_2^b, A_3^f)$, where all attributes are integer attributes. Further, consider a query V that joins U and S:

$$V :- \ U(A_1[2, 3], A_2), S(A_2, A_3[10, 15])$$

V needs a bind join because A_2 is a bind attribute. So, assume that there are four tuples t_1, t_2, t_3, and t_4 in U having values within the range $[2, 3]$ in attribute A_1 and their corresponding values in attribute A_2 are 2, 5, 9, and 10, respectively. Then, the bind join is carried out with S by binding the values 2, 5, 9, and 10 to S's attribute A_2. Note that when retrieving tuples from S whose attribute A_2 has a value, say, 2, those tuples have to satisfy the other condition $A_3[10, 15]$ as well. Figure 3.9a illustrates the above process.

Now, assume the query results of V are stored in the semantic store and let us consider a query Q that shares the same query template as V but with a different query range:

$$Q :- \ U(A_1[2, 5], A_2), S(A_2, A_3[8, 18])$$

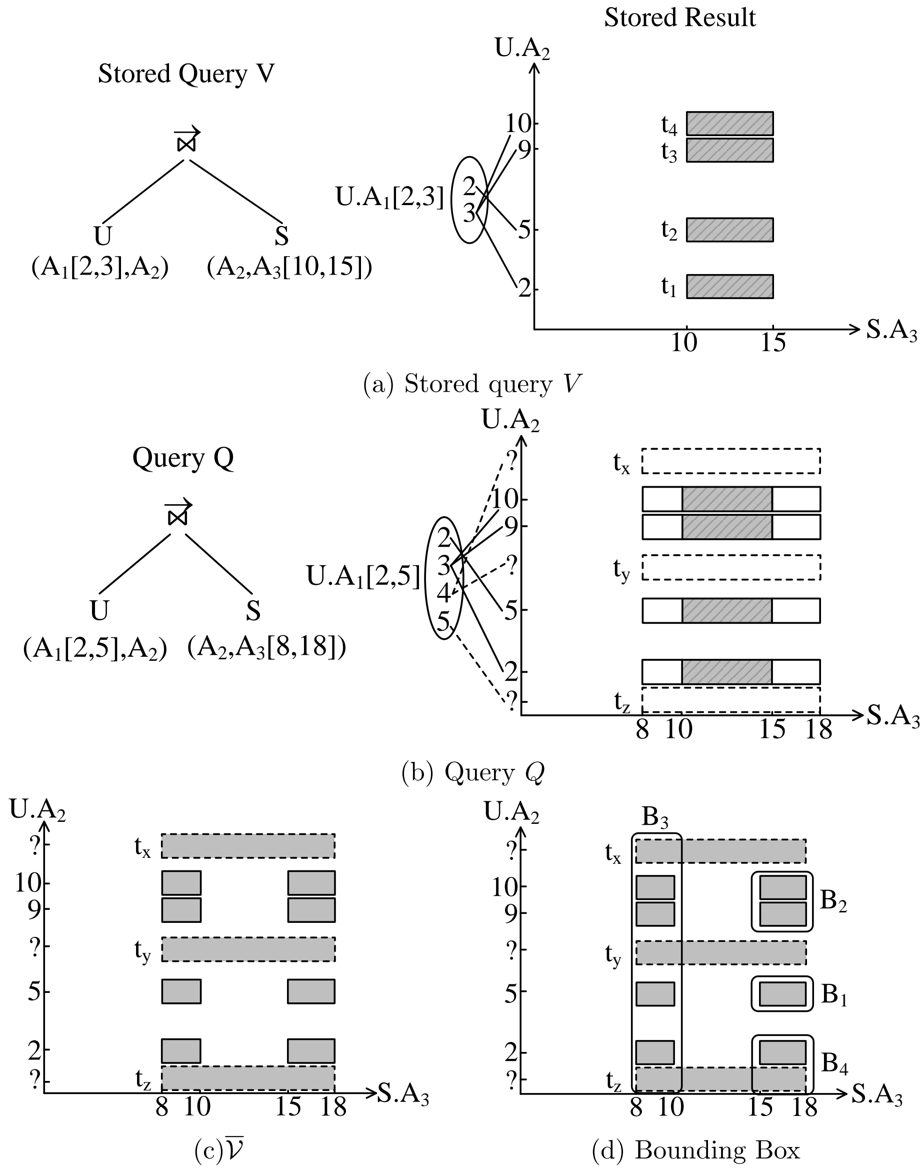


Figure 3.9. Example for 2D-Bind

Note that in this case, assuming that we can estimate that two tuples t_x and t_y will be retrieved from U for $A_1 = 4$, one tuple t_z will be retrieved from U for $A_1 = 5$ (we don't need to estimate the cardinality for $A_1 = 2$ and $A_1 = 3$ because we know the exact cardinality from V), exact values of t_x, t_y, t_z's attribute A_2 are

Algorithm 3.2 PayLess Query Optimization

Input (a query Q, a set $\mathcal{V}$ of RESTful queries and their stored results, the metadata $\mathcal{M}$ for cost estimation)

Output (the optimal plan $P^* : Best(Q)$ for the query Q)

1: $\mathcal{R}_{local} \leftarrow \{C_i \in Q : \phi(C_i) = 0\}$; $\mathcal{R}' \leftarrow \{C_i \in Q\} - \mathcal{R}_{local}$
2: $P_{local} \leftarrow$ the best subplan for $\mathcal{R}_{local}$; found by offloading to a DBMS's optimizer
3: **for** each $C_i \in Q$ **do** ▷ size-1 subplans
4: $\quad Best(C_i) \leftarrow \texttt{SemanticRewrite}(C_i, \mathcal{V}, \mathcal{M})$
5: execute Line 1 again to update $\mathcal{R}_{local}$ and $\mathcal{R}'$
6: **for** each k from 2 to $|\mathcal{R}'|$ **do** ▷ Theorem 2
7: $\quad$ **for** each size-k subset $\mathcal{R}^k$ of $\mathcal{R}'$ **do**
8: $\quad\quad$ **if** $\mathcal{R}_{local} \cup \mathcal{R}^k$ form ℓ disjoint subsets **then** ▷ Theorem 3
9: $\quad\quad\quad Best(\mathcal{R}^k) \leftarrow Best(\mathcal{R}^k_1) \times Best(\mathcal{R}^k_2) \times \cdots \times Best(\mathcal{R}^k_\ell)$
10: $\quad\quad$ **else for** each call $C_i \in \mathcal{R}^k$ ▷ Theorem 1
11: $\quad\quad\quad$ rewrite C_i as $\overrightarrow{C_i}$ by using binding from $P_{local} \bowtie Best(\mathcal{R}^k - C_i)$
12: $\quad\quad\quad P_{bind} \leftarrow \texttt{SemanticRewrite}(\overrightarrow{C_i}, \mathcal{V}, \mathcal{M})$
13: $\quad\quad\quad P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \bowtie Best(C_i)$
14: $\quad\quad\quad$ **if** $\phi(P_{bind}) \leq \phi(Best(C_i))$ **then**
15: $\quad\quad\quad\quad P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \overrightarrow{\bowtie} P_{bind}$
16: $\quad\quad\quad$ update $Best(\mathcal{R}^k) \leftarrow P_{temp}$ if $\phi(Best(\mathcal{R}^k)) \geq \phi(P_{temp})$

still unknown (denoted as ? in Figure 3.9b). In this case, it will generate $\overline{\mathcal{V}}$ like Figure 3.9c. Consequently, when enumerating the set of candidate bounding boxes, we can generate a bounding box for each individual elementary box (e.g., B_1), for a range of known values (e.g., B_2), or for the whole domain (e.g., B_3). In contrast, we cannot generate a bounding box like B_4 because the exact value for A_2 of t_z is actually unknown.

Algorithm 3.2 shows the pseudo code of PayLess optimization. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a walkthrough here.

3.3.3 Discussion

We end this section with a number of discussions about our query optimization approach.

Optimization without statistics. First, as in traditional cost-based query optimization, our approach relies on metadata like histograms. In the beginning when no rich statistics such as value distributions are available, PayLess's optimizer would carry out the cardinality estimation using the basic textbook methods (e.g., using the domain size and uniform distribution assumption).

Consistency in PayLess. Second, answering a query using the stored query results may include obsolete tuples if datasets permit *in-place* data update. However, so far the datasets we found in Windows Azure Marketplace are append-only. In case in-place data update exists, we will introduce several *consistency levels* into PayLess. That would allow organizations that install PayLess to choose between consistency levels like *(i) weak consistency* or *(ii) full consistency.* If the data buyers allow partially obsolete results, then weak consistency will be an appropriate choice. Weak consistency means all RESTful queries and their results are stored in the semantic store (with obsolete results get updated if new results are retrieved). Under weak consistency, semantic query rewriting is always enabled. If the data buyers require accurate results, then strong consistency will be a better choice. Strong consistency means semantic query writing is simply disabled and PayLess always go to the data market to obtain the latest results. These options are trade-off between price-to-pay and the freshness of the result, and Table 3.1 compares these two options.

Table 3.1. Query result accuracy options

Method	Result Accurate?	Query Execution Price
PayLess	No	Low
PayLess w/o SQR (i.e., disabling semantic query rewriting in PayLess)	Yes	High

Organization in semantic store. Third, the data in the semantic store includes issued RESTful queries and records received from the data market. The records are stored in hard disk and used in query execution in local DBMS. Thus, the organization of retrieved records is handled by the local DBMS query execution engine instead of PayLess. For the RESTful queries, they are clustered according to the relation they accessed in the data market. Each time, when a new RESTful query is inserted into semantic store, it will be merged with existing RESTful queries of the same relation. Since the number of RESTful queries is not large, we do not build any index on them.

3.4 Experimental Evaluation

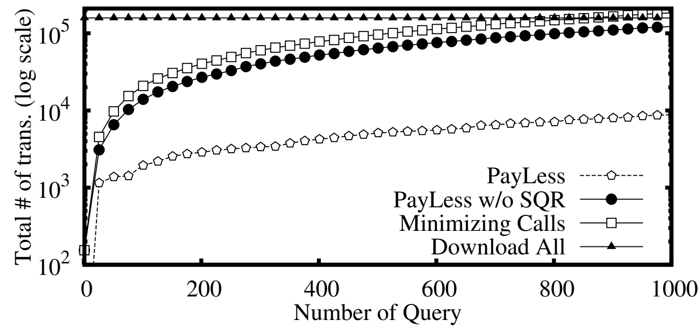
PayLess aims to help organizations to pay less when their end users have to query against the data market. Without PayLess, one option is to employ query optimizers for data sources with limited access pattern because those optimizers at least consider binding patterns and bind joins in their architecture. Another option is to download all required tables from the data market upfront and carry out local processing afterwards. Notice this "Download All" option is not always bad. First, it is optimal if the queries have to scan the whole dataset. In this case,

Table 3.2. Query templates on real DataSets

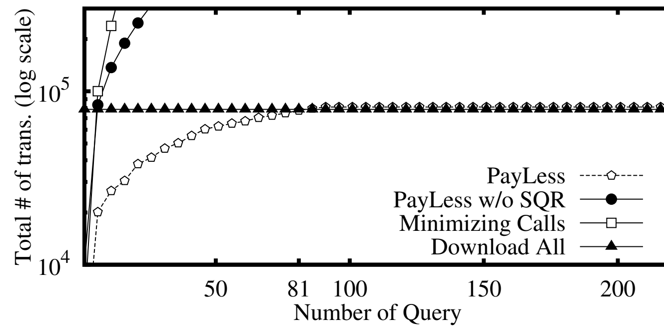
Q_1	SELECT * FROM Weather WHERE Weather.Country = ? AND Weather.Date >= ? AND Weather.Date <= ?
Q_2	SELECT COUNT(ZipCode) FROM Pollution WHERE Pollution.Rank >= ? AND Pollution.Rank <= ?
Q_3	SELECT AVG(Temperature) FROM Station, Weather WHERE Station.Country = Weather.Country = ? AND Weather.Date >= ? AND Weather.Date <= ? AND Station.StationID = Weather.StationID GROUP BY City
Q_4	SELECT Temperature FROM Station, Weather, ZipMap WHERE Station.Country = Weather.Country = ? AND ZipMap.ZipCode = ? AND Weather.Date >= ? AND Weather.Date <= ? AND Station.City = ZipMap.City AND Station.StationID = Weather.StationID
Q_5	SELECT * FROM Pollution, Station, Weather, ZipMap WHERE Station.Country = Weather.Country = ? AND Weather.Date >= ? AND Weather.Date <= ? AND Pollution.Rank >= ? AND Pollution.Rank <= ? AND Pollution.ZipCode = ZipMap.ZipCode AND ZipMap.City = Station.City AND Station.StationID = Weather.StationID

once the whole dataset is downloaded, all queries can work on the downloaded data locally. Second, if the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set, then downloading the whole dataset upfront would be a more economical option. However, we re-iterate that it is always tough to predict how many user queries would eventually be issued in practice. Consider that the users walk away from the dataset forever after issuing just a few queries (maybe due to no interesting information is found), then downloading the whole dataset would become a very costly option.

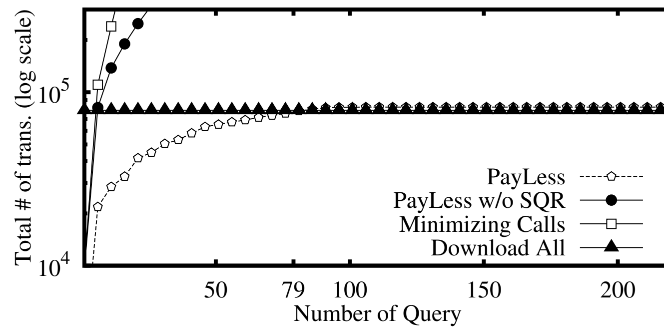
In this section, we evaluate the effectiveness of PayLess using both real data and synthetic data. Specifically, we extract query templates from a meteoro-



(a) Real data



(b) TPC-H



(c) TPC-H skew

Figure 3.10. Overall effectiveness

logical application that involves queries to the Worldwide Historical Weather (WHW) [26] and Environmental Hazard Rank (EHR) [12] datasets in Windows Azure Marketplace. Table 3.2 lists the query templates and Figure 3.1a lists the sizes of the tables. We generate *valid* query instances from those templates

by randomly assigning values to the parameters. A query instance is valid if it returns non-empty results (e.g., we would not instantiate Q_4 with a country equals 'USA' but a zip code in Germany). We also use the TPC-H workload in the experiments. We generate 1G of TPC-H data and 1G of TPC-H skew data [45] with $zipf = 1$. All parametric attributes in TPC-H queries are set as free attributes in the experiments. We set the relations `Nation` and `Region` local. By default, we set 100 tuples as one transaction (i.e., $t = 100$).

Overall effectiveness. We first study the overall effectiveness of PayLess under different workloads and datasets. For comparison, we include the results of using [63] to optimize the queries (denoted as "Minimizing Calls" in the figure). We also include the results of disabling semantic query rewriting (SQR) in PayLess (denoted as "PayLess w/o SQR" in the figure). We respectively generated q query instances per template. The query instances are issued in a random order and the results are reported as an average over 30 repeated experiments. In this experiment, we set $q = 10$ and $q = 200$ for TPC-H workload and real workload, respectively.

Figure 3.10a illustrates the total (cumulative) number of data market transactions used to answer the real queries. Except the "Download All" option, when more queries are issued, the total (cumulative) number of data market transactions increases. Comparing with those data buyers who recklessly download the whole dataset upfront, PayLess can now help them to answer the queries using about two orders of transactions fewer. The number of transactions used by PayLess grows slowly because many queries are rewritten using the stored results in the semantic cache. PayLess can answer the queries using about an order of

transactions fewer than queries optimized using [63]. That is because semantic query rewriting (SQR) is not applicable to their setting but is a powerful helper here in our data market setting. When we disable SQR, PayLess still outperforms [63]. That is because PayLess can find optimal plans in a reduced search space using progressively refined statistics. In contrast, [63] has to find plans in a larger search space (including bushy trees) using heuristics.

Figures 3.10b and c show the results of using TPC-H workload. TPC-H queries scan a large portion of data. Therefore, without rewriting the queries using the stored data, each query optimized by [63] and PayLess (if SQR is disabled) would retrieve a large portion of the data from the data market, and those data are largely overlapping with each other. That explains why they are worse than "Download All", because the latter only downloads the whole dataset once. When PayLess is in full power with semantic query rewriting, we see that the subsequent queries can largely reuse the stored results, thereby saving a lot more transactions than "Download All" until about 80 queries have been issued. When about 80 queries have been issued, all the data required by TPC-H queries (indeed the whole TPC-H dataset) are stored by PayLess, therefore PayLess would not repeatedly retrieve the data from the data market anymore. From the above experimental results, we regard PayLess to be practically better than "Download All" in all means because nobody could have known the number of queries to be issued and the distribution of the data in practice. A data buyer can freely query against any dataset in the data market and walk away from that dataset anytime — she does not need to worry whether it is worth or not to download the whole dataset in the beginning, or switch to download the whole dataset when she finds out that she has to ask more queries after she has burned a certain amount of

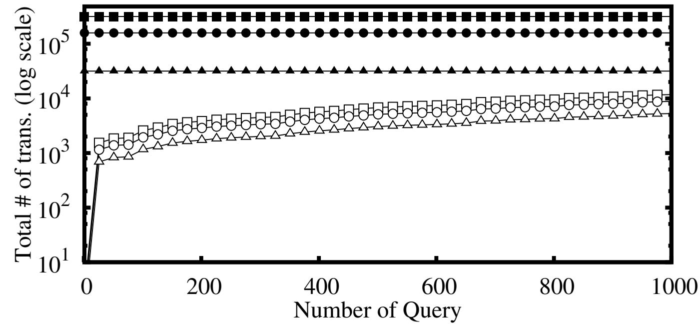
money.

Influence of number of tuples per transaction. We next study whether the effectiveness of PayLess would be influenced by the number of tuples per transaction, which could be a different value in different data markets. Since [63] is consistently outperformed by PayLess in all our experiments, so we remove it, together with PayLess with semantic query rewriting disabled, from our discussion.

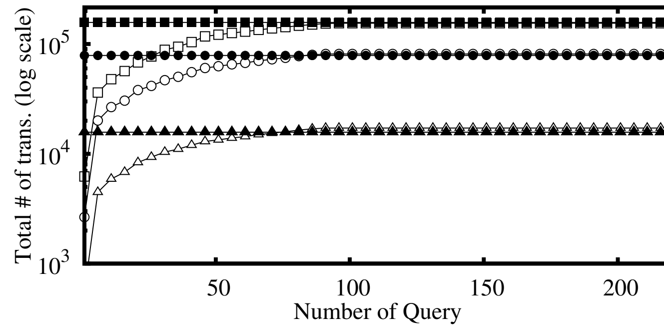
Figure 3.11 shows the effectiveness of PayLess when we vary the number of tuples per transaction t. Note that when t is smaller, more transactions are required to retrieve the same number of tuples from the data market. Therefore, the number of transactions used by both PayLess and "Download All" must increase. Nevertheless, we see that the effectiveness of PayLess is not influenced by that data market parameter. PayLess still outperforms "Download All" under real data in all cases. In addition, it still outperforms "Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

Influence of number of query instances per query template. We next study whether the effectiveness of PayLess would be influenced by q, the number of query instances per query template. Figure 3.12 shows that the effectiveness of PayLess is not influenced by that parameter. We see that PayLess still consistently outperforms "Download All" on real data in all cases. In addition, it still outperforms "Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

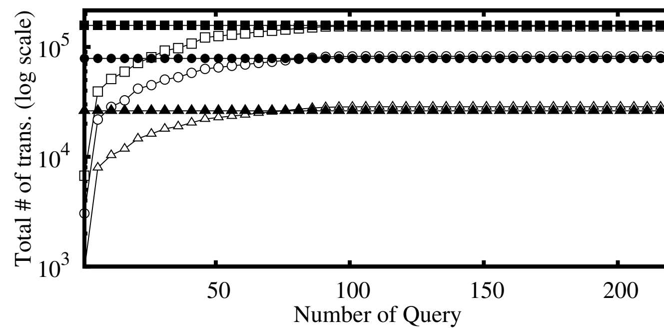
Influence of data size. We also study whether the effectiveness of PayLess



(a) Real data



(b) TPC-H



(c) TPC-H skew

Figure 3.11. Varying the number of results t per transaction

would be influenced when the size of the data is varied. As we cannot control the size of the real data, we control only the size of the synthetic data.

Note that when the data size increases, “Download all” needs more trans-

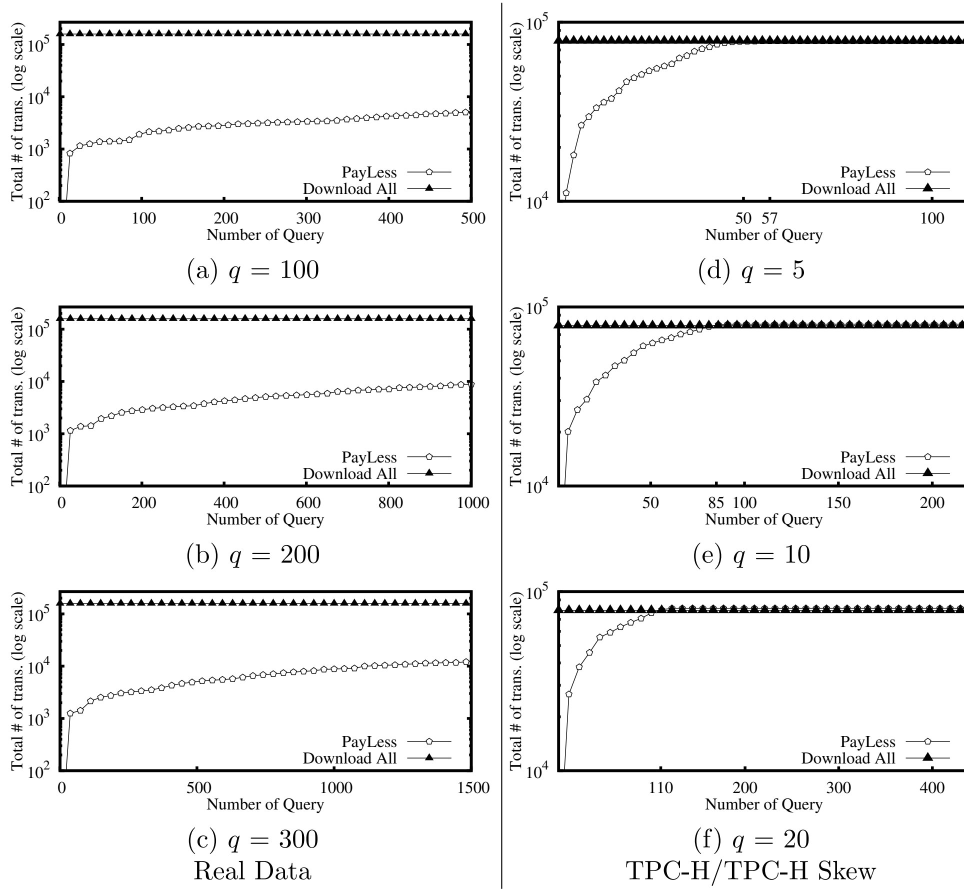


Figure 3.12. Varying the number of query instances (q) per template

actions to download the whole dataset. But PayLess also needs to retrieve more tuples for each query. Figure 3.13 shows that PayLess still outperforms "Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

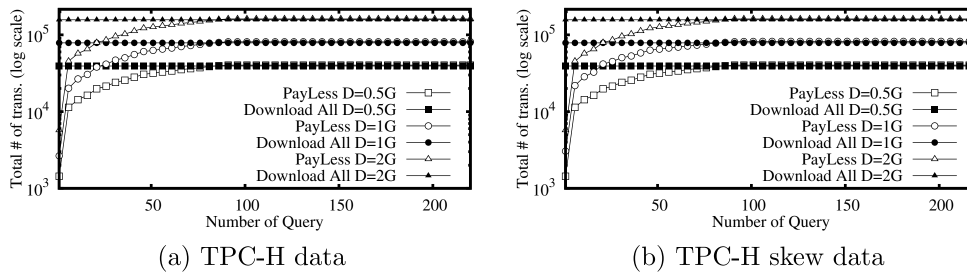
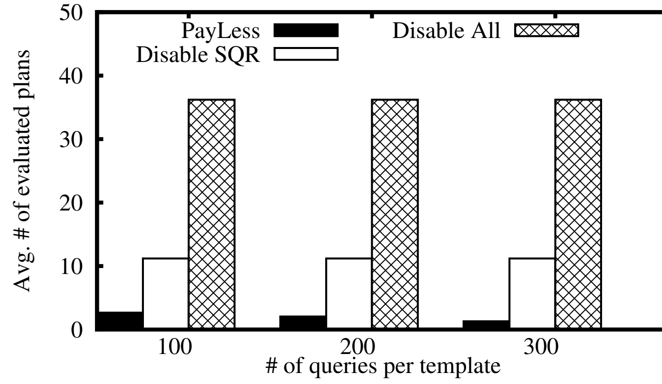


Figure 3.13. Varying data size

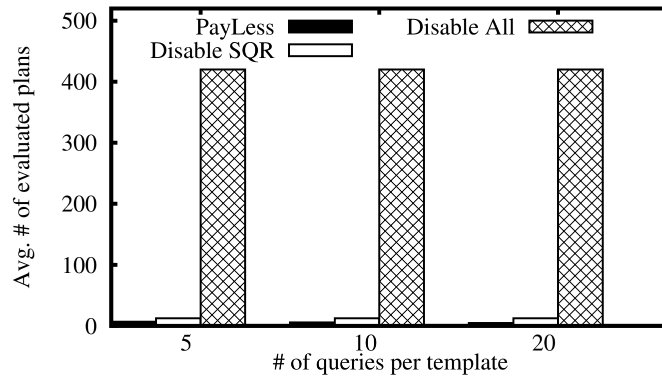
Effectiveness of search space reduction techniques. We have also carried out an experiment to evaluate the effectiveness of our techniques devoted to reducing the search space size. Figure 3.14 shows the average number of candidate (sub)plans for all query instances under our default setting. We report the case when (i) SQR is disabled (Disable SQR), (ii) both SQR and search space pruning (Theorems 1 to 3) are disabled (Disable All), and (iii) nothing is disabled (PayLess). We can see that our techniques significantly reduce the search space by orders of magnitude. This is actually what enables us to look for optimal plans. We notice that enabling SQR indeed reduces the search plan because SQR would cause some relations become local, which can then trigger Theorem 2. This also explains why the average number of candidate (sub)plans PayLess has to considered decreases when we increase the number of query instances generated for each template. That is because if we increase the number of query instances generated for each template, that would retrieve more data from the data market, which in turn increases the chance of using Theorem 2 to reduce the search space.

Effectiveness of bounding box pruning. Our last experiment is to evaluate the effectiveness of the bounding box pruning rules in Algorithm 3.1. Figure 3.15 shows the average number of bounding boxes generated for all query instances under our default setting. We see that the two pruning rules can reduce about an order bounding boxes generated.

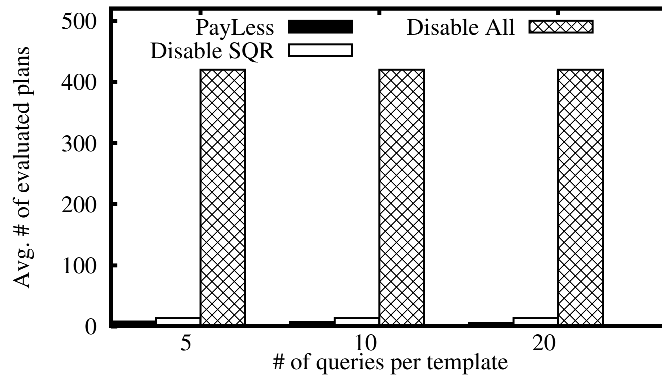
Efficiency. In all experiments, to find the optimal execution plan, PayLess optimizer takes about 0.01s in average over real data and 0.05s over TPC-H



(a) Real data

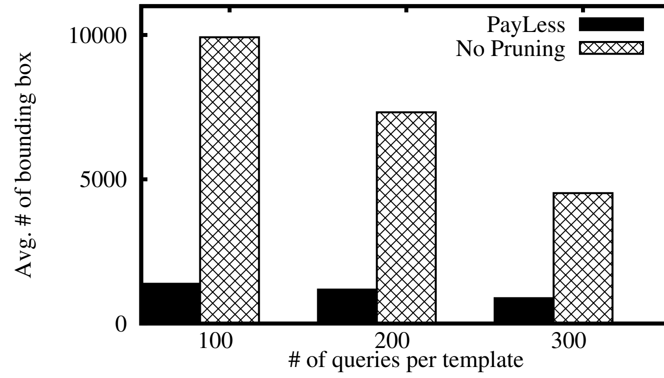


(b) TPC-H data

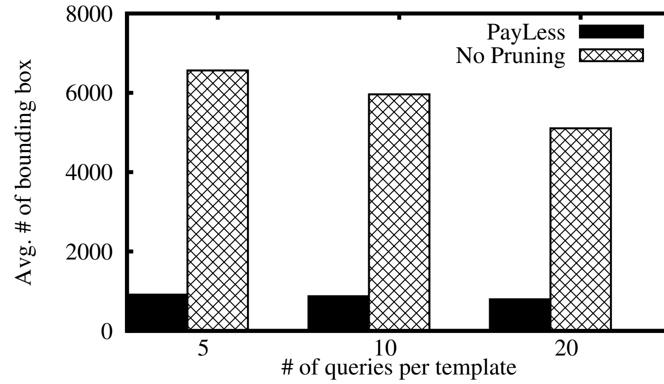


(c) TPC-H skew data

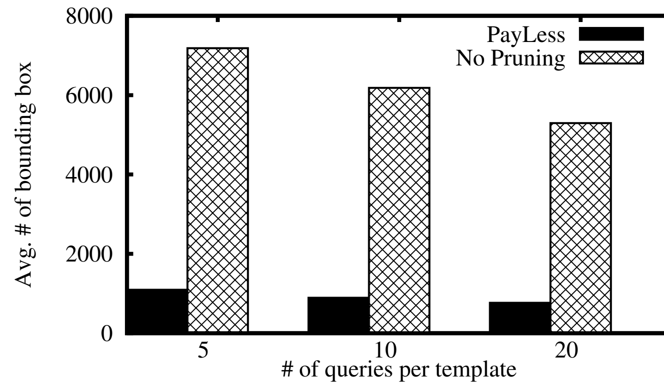
Figure 3.14. Effectiveness of search space reduction techniques



(a) Real data



(b) TPC-H data



(c) TPC-H skew data

Figure 3.15. Effectiveness of bounding box pruning rules

data to find the optimal execution plan. For comparison, it takes about 2s in average to access one page (i.e., a transaction with no more than 100 records) from Microsoft Azure Data Marketplace. Thus, the execution time of a query is dominated by the RESTful calls to the data market.

3.5 Related Work

The related work relevant to this chapter is described in Chapter 2,especially in Sections 2.1 and 2.2. This section will present the differences between this chapter and those related work presented in Chapter 2.

To the best of our knowledge, this chapter is the first to tackle the issue of optimizing queries that access the data market. So far, projects related to the data market are mainly developed for *query market*. In their setting, the query market can support SQL. A data buyer sends a SQL query that accesses a dataset in the query market. The query market computes the results of the query and returns the answer to the buyer. The research focus is how to set the price of arbitrary SQL queries (e.g. [28, 35, 79, 80, 94, 97, 125]). The setting of query market is different from our data market setting. Specifically, existing data market like Windows Azure Marketplace [4] and Xignite [27] are still charging data buyers according to the size of retrieved data.

In terms of problem setting, PayLess is indeed more similar to projects that support queries over remote data sources with limited access patterns (e.g., [42, 47, 59, 63, 86–89, 104, 117]). Nevertheless, as mentioned, all these projects have a very different focus with us — they are designed to minimize the number of

calls to external data and/or the execution time. In contrast, PayLess focuses on minimizing the amount of intermediate retrieved data measured in terms of data market transactions. Besides, the optimization of distributed queries with semi-join/magic sets [46, 112] are similar to PayLess; however, they do not consider limited access patterns.

In terms of implementation, PayLess has borrowed the idea of learning optimizer from LEO [118] and has used feedback driven histogram ISOMER [116]. However, PayLess has to develop its own architecture, construct its own plan search space, and devise its own semantic query rewriting technique (e.g., [50, 53, 84, 105]) to fit the data market. In computational geometry, the problem of partitioning an orthogonal polygon into rectangles (PiR) [62] is similar to our remainder query generation problem, but they are not the same. Using Figure 3.7b as an example, the PiR problem would NOT consider Q_7^{Rem}, which contains some empty regions. In contrast, in our context, Q_7^{Rem} could be a good choice according to our cost function.

3.6 Chapter Summary

This chapter presents PayLess, a system that helps data buyers to freely query against any dataset in the data market and walk away from that dataset anytime. The data buyers do not need to worry whether it is worth or not to download the whole dataset in the beginning. They can simply issue their queries to PayLess and PayLess optimizes their queries with the objective of minimizing their money-to-pay-to-data-sellers. Currently, our use-case does not cover many end users using PayLess simultaneously. When it does, we will incorporate multi-

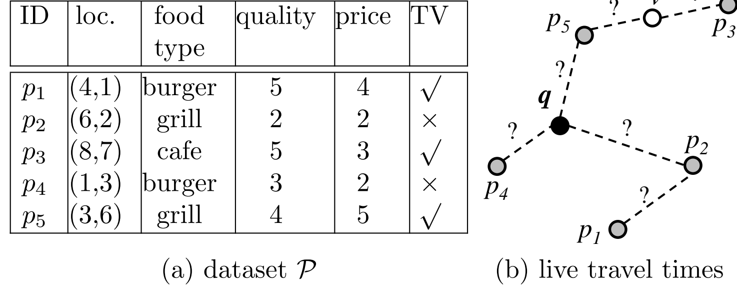
query optimization in PayLess if users are willing to defer theirs to become a batch.

Chapter 4

Query Processing using Route APIs

The availability of GPS-equipped smartphones leads to a huge demand of location-based services (LBSs), like city guides, restaurant rating, and shop recommendation websites, e.g.,OpenTable, Hotels, UrbanSpoon.[4.1] They manage points-of-interest (POIs) specific to their applications, and enable mobile users to query for POIs that match with their preferences and time constraints. As an example, consider a restaurant rating website that manages a dataset of restaurants $\mathcal{P}$ (see Fig. 4.1a) with various attributes like: location, food type, quality, price, etc. Via the LBS (website), a mobile user q could query restaurants based on these attributes as well as travel times on road network to reach them. Here are examples for a range query and a KNN query, based on travel times on road network.

[4.1] www.opentable.com www.hotels.com www.urbanspoon.com



ID	loc.	food type	quality	price	TV
p_1	(4,1)	burger	5	4	√
p_2	(6,2)	grill	2	2	×
p_3	(8,7)	cafe	5	3	√
p_4	(1,3)	burger	3	2	×
p_5	(3,6)	grill	4	5	√

(a) dataset $\mathcal{P}$ (b) live travel times

Figure 4.1. A restaurant rating website: data and queries

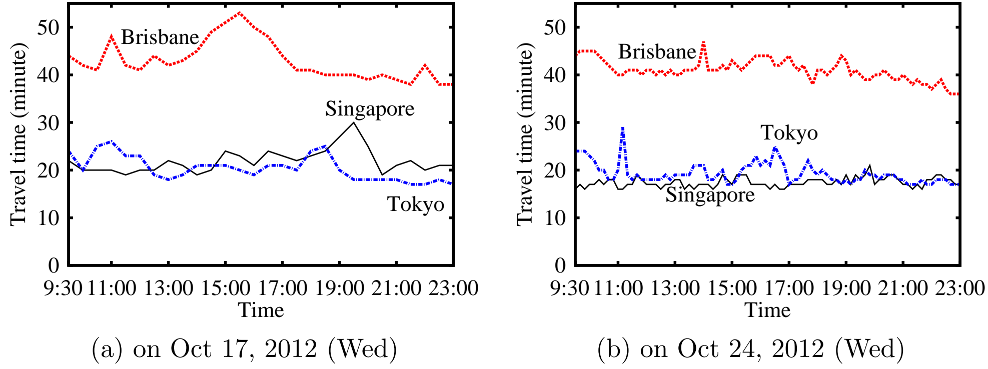
```
select *  from P  where  P.TV = 'yes'
                    and TIME(q,P.loc) < 10
```

```
select *  from P  where  P.price < 5
       order by TIME(q,P.loc)  limit 2
```

A successful LBS must fulfill two essential requirements: ($\mathcal{R}1$) accurate query results, and ($\mathcal{R}2$) reasonable response time. Query results with inaccurate travel times may disrupt the users' schedules, cause their dissatisfaction, and eventually risk the LBS losing its users and advertisement revenues. Similarly, high response time may drive users away from the LBS.

Observe that the live travel times from user q to POIs vary dynamically due to road traffic and factors like rush hours, congestions, road accidents. As a case study, we used Google Maps to measure the live travel times for three pairs of locations in Brisbane, Singapore, and Tokyo, on two days (see Fig.4.2). Even on the same weekday (Wednesday), the travel times exhibit different trends. Thus, historical traffic data may not provide accurate estimates of live travel times.

Unfortunately, if the LBS estimates travel times based on only local information (distances of POIs from user q), then query results (for range and KNN)



(a) on Oct 17, 2012 (Wed) (b) on Oct 24, 2012 (Wed)

Figure 4.2. Measurement of live travel times

would have low accuracy (50% for NoAPI, see Fig. 4.7). Typical LBS lacks the infrastructure and resources (e.g., road-side sensors, cameras) for monitoring road traffic and computing live travel times [131] [132]. To meet the accuracy requirement ($\mathcal{R}1$), the framework SMashQ [131] [132] is proposed for the LBS to answer KNN queries accurately by retrieving live travel times (and routes) from *online route APIs* (e.g., Google Directions API [16], Bing Maps API [6]), which have live traffic information [15]. Given a query q, the LBS first filters POIs by local attributes in $\mathcal{P}$. Next, the LBS calls a route API to obtain the routes (and live travel times) from q to each remaining POI, and then determines accurate query results for the user. As a remark, online maps (e.g., Google Maps, Bing Maps), on the other hand, cannot process queries for the LBS either, because those queries may involve specific attributes (e.g., quality, price, facility) that are only maintained by the LBS.

Using online route APIs raises challenges for the LBS in meeting the response time requirement ($\mathcal{R}2$). It is important for LBS to reduce the number of route requests for answering queries because a route request incurs considerable time (0.1s-0.3s) which is high compared to CPU time at LBS (see Fig. 4.8 and 4.11).

SMashQ [131] [132] obtains the latest travel times for queries from online route API. Though it guarantees accurate query results, it may still incur a considerable number of route requests.

In this chapter, we exploit an observation from Fig. 4.2, namely that travel times change smoothly within a short duration. Routes recently obtained from online route APIs (e.g., 10 minutes ago) may still provide accurate travel times to answer current queries. This property enables us to design a more efficient solution for processing range and KNN queries. Our experiments show that our solution is 3 times more efficient than SMashQ, and yet achieves high result accuracy (above 98%). Specifically, our method Route-Saver keeps at the LBS the routes which were obtained in the past δ minutes (from an online route API), where δ is the expiry time parameter [57]. For instance, based on Fig. 4.2, we may set δ to 10 minutes. These recent routes are then utilized to derive lower/upper bounding travel times to reduce the number of route requests for answering range and KNN queries.

Another related work [121] studies how to cache shortest paths for reducing the response times on answering shortest path queries (but not range/KNN queries in this chapter). They mainly exploit the *optimal subpath* property [52] of shortest paths, i.e., all subpaths of a shortest path must also be shortest paths. Given a shortest path query (s, t), if both nodes s, t fall on the same (cached) shortest path, then the shortest path from s to t can be extracted from that cached path. Unfortunately, this optimal subpath property is not powerful enough in reducing the number of route requests significantly in our problem. This is because each path contains a few data points and thus the probability for points lying on the same path with the query point is small. We show in

an experiment (see Fig. 4.10) that the optimal subpath property ('$\tau_{\mathcal{L}}$' in black) saves very few route requests, whereas our techniques (e.g., lower/upper bounds to be discussed below) provide the major savings in route requests. Furthermore, Ref. [121] has not considered the expiry time requirement as in our work.

To reduce the number of route requests while providing accurate results, we combine information across multiple routes in the log to derive tight lower/upper bounding travel times. We also propose effective techniques to compute such bounds efficiently. Moreover, we examine the effect of different orderings for issuing route requests on saving route requests. And we study how to parallelize route requests in order to reduce the query response time further.

In the following, we first review related work in Section 4.1. Then, we describe the system architecture and our objectives in Section 4.2. Our contributions are:

– Combine information across multiple routes in the log to derive lower/upper bounding travel times, which support efficient and accurate range and KNN search (Section 4.3);

– Develop heuristics to parallelize route requests for reducing the query response time further (Section 4.4);

– Evaluate our solutions on a real route API and also on a simulated route API for scalability tests (Section 4.5).

Finally, we conclude this chapter in Section 4.6.

4.1 Related Work

The related work relevant to this chapter is described in Chapter 2,especially Section 2.2 and Section 2.3. This section will present the differences between this chapter and the related work presented in Chapter 2.

4.1.1 Query Processing on Road Networks

Indexing on road networks have been extensively studied in the literature [69, 73, 77, 101, 108]. Various shortest path indices [69, 73, 108] have been developed to support shortest path search efficiently. Papadias et al. [101] study how to process range queries and KNN queries over points-of-interest (POIs), with respect to shortest path distances on a road network. The evaluation of range queries and KNN queries can be further accelerated by specialized indices [69, 77, 108].

In our problem scenario, query users require accurate results that are computed with respect to live traffic information. All the above works require the LBS to know the weights (travel times) of all road segments. Since the LBS lacks the infrastructure for monitoring road traffic, the above works are inapplicable to our problem. Some works [54, 74] attempt to model the travel times of road segments as time-varying functions, which can be extracted from historical traffic patterns. These functions may capture the effects of periodic events (e.g., rush hours, weekdays). Nevertheless, they still cannot reflect live traffic information, which can be affected by sudden events, e.g., congestions, accidents and road maintenance.

Landmark [82, 83, 103] and distance oracle [109] can be applied to estimate

shortest path distance bounds between two nodes in a road network, which can be used to prune irrelevant objects and early detect results. The above works are inapplicable to our problem because they consider constant travel times on road segments (as opposed to live traffic). Furthermore, in this chapter, we propose novel lower/upper travel time bounds derived from both the road network and the information of previously obtained routes; these bounds have not been studied before.

4.1.2 Querying on Online Route APIs

Online route APIs. An online route API [6, 16] has access to current traffic information [15]. It takes a route request as input and then returns a route along with travel times on route segments. The example below illustrates the request and response format of Google Directions API [16]. Bing Maps API [6] uses a similar format.

Table 4.1. Example Google Directions API

HTTP request

```
http://maps.googleapis.com/maps/api/directions/xml?
origin=44.94033,-93.22294&destination=44.94198,-93.23722
mode=driving
```

XML response

```
<step>
   <start_location>
      <lat>44.9403300</lat> <lng>-93.2229400</lng>
   </start_location>
   <end_location>
      <lat>44.9395900</lat> <lng>-93.2229500</lng>
   </end_location>
   <duration> <value>8</value> </duration>
</step>
      ...... remaining steps ......
```

The request is an HTTP query string, whose parameters contain the origin and destination locations in latitude-longitude, as well as the travel mode. In this example, the origin is at $(44.94033, -93.22294)$, the destination is at

(44.94198, −93.23722), and the user is at 'driving' mode.

The response is an XML document that stores a sequence of route segments from the origin to the destination. Each segment, enclosed by `<step>` tags, contains its endpoints and its travel time by driving (see the `<duration>` tags). The segment in this example takes 8 seconds to travel. We omit the remaining segments here for brevity. Besides, the XML response contains the total travel time on this route (the sum of travel times on all segments).

Query processing algorithms. Thomsen et al. [121] study the caching of shortest paths obtained from online route APIs. They exploit the optimal subpath property [52] on cached paths to answer shortest path queries. As we discussed in the introduction and verified in experiments, this property cannot significantly reduce the number of route requests in our problem. Also, they have not studied the processing of range/KNN queries, the lower/upper bound techniques developed in this chapter, as well as the accuracy of query results.

The framework SMashQ [131, 132] is the closest work to our problem. It enables the LBS to process KNN queries by using online route APIs. To reduce the number of route requests (for processing queries), SMashQ exploits the maximum driving speed $\mathbb{V}_{MAX}$ and the static road network G_S (with only distance information) stored at the LBS. Upon receiving a KNN query from user q, the LBS first retrieves K objects with the smallest network distance from q and issues route requests for them. Let γ be the Kth smallest current travel time (obtained so far). The LBS inserts into a candidate set C the objects whose network distance to q is within $\gamma \cdot \mathbb{V}_{MAX}$. Next, SMashQ groups the points in C to road junctions, utilizes historical statistics to order the road junctions, and

then issues route requests for junctions in above order. Compared with our work, SMashQ does not utilize route log to derive exact travel times nor lower/upper bounds to boost the query performance of the LBS. As we will show in the experiments, even if we extend SMashQ to use a route log and apply the optimal subpath property [52] [121] to save route requests, it still incurs much more route requests than our proposed method.

Efficient algorithms [81, 111] have been developed for KNN search on data objects with respect to generic distance functions. It is expensive to compute the exact distance from a query object q to a data object p (e.g., using exact spatial object geometry). On the other hand, it is cheap to compute the lower/upper bound distance from q to p (e.g., using bounding rectangle). Seidl et al. [111] propose a KNN search algorithm that fetches the optimal number of objects from the dataset $\mathcal{P}$. These generic solutions [81, 111] are applicable to our problem; however, they do not exploit the rich information of routes that are specific in our problem. In our problem, the exact route from q to p reveals not only the current travel time to p, it may also provide the current travel times to other objects p' on the route, and may even offer tightened lower/upper bounds of travel times to other objects, as we will illustrate in Section 4.3.

4.2 Problem Statement

In this section, we first describe the system architecture and then formulate the objectives of our problem.

System architecture and notations. In this chapter, we adopt the system

architecture as depicted in Fig. 4.3. It consists of the following entities:

– **Online Route API**. Examples are: Google / Bing route APIs [16] [6]. Such API computes the shortest route between two points on a road network, based on live traffic [15]. It has the latest road network G with live travel time information.

– **Mobile User**. Using a mobile device (smartphone), the user can acquire his current geo-location q and then issue queries to a location-based server. In this chapter, we consider range and KNN queries based on live traffic.

– **Location-Based Service/Server (LBS)**. It provides mobile users with query services on a dataset $\mathcal{P}$, whose POIs (e.g., restaurants, cafes) are specific to the LBS's application. The LBS may store a road network G with edge weights as spatial distances, however G cannot provide live travel times. In case $\mathcal{P}$ and G do not fit in main memory, the LBS may store $\mathcal{P}$ as an R-tree and store the G as a disk-based adjacency list [101].

We then define route, travel time, and queries formally.

Definition 1 (Route and travel time). *The route $\psi_t(v_s, v_d)$ between v_s and v_d, obtained from route API at timestamp t, is a sequence of pairs $\{ \langle v_i, \tau_t(v_s, v_i)\rangle : v_i \in \psi_t \}$. Each pair stores a node v_i and its travel time $\tau_t(v_s, v_i)$ from the source v_s. Let $\tau_t(v, v')$ be the (shortest) travel time between two locations v and v' (obtained at timestamp t).*

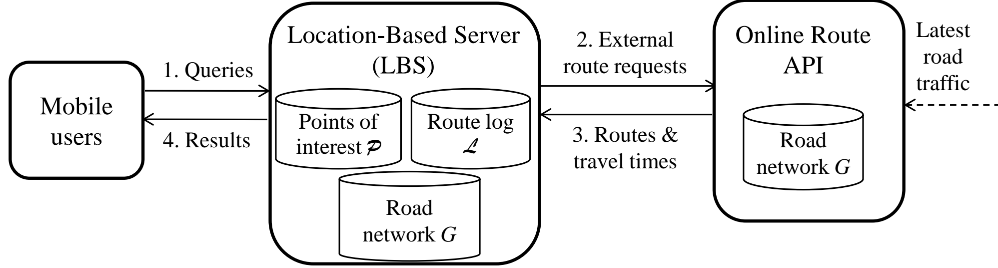


Figure 4.3. System architecture

Definition 2 (Query results).

Let q be a query point and t_{now} be the current time.

Given q and a travel time limit T, the result set of range query *is:* $R = \{\ p \in \mathcal{P} : \tau_{t_{now}}(q, p) \leq T\ \}$.

Given q and a result size K, the result set of KNN query *is:* $R = \{\ p \in \mathcal{P} : \tau_{t_{now}}(q, p) \leq \tau_{t_{now}}(q, p'), p' \in \mathcal{P} - R\ \}$ *with size K.*

As discussed in the introduction, queries in real applications may involve filters on (i) non-spatial features (e.g., quality, price) of $\mathcal{P}$ as well as (ii) live travel times from the query point q to POIs in $\mathcal{P}$. These queries cannot be solved by the LBS alone nor an online map (e.g., Google Map) alone. LBS lacks access to live traffic information (i.e. travel times), whereas the dataset $\mathcal{P}$ maintained by the LBS is not available to an online map.

The flow of the system is as follows. A user first issues a query to the LBS via his/her mobile client (Step 1). The LBS then determines the necessary route requests for the query and submits them to the route API (Step 2). Next, the route API returns the corresponding routes back to the LBS (Step 3). Having such information, the LBS can compute the query results and report them back to the user (Step 4). As a remark, our system architecture is similar to [131],

except our LBS maintains a route log $\mathcal{L}$ and some additional attributes with edges on G (to be elaborated soon).

Objective and our approach. Our objective is to reduce the response time of queries (i.e., requirement $\mathcal{R}2$) while offering accurate query results (i.e., requirement $\mathcal{R}1$). It is important to minimize the number of route requests issued by the LBS because route requests incur considerable time (see introduction).

As observed in Fig. 4.2, travel times change slightly within a short duration (e.g., 10 minutes). Based on this observation, we approximate the travel time (from v to v') at current time t_{now} as the travel time obtained from a route API at an earlier time t':

ASSUMPTION 1 (Temporal approximation). *For any locations v, v', we have: $\tau_{t_{now}}(v, v') \approx \tau_{t'}(v, v')$ if $t' \geq t_{now} - \delta$.*

This approximation enables the LBS to save route requests significantly, while still providing high accuracy. Specifically, at the LBS, we employ a log $\mathcal{L}$ of routes that were requested from an online route API within the last δ minutes.

Like in [131] [132], we assume that the road network G used in LBS is the same with that used in route service. This is feasible when the LBS can obtain accurate maps from the government [23], route service providers [18] or their map suppliers [5]. However, when the LBS cannot have access to the same G as the route service, we will discuss the applicability of our techniques in Section 4.3.5.

To achieve low response time, we will exploit the route log and road network G to reduce the number of external route requests (issued to online route API) for answering queries (Section 4.3). We will also parallelize route requests

(Section 4.4) to further reduce the response time.

4.3 Query Processing

This section presents our approach Route-Saver for answering queries efficiently. First, we discuss the maintenance of the time-tagged road network G and the route log $\mathcal{L}$ (Section 4.3.1). Then, we exploit G and $\mathcal{L}$ to design effective bounds for travel times (Section 4.3.2). Next, we present our algorithms for answering range and KNN queries in Sections 4.3.3, 4.3.4 respectively. Finally, we discuss the applicability of our techniques when no local maps are available in Section 4.3.5.

In subsequent discussion, we drop the subscript t in $\tau_t(v, v')$ as we only use valid routes (and their travel times).

4.3.1 Maintenance of Structures at LBS

Conservative travel time bounds. Given an edge $e(v, v')$, we define $c\omega^-(e)$ and $c\omega^+(e)$ as *conservative* lower-bound and upper-bound of travel time on e, respectively. Observe that the lower-bound $c\omega^-(e)$ is limited by the Euclidean distance of e and the maximum driving speed $\mathbb{V}_{MAX}$:

$$c\omega^-(e) = dist(e)/\mathbb{V}_{MAX} \tag{4.1}$$

On the other hand, the upper-bound is $c\omega^{+}(e) = \infty$ because the travel time on e can be arbitrarily long in case of traffic congestion.

Structures. We employ a route log $\mathcal{L}$ and a time-tagged network G in the LBS.

The *route log* $\mathcal{L}$ stores all routes obtained from an online route API within the last δ time units, as described in Section 4.2. Recall from Definition 1 that the timestamp of a route $\psi_t(v, v')$ is indicated by its subscript t. Assume that we use $\delta = 2$ in Fig. 4.4a. At time $t_{now} = 4$, $\mathcal{L}$ keeps the routes obtained during time 2–4.

To support query operations efficiently, we summarize the travel times of edges in $\mathcal{L}$ into a time-tagged network G. Specifically, each edge e in G is tagged with a tuple $(c\omega^{-}(e), \omega(e))_{\mu(e)}$, where $c\omega^{-}(e)$ is the conservative lower-bound travel time on e (Eqn. 4.1), $\omega(e)$ is the exact travel time stored in $\mathcal{L}$, and $\mu(e)$ is the last-update timestamp for $\omega(e)$. We call an edge e to be *valid* if its last-update timestamp $\mu(e)$ satisfies $\mu(e) \geq t_{now} - \delta$.

As an example, consider the time-tagged network G at current time $t_{now} = 4$ in Fig. 4.4b. Assume that the expiry time is $\delta = 2$. We draw valid edges by solid lines and invalid edges by dotted lines. For the solid edge (v_3, v_6), the tuple $(25, 42)_3$ means that its conservative lower bound $c\omega^{-}(v_3, v_6)$ is 25, its exact travel time $\omega(v_3, v_6)$ is 42, and its last-update timestamp $\mu(v_3, v_6)$ is 3. The dotted edge (v_2, v_3) is invalid since its timestamp $\mu(v_2, v_3) = 1$ is less than $t_{now} - \delta = 4 - 2 = 2$.

Maintenance. We then discuss how to maintain the route log $\mathcal{L}$ and the time-tagged road network G.

To support efficient lookup on $\mathcal{L}$, we employ inverted lists of routes for each node [121]. Specifically, each inverted list of node v stores a list of route IDs that contain v. The insertion/deletion of a route can be implemented to take $O(|\psi|)$ time, where $|\psi|$ is the number of vertices on a route.

At time t_{now}, we remove from $\mathcal{L}$ the routes ψ_t having $t < t_{now} - \delta$ (i.e., expired). E.g., at $t_{now} = 5$, we remove $\psi_2(v_5, v_6)$ from $\mathcal{L}$ (see Fig. 4.4a). Also, we update the inverted lists for $v_4, v_5, v_6 \in \psi_2(v_5, v_6)$. We need not update G now because it stores the last-update timestamps of edges.

When we retrieve a route $\psi_{t_{now}}$ from online route API, e.g., $\psi_5(v_1, v_7)$: $v_1 \to v_8 \to v_7$, we insert it into $\mathcal{L}$ (see Fig. 4.4a), and update the inverted lists for nodes v_1, v_7, v_8. For the edges on $\psi_5(v_1, v_7)$, e.g., (v_1, v_8), (v_8, v_7), we update their $\omega(e)$ and $\mu(e)$ in G (see Fig. 4.4c).

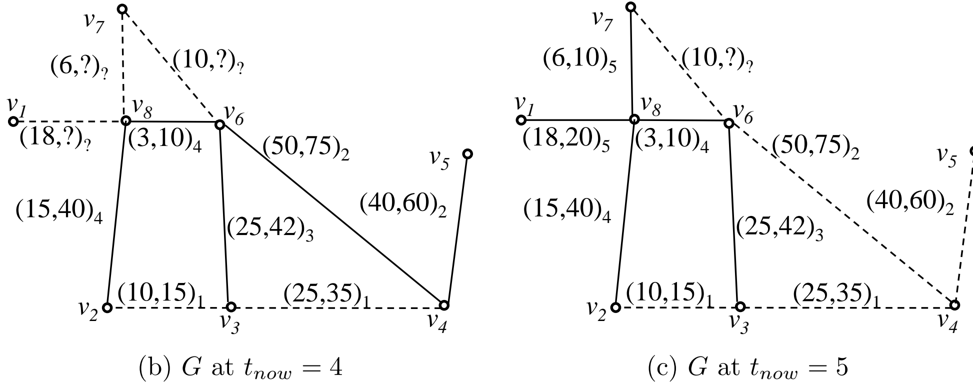
4.3.2 Exact Travel Times and Their Bounds

In this section, we exploit the time-tagged road network G and the route log $\mathcal{L}$ to derive lower and upper bounds of travel times for data points. As we will elaborate soon, these bounds enable us to save route requests during query processing.

Before presenting these techniques, we first show an example of data stored at LBS (see Fig. 4.5). Besides G and $\mathcal{L}$, the LBS also stores a dataset $\mathcal{P}$ (points p_j with locations). Assume the current time $t_{now} = 9$ and the expiry time $\delta = 5$. The route log $\mathcal{L}$ contains only valid routes (not yet expired). For the time-tagged network G (see Fig. 4.5c), solid edges are valid while dotted edges are not. Each edge e is tagged with $c\omega^-(e)$ and $\omega(e)$ (underlined), and the icons of routes via

Route ID	Route Content	t_{now}=4	t_{now}=5	t_{now}=6
$\psi_1(v_2, v_4)$	$(v_2, 0)$, $(v_3, 15)$, $(v_4, 50)$			
$\psi_2(v_5, v_6)$	$(v_5, 0)$, $(v_4, 60)$, $(v_6, 135)$	√		
$\psi_3(v_3, v_6)$	$(v_3, 0)$, $(v_6, 42)$	√	√	
$\psi_4(v_2, v_6)$	$(v_2, 0)$, $(v_8, 40)$, $(v_6, 50)$	√	√	√
$\psi_5(v_1, v_7)$	$(v_1, 0)$, $(v_8, 20)$, $(v_7, 30)$		√	√
$\psi_6(v_2, v_3)$	$(v_2, 0)$, $(v_3, 15)$			√

(a) Route Log $\mathcal{L}$ (at different times)



(b) G at $t_{now} = 4$ (c) G at $t_{now} = 5$

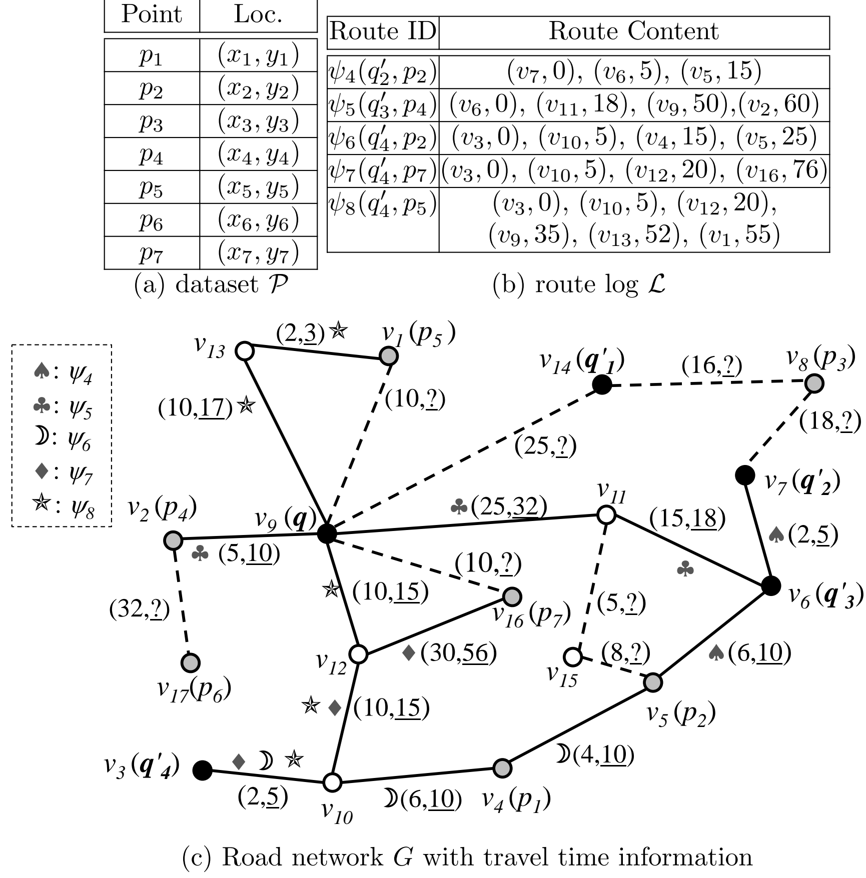
Figure 4.4. Example route log $\mathcal{L}$ and time-tagged road network G, with expiry time $\delta = 2$; solid edges have valid travel times

e (if any). For clarity, we omit $\mu(e)$ (i.e., the last-update timestamp) of edges.

We first introduce the concept of travel time bounds:

DEFINITION 3 (Travel time bounds). *Given a query point q and a data point p, we denote $p.\tau^-$ and $p.\tau^+$ as a lower bound and an upper bound of the exact travel time $\tau(q,p)$. Specifically, we require that $p.\tau^- \leq \tau(q,p) \leq p.\tau^+$. For convenience, we may denote $\tau(q,p)$ by $p.\tau$.*

As an example, consider a data point p and a range query with travel time limit T. The upper-bound time $p.\tau^+$ helps detect true results early. If p satisfies $p.\tau^+ \leq T$, then p must be a result. The lower-bound time $p.\tau^-$ enables pruning unpromising points. If p satisfies $p.\tau^- > T$, then p cannot be a result. In either



Point	Loc.
p_1	(x_1, y_1)
p_2	(x_2, y_2)
p_3	(x_3, y_3)
p_4	(x_4, y_4)
p_5	(x_5, y_5)
p_6	(x_6, y_6)
p_7	(x_7, y_7)

(a) dataset $\mathcal{P}$

Route ID	Route Content
$\psi_4(q_2', p_2)$	$(v_7, 0)$, $(v_6, 5)$, $(v_5, 15)$
$\psi_5(q_3', p_4)$	$(v_6, 0)$, $(v_{11}, 18)$, $(v_9, 50)$,$(v_2, 60)$
$\psi_6(q_4', p_2)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_4, 15)$, $(v_5, 25)$
$\psi_7(q_4', p_7)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_{12}, 20)$, $(v_{16}, 76)$
$\psi_8(q_4', p_5)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_{12}, 20)$, $(v_9, 35)$, $(v_{13}, 52)$, $(v_1, 55)$

(b) route log $\mathcal{L}$

(c) Road network G with travel time information

Figure 4.5. Data stored at the LBS, at $t_{now} = 9$ ($\delta = 5$); each edge is tagged with ($c\omega^-(e)$, $\underline{\omega(e)}$)

case, we save a route request for p. Observe that a tight upper bound should be as small as possible because it is more likely to satisfy $p.\tau^+ \leq T$. Similarly, a tighter lower bound should be as large as possible to satisfy $p.\tau^- > T$. Techniques discussed below aim to derive tight bounding travel times for data points.

Conservative lower-bound. Let $spt_{c\omega^-}(q, p)$ be the shortest travel time from q to p defined on the edge weight $c\omega^-(e)$ (see Eqn.4.1). The conservative lower-

bound travel time from q to p is:

$$p.\tau_c^- = spt_{c\omega^-}(q,p) \tag{4.2}$$

We take the point p_3 in Fig. 4.5c as an example. With respect to the weight $c\omega^-(e)$, the shortest path from q to p_3 is $q \to v_{14} \to p_3$, with length $c\omega^-(q,v_{14}) + c\omega^-(v_{14},p_3) = 41$. Table 4.2 shows $p.\tau_c^-$ for each data point p.

Bounding travel times based on ω^+ and ω^-. Recall that the exact travel time $\tau(q,p)$ is defined as the shortest travel time based on live traffic information. Thus, we have: $\tau(q,p) = spt_{\omega^*}(q,p)$, where $\omega^*(e)$ denotes the current travel time for an edge e. We use the notations $\tau(q,p)$ and $spt_{\omega^*}(q,p)$ interchangeably in the following discussion.

Our idea is to define upper-bound weight $\omega^+(e)$ and lower-bound weight $\omega^-(e)$ for each edge e, by using the information in the time-tagged road network G.

$$\omega^+(e) = \begin{cases} \omega(e) & \text{if } \mu(e) \geq t_{now} - \delta \\ \infty & \text{otherwise} \end{cases} \tag{4.3}$$

$$\omega^-(e) = \begin{cases} \omega(e) & \text{if } \mu(e) \geq t_{now} - \delta \\ c\omega^-(e) & \text{otherwise} \end{cases} \tag{4.4}$$

Note that $\omega^*(e)$ is unknown to the LBS in general. If $\mu(e) \geq t_{now} - \delta$, then the last-update travel time $\omega(e)$ (in above equations) serves as an approximation of $\omega^*(e)$, due to Assumption 1.

Table 4.2. Example travel time information (for user q), $t_{now} = 9$

Point	$p.\tau_c^-$	$p.\tau_G^-$	$p.\tau_G$	$p.\tau_G^+$	$p.\tau_I^-$
p_1 (v_4)	26	40	40	40	20
p_2 (v_5)	30	45	NIL	50	40
p_3 (v_8)	41	41	NIL	NIL	NIL
p_4 (v_2)	5	10	10	10	NIL
p_5 (v_1)	10	10	NIL	20	NIL
p_6 (v_{17})	37	42	NIL	NIL	NIL
p_7 (v_{16})	10	10	NIL	71	41

With these edge weights, we establish the upper and lower bounds for travel times from q to p (Lemma 1).

LEMMA 1 (bounding travel times on ω^+, ω^-).
Let $spt_{\omega^+}(q,p)$ be the shortest travel time from q to p with respect to the edge weight $\omega^+(e)$, on a time-tagged road network G.
Similarly, let $spt_{\omega^-}(q,p)$ be the shortest travel time with respect to the edge weight $\omega^-(e)$. With Assumption 1, we have: $spt_{\omega^-}(q,p) \leq spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$.

Proof. We first aim to prove: $spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$. Let SP^* and SP^+ be the shortest path between q and p defined on the edge weights $\omega^*(e)$ and $\omega^+(e)$, respectively. According to Eqn. 4.3, we have $\omega(e) \leq \omega^+(e)$. By Assumption 1, we approximate $\omega^*(e)$ by $\omega(e)$. Thus, we have $\omega^*(e) \leq \omega^+(e)$. Applying it on all edges on SP^+, we obtain: $\sum_{e \in SP^+} \omega^*(e) \leq \sum_{e \in SP^+} \omega^+(e)$ —(▲). By the definition of shortest path on the edge weight $\omega^*(e)$, the travel time of SP^* is no larger than that of SP^+. Thus, we have: $\sum_{e \in SP^*} \omega^*(e) \leq \sum_{e \in SP^+} \omega^*(e)$ —(▼). Combining inequalities (▲) and (▼), we obtain: $\sum_{e \in SP^*} \omega^*(e) \leq \sum_{e \in SP^+} \omega^+(e)$. Therefore, $spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$.

The proof for $spt_{\omega^-}(q,p) \leq spt_{\omega^*}(q,p)$ is similar to the above proof, except that we apply Eqn. 4.4 instead. □

In subsequent discussion, we represent the bounds $spt_{\omega^+}(q,p)$ and $spt_{\omega^-}(q,p)$ by $p.\tau_G^+$ and $p.\tau_G^-$ respectively. Observe that we can compute $p.\tau_G^+$ (or $p.\tau_G^-$) for all points efficiently by running the Dijkstra algorithm using edge weight $\omega^+(e)$ (or $\omega^-(e)$).

Table 4.2 shows the upper-bound $p.\tau_G^+$ and lower-bound $p.\tau_G^-$ for all data points. We take the candidate p_2 in Fig. 4.5c as an example. After running Dijkstra on G using $\omega^-(e)$, the shortest path from q to p_2 is $q \to v_{11} \to v_{15} \to p_2$ with the length as $\omega(q, v_{11}) + c\omega^-(v_{11}, v_{15}) + c\omega^-(v_{15}, p_2) = 45$. After running Dijkstra on G using $\omega^+(e)$, the shortest path from q to p_2 as $q \to v_{12} \to v_{10} \to v_4 \to p_2$ with length as $\omega(q, v_{12}) + \omega(v_{12}, v_{10}) + \omega(v_{10}, v_4) + \omega(v_4, p_2) = 50$.

Condition for exact travel time. When a point p satisfies certain condition (see Lemma 2), its lower-bound travel time ($p.\tau_G^-$) serves as its exact travel time from q (denoted by $p.\tau_G$). In this case, we save a route request for p regardless of the value of $p.\tau_G$.

LEMMA 2 (Road network exact travel time).
Let SP^- (with travel time $spt_{\omega^-}(q,p)$) be the shortest path from q to p with respect to the edge weight $\omega^-(e)$, on a time-tagged road network G. If each edge on SP^- satisfies $\mu(e) \geq t_{now} - \delta$, then we have: $spt_{\omega*}(q,p) = spt_{\omega^-}(q,p)$.

Proof. If an edge e satisfies $\mu(e) \geq t_{now} - \delta$ (i.e., valid edge), then we have: $\omega^-(e) = \omega^+(e) = \omega(e) \quad (= \omega^*(e))$. Applying this to each edge on SP^-, we obtain: $spt_{\omega^-}(q,p) = \sum_{e \in SP^-} \omega^-(e) = \sum_{e \in SP^-} \omega^*(e)$.

We then claim that SP^- is the shortest path with respect to the edge weight ω^*. For the sake of contradiction, assume there exists a path SP' shorter than

SP^- on edge weight ω^*. Then, SP' has a shorter travel time than $spt_{\omega^-}(q,p)$, contradicting the fact that $spt_{\omega^-}(q,p)$ is a lower-bound. Thus, this lemma is proved. □

Table 4.2 lists the exact travel time $p.\tau_G$ of all data points. Take the candidate p_1 in Fig. 4.5c as an example. With respect to edge weight $\omega^-(e)$, the shortest path from q to p_1 is $q \to v_{12} \to v_{10} \to p_1$. Since each edge on this path has valid exact travel time, thus we obtain: $p_1.\tau_G = \omega(q, v_{12}) + \omega(v_{12}, v_{10}) + \omega(v_{10}, v_4) = 40$.

Tightening the lower-bound using route log. As we will illustrate soon, the lower-bound $p.\tau_G^-$ (derived from edge weight $\omega^-(e)$) may not be tight for some data points.

Next, we utilize a shared node $\imath$ among routes in $\mathcal{L}$ to derive another lower-bound travel time for candidates (see Lemma 3). We denote this lower-bound travel time as $p.\tau_I^- = |\tau(q,\imath) - \tau(p,\imath)|$.

LEMMA 3 (Route log lower-bound travel time). *Let ψ_i, ψ_j be two different routes in the route log $\mathcal{L}$ such that they share a node $\imath$. If q, p fall on ψ_i, ψ_j respectively, then we have: $|\tau(q,\imath) - \tau(p,\imath)| \leq \tau(q,p)$ (i.e. $spt_{\omega^*}(q,p)$).*

Proof. For the sake of contradiction, assume that: $\tau(q,p) < |\tau(q,\imath) - \tau(p,\imath)|$. For case I ($\tau(q,\imath) \geq \tau(p,\imath)$), we obtain: $\tau(q,p) + \tau(p,\imath) < \tau(q,\imath)$. This contradicts with that $\tau(q,\imath)$ is the shortest between q and $\imath$. For case II ($\tau(q,\imath) < \tau(p,\imath)$), we obtain: $\tau(p,q) + \tau(q,\imath) < \tau(p,\imath)$. This contradicts with that $\tau(p,\imath)$ is the shortest between p and $\imath$. Thus, the lemma is proved. □

Since ψ_i, ψ_j are routes in $\mathcal{L}$, so the values $\tau(q, \imath), \tau(p, \imath)$ can be directly obtained from $\mathcal{L}$ according to *optimal subpath property* [52]. Through using the inverted node index [121] of $\mathcal{L}$, we can efficiently retrieve the subset of routes which contains q (i.e. L_q) and the corresponding subset for each remaining candidate point p (i.e. L_p). Then, we can identify the shared node between routes in L_q and L_p, and use it to calculate $p.\tau_I^-$.

Continuing with the example, we show how to derive $p_7.\tau_I^-$ of point p_7 (in Fig. 4.5c). First, we find the subset of routes that contain q, i.e., $L_q = \{\psi_5, \psi_8\}$. Then, we find the subset of routes that contain p_7, i.e., $L_{p_7} = \{\psi_7\}$. Next, we identify a shared node between L_q and L_{p_7}, which is the node v_{12} on the routes $\psi_8 \in L_q$ and $\psi_7 \in L_{p_7}$. With these routes, we obtain these exact travel times: $\tau(q, v_{12}) = 15$ and $\tau(p_7, v_{12}) = 56$. Thus, we derive $p_7.\tau_I^- = |\tau(p_7, v_{12}) - \tau(q, v_{12})| = |56 - 15| = 41$. Observe that this bound $p_7.\tau_I^- = 41$ is tighter than the bound $p_7.\tau_G^- = 10$.

Nevertheless, the bound $p.\tau_I^-$ can be looser or unavailable for some points, e.g., p_1, p_3 in Table 4.2. So, we combine bounds τ_G^- and $p.\tau_I^-$ into a tighter lower bound for p:

$$p.\tau^- = \max\{p.\tau_G^-, p.\tau_I^-\} \tag{4.5}$$

4.3.3 Range Query Algorithm

In this section, we present our Route-Saver algorithm for processing a range query (q, T). It applies the travel time bounds discussed above to reduce the number of route requests. To guarantee the accuracy of returned results, it removes all expired routes ψ_t in $\mathcal{L}$. The algorithm first conducts a distance

range search $(q, T \cdot \mathbb{V}_{MAX})$ for $\mathcal{P}$ on G [101] to obtain a set C of candidate points. Algorithm 4.1 consists of two phases to process the candidate points in C and store the query results in the set R.

The first phase (Lines 4–17) aims to shrink the candidate set C, so as to reduce the number of route requests to be issued in the second phase. First, we execute Dijkstra on G two times, using edge weight $\omega^-(e)$ and $\omega^+(e)$ respectively. Then, we obtain the bounds $p.\tau_G^+$, $p.\tau_G^-$ and $p.\tau_G$ for every candidate $p \in C$. If $p.\tau_G^+ \leq T$ or $p.\tau_G \leq T$, then p must be a true result so we place it into R. If $p.\tau_G^- > T$, then p cannot become a result and it gets removed from C. Next, for each candidate p remaining in C, we compute its exact travel time $p.\tau_{\mathcal{L}}$ using optimal subpath property in $\mathcal{L}$ [52] [121], and use $p.\tau_{\mathcal{L}}$ to detect true result. Moreover, we derive the lower bound travel time $p.\tau_I^-$ using route log $\mathcal{L}$ for pruning.

In the second phase, we issue route requests for the remaining candidates in C, based on a certain ordering. We will elaborate the effect of candidate ordering at the end of this section. For the moment, suppose that we examine candidates in ascending order, i.e., pick a candidate $p \in C$ with the minimum $p.\tau^-$ (Line 19). Next, we issue a route request for p and then insert the returned route $\psi_{t_{now}}$ into the route log $\mathcal{L}$. For each edge on the returned route $\psi_{t_{now}}$, we update its $\omega(e)$ and $\mu(e)$ accordingly.

This route provides not only the exact travel time for p, but also potential information for updating the bounds for other candidate $p' \in C$. We remove p' from C if (i) it cannot become result, i.e., $p'.\tau^- > T$, or (ii) its exact travel time $p'.\tau_{\mathcal{L}}$ is known (i.e., p' lies on route $\psi_{t_{now}}$). In case $p'.\tau_{\mathcal{L}} \leq T$, we insert p' into

Algorithm 4.1 Route-Saver Algorithm for Range Queries

```
    function Route-Saver-Range ( Query (q,T), Dataset P )
                  ▷ system parameters: time-tagged graph G, route log L, expiry time δ
 1: Remove from the log L any route ψ_t with t < t_now − δ
 2: Create a result set R ← ∅
 3: Cand. set C ← range search (q, T · V_MAX) for P on G                    ▷ By [101]
 4: Run Dijkstra for q on G using ω+(e) and ω−(e) to retrieve p.τ+_G, p.τ−_G, p.τ_G  ▷
    Phase 1:detect results,prune objects
 5: for each p ∈ C do                                           ▷ use time-tagged graph G
 6:     if p.τ_G is known or p.τ−_G > T or p.τ+_G ≤ T then
 7:         Remove p from C
 8:     if p.τ_G ≤ T or p.τ+_G ≤ T then
 9:         Insert p into R
10: for each p ∈ C do                                                  ▷ use route log L
11:     if ∃ route ψ ∈ L such that ψ contains p and q then
12:         Compute p.τ_L                          ▷ optimal subpath property [52] [121]
13:         if p.τ_L is known and p.τ_L ≤ T then
14:             Insert p into R
15:     Compute p.τ− as max{p.τ−_G, p.τ−_I}
16:     if p.τ− > T or p.τ_L is known then
17:         Remove p from C
18: while C is not empty do                             ▷ Phase 2: Issue route requests
19:     Pick an object p ∈ C with minimum p.τ−                               ▷ ordering
20:     Route ψ_{t_now} ← RouteRequest(q, p)                         ▷ call external API
21:     Insert ψ_{t_now} into L;    Update ω(e), μ(e) in G for e ∈ ψ_{t_now}
22:     Update p.τ_L for all p on ψ_{t_now}        ▷ optimal subpath property [52] [121]
23:     Run incremental Dijkstra to update all p.τ−                           ▷ By [44]
24:     for each p′ ∈ C do
25:         if p′.τ− > T or p′.τ_L is known then
26:             Remove p′ from C
27:         if p′.τ_L ≤ T then
28:             Insert p′ into R
29: Return R
```

R. Whenever C becomes empty, the loop terminates and the algorithm reports R as the result set.

Example. Consider the range query at q with $T = 40$ in Fig. 4.5c. We illustrate the running steps of Route-Saver in Table 4.3. Entries without values are labeled as '/'.

Suppose that $\mathbb{V}_{MAX} = 110\ km/h$. First, we do a range search at q with distance $T \cdot \mathbb{V}_{MAX}$, and obtain the candidate set $C = p_1, p_2, p_4, p_5, p_6, p_7$. Note that further away points (e.g. p_3) are not in C. Then, we derive the $p.\tau_G^+, p.\tau_G^-$ and $p.\tau_G$ using the time-tagged road network G, as shown in the first three columns of Table 4.3. Candidates p_1, p_4, p_5 are inserted into the result set R, since their exact or upper-bound travel times are smaller than $T = 40$. Candidates p_2, p_6 are pruned with lower bounds larger than $T = 40$. Then, we compute the lower bound for the remaining candidate using $\mathcal{L}$: $p_7.\tau_I^- = 41$, and p_7 is pruned. We skip the second phase as the candidate set becomes empty. Thus, the algorithm returns $R = \{p_1, p_4, p_5\}$ to the user. In this example, Route-Saver issues 0 route request.

Table 4.3. Range/KNN query example for Route-Saver, $T = 40$

	$p.\tau_G^-$	$p.\tau_G$	$p.\tau_G^+$	$p.\tau_\mathcal{L}$	$p.\tau_I^-$	$p.\tau$ by route API	Is result?
p_1	40	40	40	/	/	/	√
p_2	45	/	50	/	/	/	×
p_3	41	/	/	/	/	/	×
p_4	10	10	10	/	/	/	√
p_5	10	/	20	/	/	/	√
p_6	42	/	/	/	/	/	×
p_7	10	/	71	/	41	/	×

Candidate ordering and its analysis. This section studies the effect of candidate orderings on the cost of Algorithm 4.1, i.e., the number of route requests issued. Various orderings can be used for processing the candidates (in phase 2). We consider two orderings for picking the next candidate $p \in C$ (at Line 19):

Ascending order (ASC): Pick a candidate with the minimum $p.\tau^-$. This order is the same as in Algorithm 4.1.

Descending order (DESC): Pick a candidate with the maximum $p.\tau^-$. The

rationale is that longer routes are more likely to cover other candidates and thus save route requests for them.

We proceed to analyze the number of route requests incurred by ASC and DESC.

For simplicity, we assume that the underlying graph is a unit-weight grid network (in 2D space). Let q be at the origin $(0, 0)$ and T be the travel time limit. Let α be the data density, i.e., the probability that a node contains a point. Let the layer i be the set of nodes whose travel times from q equal to i. Observe that, in layer i, there are $4i$ nodes and $4\alpha i$ candidates. Summing up this from layer 1 to layer T, the number of candidates is: $Cand(\alpha, T) = \sum_{i \in [1,T]} 4\alpha i \approx 2\alpha T^2$

ASC issues route requests for candidates in ascending order of their layers. Thus, it cannot save any route request for candidates. The cost of ASC is:

$$Cost_{ASC}(\alpha, T) = Cand(\alpha, T) \approx 2\alpha T^2$$

On the other hand, DESC issues route requests for candidates in descending order of their layers. Consider a node v in the layer i. Note that the number of candidates from layer $i+1$ to layer T is: $2\alpha T^2 - 2\alpha i^2 = 2\alpha(T^2 - i^2)$. If the route from q to any of these candidates passes v, then we can save a route request for v. Since there are $4 \cdot i$ possible locations for v, the probability of saving a route request for v is: $\max\{\frac{2\alpha(T^2-i^2)}{4i}, 1\} = \max\{\frac{\alpha(T^2-i^2)}{2i}, 1\}$. Thus, the cost of DESC is: $Cost_{DESC}(\alpha, T) = \sum_{i \in [1,T]} 4\alpha i \cdot (1 - \max\{\frac{\alpha(T^2-i^2)}{2i}, 1\})$ To simplify the above equation, we find the maximum value for i such that: $\frac{\alpha(T^2-i^2)}{2i} \geq 1$. By solving this quadratic inequality, we get: $i \leq \frac{\sqrt{1+1\alpha^2T^2}-1}{\alpha}$. When $T > \frac{1}{\alpha}$, the cost of

DESC is upper-bounded by:

$$Cost_{DESC}(\alpha, T) \leq 4T$$

In summary, DESC incurs a much lower cost than ASC.

4.3.4 KNN Query Algorithm

In this section, we extend our Route-Saver algorithm for processing KNN queries. We will also examine suitable orderings for processing candidates.

Unlike range queries, KNN queries do not have a (fixed) travel time limit T for obtaining a small candidate set. Instead, we first compute a (temporary) result set R so that it contains K candidates with the smallest $p.\tau_G^+$ or $p.\tau_G$. Recall that we can obtain these bounds/values for all candidates efficiently by two Dijkstra traversal on G. Let γ be the largest $p.\tau_G^+$ or $p.\tau_G$ in R. Having this value γ, we can prune each candidate p that satisfies $p.\tau^- > \gamma$, as it cannot become the result.

Algorithm 4.2 is the pseudo-code of our KNN algorithm. First, we initialize the candidate set C with the dataset $\mathcal{P}$, insert K dummy pairs (with ∞ travel time) into the result set R, and set γ to the largest travel time in R. The algorithm consists of three phases. In the first phase, it obtains γ by using the idea discussed above. In the second phase, it prunes candidates whose lower bounds or exact times are larger than γ. In the third phase, it examines the candidates according to a certain order and issues route requests for them. The algorithm terminates when the candidate set contains exactly K objects, and

then reports them as query results.

Algorithm 4.2 Route-Saver Algorithm for KNN Queries

```
    function Route-Saver-KNN ( Query (q, K), Dataset P )
                  ▷ system parameters: time-tagged graph G, route log L, expiry time δ
 1: Remove from the log L any route ψ_t with t < t_now − δ
 2: Create a candidate set C ← P
 3: Create a result set R with K pairs ⟨NULL, ∞⟩
 4: γ ← the largest travel time in R
 5: Run Dijkstra for q on G using ω+(e) and ω−(e) to retrieve p.τ+_G, p.τ−_G, p.τ_G    ▷
    Phase 1: obtain the threshold γ
 6: for each p ∈ C do
 7:     Update R, γ by p with p.τ+_G or p.τ_G
 8: for each p ∈ C do                                         ▷ Phase 2: prune objects
 9:     if p.τ_G > γ or p.τ−_G > γ then
10:         Remove p from C
11:     if ∃ route ψ ∈ L such that ψ contains p and q then
12:         Compute p.τ_L                       ▷ optimal subpath property [52] [121]
13:         Update R, γ by p with p.τ_L
14:     Compute p.τ− as max{p.τ−_G, p.τ−_I}
15:     if p.τ− > γ or (p.τ_L is known and p.τ_L > γ) then
16:         Remove p from C
17: while |C| > K do                                   ▷ Phase 3: Issue route requests
18:     Pick an object p ∈ C with minimum p.τ−                             ▷ ordering
19:     Route ψ_tnow ← RouteRequest(q, p)                          ▷ call external API
20:     Insert ψ_tnow into L;   Update ω(e), μ(e) in G for e ∈ ψ_tnow
21:     Update p.τ_L for all p on ψ_tnow              ▷ optimal subpath property [52]
22:     Run incremental Dijkstra to update all p.τ−                         ▷ By [44]
23:     for each p′ ∈ C do
24:         if p′.τ− > γ or p′.τ_L > γ then
25:             Remove p′ from C
26:         if p′.τ_L < γ then
27:             Update R by p′ with p′.τ_L
28: Return R
```

Example. Consider the KNN query with $K = 3$ in Fig. 4.5c. We illustrate the running steps of Route-Saver in Table 4.3. Entries without values are marked as '/'.

In the first phase, we derive the upper bounds $p.\tau_G^+, p.\tau_G, p.\tau_G^-$ using the

time-tagged road graph G, which are shown in the first three columns in Table 4.3. Since $p_1.\tau_G$, $p_4.\tau_G$ and $p_5.\tau_G^+$ are the smallest three travel times, we insert them into R and update $\gamma = 40$. In the second phase,first we prune candidates p_2, p_5, p_6 since their $p.\tau_G^-$ are larger than γ. Then, we calculate the lower-bound travel time for p_7 using $\mathcal{L}$: $p_7.\tau_I^- = 41 > \gamma$, so p_7 is pruned. We skip the third phase as the candidate set contains exactly $K = 3$ objects, the same as the result set R. Thus, the algorithm returns $R = \{p_1, p_4, p_5\}$ as the query result. Route-Saver issues 0 route request in this example. On the other hand, SMashQ incurs 7 route requests when solving this query (see the method description in Sec. 4.1.2).

Candidate ordering. For the orderings to rank candidates in C (Line 18) in Algorithm 4.2, in addition to the orderings discussed in Section 4.3.3, we propose a new ordering:

Maximum difference (DIFF): Pick a candidate with the maximum $p.\tau^+ - p.\tau^-$. This order tends to tighten the lower and upper bounds of candidates rapidly. A tight $p.\tau^+$ helps refine the value γ whereas a tight $p.\tau^-$ helps prune the candidate itself.

4.3.5 Applicability of Techniques without Map

In this section, we discuss how to adapt the Route-Saver in case the LBS cannot obtain the same map G used in the route service. We observe that, if the LBS uses the map G' (e.g., a free map [21]) which are not the same with that used in route services, bounding travel times $p.\tau_G^-$ can be over-estimated. For example, if the real shortest path from q to p is missing in local map G', then it

is possible that Route-Saver calculates a higher $p.\tau_G^-$ for p and mistakenly prunes it from results. Therefore, the LBS is not allowed to use inaccurate maps.

In case that the LBS cannot access to the map G used in route services, the applicability of our techniques are as follows:

– $p.\tau_G^-$, $p.\tau_G$ and $p.\tau_G^+$ are not applicable because they are calculated based on G, which is not available to the LBS.

– $p.\tau_{\mathcal{L}}$ is applicable, since it is solely calculated using route logs which are obtained from route services.

– $p.\tau_I^-$ is applicable, as it is solely based on route logs.

– $p.\tau_{G_{\mathcal{L}}}^+$ is applicable, where $G_{\mathcal{L}}$ is a road network formed by routes in the log. Observe that $G_{\mathcal{L}}$ must be a subgraph of G.

4.4 Parallelized Route Requests

Our objective (see Section 4.2) is to minimize the response time of queries. Section 4.3 optimizes the response time through reducing the number of route requests. Can we further reduce the response time? In this section, we examine how to parallelize route requests in order to optimize user response time further. We propose two parallelization techniques that achieve different tradeoffs on the number of route requests and user response time.

The execution of algorithms in Section 4.3 follows a sequential schedule like Fig. 4.6a. The user response time consists of: (i) the time spent on route requests (in gray), and (ii) local computation at the LBS (in white).

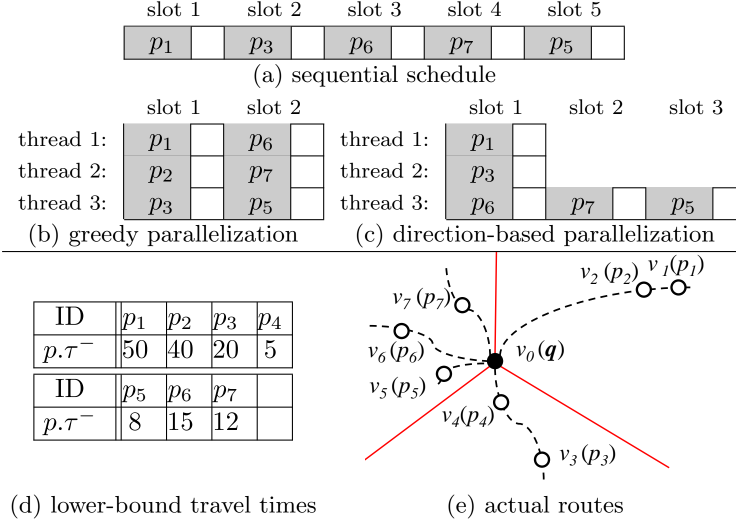
Consider the sequential schedule in Fig. 4.6a. An experiment (see Fig. 4.11) reveals that the user response time is dominated by the time spent on route requests. Let a *slot* be the waiting period to obtain a route from the route API[4.2]. In Fig. 4.6a, the sequential schedule takes 5 slots for 5 route requests. Intuitively, the LBS may reduce the number of slots by issuing multiple route requests to a route API in parallel. Fig. 4.6b illustrates a parallel schedule with 2 slots; each slot contains 3 route requests issued in parallel.

Although parallelization helps reduce the response time, it may prevent sharing among routes and cause extra route requests (e.g., request for route p_2), as we will explain later. Existing parallel scheduling techniques [58] have not exploited this unique feature in our problem. We also want to avoid extra route requests because a route API may impose a daily route request limit [17] or charge the LBS based on route requests [7].

We proceed to present two parallelization techniques. They achieve different tradeoffs on the number of route requests and the number of slots. Our discussion focuses on range queries only. Our techniques can be extended to KNN queries as well.

Greedy parallelization. Let m be the number of threads for parallel execution (per query). Our *greedy parallelization* approach dispatches route request to a thread as soon as it becomes available. Specifically, we modify Algorithm 4.1 as follows. Instead of picking one object p from the candidate set C (at Lines 19–20), we pick m candidate objects and assign their route requests to m threads in parallel. Observe that this approach minimizes the number of time slots in

[4.2] Different route requests incur similar time (see Fig. 4.8).



ID	p_1	p_2	p_3	p_4
$p.\tau^-$	50	40	20	5
ID	p_5	p_6	p_7	
$p.\tau^-$	8	15	12	

Figure 4.6. Effect of parallelization on schedules

the schedule (Fig. 4.6b).

We proceed to compare the sequential schedule with the greedy schedule on the example. Consider a range query at q with $T = 60$. Suppose that the candidate set is $C = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. Fig. 4.6d shows the lower-bound travel time of each object and Fig. 4.6e depicts the locations of all objects. Assume that the routes (dotted lines) are missing from the the route log $\mathcal{L}$ at the LBS. Here, we order the candidates using DESC ordering (see Section 4.3.3), and set the number of threads $m = 3$.

Fig. 4.6a shows a sequential schedule of route requests (issued by the original Algorithm 4.1). By the DESC ordering, the candidates will be examined in the order: $p_1, p_2, p_3, p_6, p_7, p_5, p_4$. First, a route request is issued for p_1. Since the route to p_1 covers p_2, we save a route request for p_2. Similarly, after issuing a route request for p_3, we save a route request for p_4. After that, route requests are issued for the remaining candidates p_6, p_7, p_5. Note that the sequential schedule

(Fig. 4.6a) takes 5 slots.

Fig. 4.6b illustrates a parallel schedule of route requests by using the greedy approach. First, it selects m $(= 3)$ objects in the DESC order: p_1, p_2, p_3. Thus, 3 route requests are issued for them at the same time. Since p_4 lies on the route from q to p_3, the route request for p_4 is saved. After that, 3 route requests are issued for remaining candidates p_5, p_6, p_7 at the same time. In summary, the greedy approach takes only 2 slots, but incurs 6 route requests.

Direction-based parallelization. Observe that the extra route request(s) in the greedy approach is caused by objects at similar directions from q (e.g., p_1, p_2 in Fig. 4.6e). If we issue route requests to candidates in different directions in parallel, then we may avoid extra route requests. This is the intuition behind our *direction-based parallelization* approach.

In this approach, the LBS divides the candidate set C into m groups $(C_1, C_2, \cdots, C_m)$, based on the direction angle $\measuredangle(q, p)$ of each candidate p from user q. A candidate p is inserted into the group C_i if $\frac{(i-1)\cdot 360^o}{m} \leq \measuredangle(q, p) \leq \frac{i \cdot 360^o}{m}$. This step can be implemented just before Line 18 of Algorithm 4.1. Then, we modify Lines 19–20 as follows: pick a candidate from each group C_i and then assign their route requests to m threads in parallel.

For example, in Fig. 4.6e, the candidates are divided into m $(= 3)$ groups based on their direction angles from q: $C_1 = \{p_1, p_2\}$, $C_2 = \{p_3, p_4\}$, and $C_3 = \{p_5, p_6, p_7\}$. Again, the candidates within each C_i are examined by the DESC order. Fig. 4.6c illustrates the schedule of the direction-based approach. First, this approach selects the candidates $p_1 \in C_1$, $p_3 \in C_2$, and $p_6 \in C_3$, and issues route requests for them in parallel. Since the routes to p_1 and p_3 cover p_2 and

p_4 respectively, we saved two requests. After that, C_1 and C_2 become empty. In each subsequent slot, only one route request (for a candidate in C_3) is issued to the route APIs. In total, the direction-based approach incurs only 5 route requests, but it takes 3 slots.

Comparison. In summary, the greedy approach offers the best response time but with considerable extra route requests; the direction-based approach reduces the number of extra route requests and yet provides a competitive response time.

4.5 Experimental Evaluation

In this section, we compare the accuracy and the performance of our RouteSaver (abbreviated as RS) with an existing method SMashQ (abbreviated as SMQ) [132]. Although SMQ handles only KNN queries, we also adapt it to process range queries. Note that SMQ does not utilize any route log to save route requests. We also consider an extension of SMQ, called SMQ*, which keeps the routes within expiry time into a route log. SMQ* applies only the optimal subpath property [52] [121] and retrieves exact travel times from the log; however, it does not apply the upper/lower bounding techniques in this chapter. By default, RS uses the DESC and DIFF orderings for range and KNN queries respectively.

Section 4.5.1 describes our experimental setting. We first examine the accuracy of the methods on real traffic data in Section 4.5.2. Then, we study the performance and scalability of the methods in Section 4.5.3. Finally, in Section 4.5.4, we conduct small-scale experiments on Google Directions API [16], as

it imposes a daily request limit 2,500 per evaluation user [17]. Due to this limit, we use a simulated route API in Sections 4.5.2, 4.5.3.

4.5.1 Experimental Setting

Road networks. For accuracy experiments on real traffic data, we will discuss the road network and traffic data in Section 4.5.2.

For the performance and scalability study (Section 4.5.3), we obtain three road maps in USA from [1]: *Chowan* County, in North Carolina (14K nodes, 14K edges), *Erie* County, in Pennsylvania (106K nodes, 115K edges) and *Florida* State (1,049K nodes, 1,331K edges). Following [131], the maximum speed limit $\mathbb{V}_{MAX}$ is set to 110 km/h. According to [2], the travel speed of each road segment is set to a fraction of $\mathbb{V}_{MAX}$, based on its road category.

For the experiments on Google Directions API [16], we consider the *Manhattan* region (in New York), whose area is 87.5km^2.

Performance measure and parameters. For each method, we measure its result accuracy (Sec. 4.5.2), its number of route requests and user response time (Sec. 4.5.3). Table 4.4 summarizes the default values and ranges of parameters used in our experiments. The values for *dataset size* $|\mathcal{P}|$, K, T follow [131]. The default expiry time δ is 10 minutes, according to Fig. 4.2. To simulate the arrival of queries, we set the default query rate λ to 60 queries / min and uniformly generate query points on the road network. This query rate (60 queries / min) is justified by visit statistics[4.3] from restaurant and travel guide websites [22].

[4.3] E.g., *Hotels* has 2.38 million monthly visits, corresponding to the query rate λ = 2,380,000 / (30 · 24 · 60) = 55.1 queries / min. Similarly, *OpenTable* and *UrbanSpoon* have 2.9 and 4

Table 4.4. Experiment parameters

Parameters	Default	Range
Road map [only for simulation]	Erie	Chowan, Erie, Florida
Dataset size $\lvert\mathcal{P}\rvert$	10 (K)	1, 5, 10, 15, 20 (K)
Distribution of query q	uniform	uniform, gaussian
For KNN: Result size K	10	1, 5, 10, 15, 20
For range: Time limit T (*seconds*)	60	10, 30, 60, 90, 120
Expiry time δ (*minutes*)	10	0,2,5,10,20,30
Query rate λ (*queries/minute*)	60	30, 60, 120, 300
Number of threads	1 [sequential]	1, 2, 4, 6, 8, 10

All methods were implemented in C++ and ran on an Ubuntu 11.10 machine with a 3.4GHz Intel Core i7-3770 processor and 16GB RAM. In experiments, the route log contains at most 30,000 routes and occupies at most 30 MB. The largest road network (Florida) and dataset occupies 87 MB and 1 MB respectively. Thus, the largest map, route log, and dataset can fit in the main memory.

4.5.2 Accuracy on Real Traffic Data

In this section, we test the result accuracy of the methods on real traffic data, for various expiry time δ (2, 5, 10, 20 and 30 minutes). Other parameters $(|\mathcal{P}|, K, T)$ are set to default values in Table 4.4.

Real traffic data. We downloaded historical real traffic on freeways in Los Angeles from PeMS[4.4]. The corresponding road network contains 17,563 nodes and 17,694 edges. We use the traffic data on 31 Dec. 2012 for 24 hours; the travel times on edges are updated every 30 seconds. We also conduct this experiment

million monthly visits respectively, corresponding to $\lambda = 67.1$ and $\lambda = 92.4$. See the statistics at: http://www.quantcast.com/hotels.com http://www.quantcast.com/opentable.com http://www.quantcast.com/urbanspoon.com

[4.4]California Dept. of Transportation http://pems.dot.ca.gov/

with traffic data on other dates, and obtain similar results.

Accuracy measure. Besides the methods discussed before, we also consider a baseline method NoAPI, which uses only local distance information to answer queries, without issuing route requests.

We measure the *accuracy* of a method by the $F1$ score:

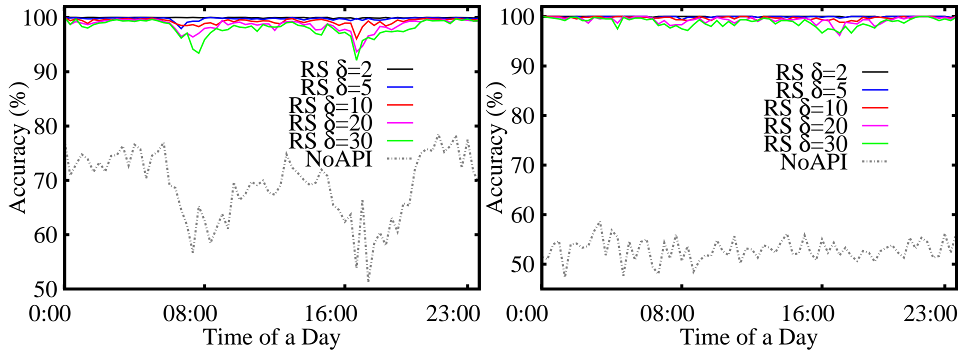
$$\begin{aligned} F1 &= 2 \cdot precision \cdot recall/(precision + recall) \\ precision &= \mid R_{method} \cap R^* \mid / \mid R_{method} \mid \\ recall &= \mid R_{method} \cap R^* \mid / \mid R^* \mid \end{aligned}$$

where R^* is the exact result set derived from the current traffic and R_{method} is the result set obtained by a method.

The accuracy of SMQ is always 100% because it does not use route log. We only measure the accuracy of RS, SMQ*, NoAPI in this experiment. Fig. 4.7a shows the average accuracy of the methods on a day. NoAPI has low accuracy as it does not use live traffic information. Our proposed RS and SMQ* can find results with very high accuracy. When the expiry time δ increases, the route log contains less accurate travel time information and thus the accuracy decreases. The standard deviation of the accuracy is within 1.5% for SMQ* and RS, whereas NoAPI has a higher standard deviation. Fig. 4.7b, 4.7c show the accuracy of RS and NoAPI along the timeline. As a remark, the traffic changes most rapidly during rush hours in the morning and the evening. During those intervals, the accuracy of the methods on range queries drops because their result sizes are sensitive to the traffic. The accuracy on KNN queries is insensitive to the traffic

δ (Min)	range			KNN		
	SMQ*	RS	NoAPI	SMQ*	RS	NoAPI
2	99.97	99.95	69.04	99.99	99.99	52.95
5	99.89	99.75		99.95	99.94	
10	99.36	99.28		99.68	99.65	
20	99.02	98.60		99.12	99.10	
30	98.63	98.11		98.95	98.86	

(a) average accuracy (%) on a day



(b) accuracy [range]

(c) accuracy [KNN]

Figure 4.7. Result accuracy on real traffic data [in color]

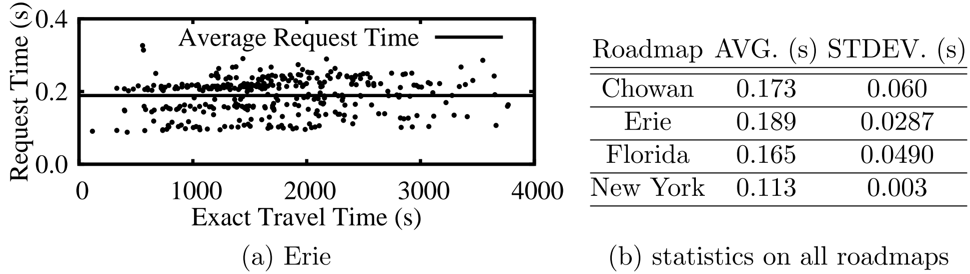
due to the fixed result size.

As we will show in Section 4.5.3, RS issues much fewer route requests than SMQ*. RS still achieves high accuracy because our proposed bounding techniques offers tight lower/upper bounds. We found in our experiments that, the upper bounds, if exist, are almost equal to the exact travel time in most cases, and the lower bounds are at least 60% of the exact travel time.

4.5.3 Performance and Scalability Study

For the sake of obtaining the user response time in our simulations, we measure the time of route requests on Google Directions API [16]. On each roadmap, we randomly sample 400 pairs of points and issue route requests for

them to Google Directions API. Fig. 4.8a plots the time of each route request versus its length (exact travel time), on the Erie roadmap. Fig. 4.8b summarizes the average and standard deviation of route request time on all roadmaps.



(a) Erie

Roadmap	AVG. (s)	STDEV. (s)
Chowan	0.173	0.060
Erie	0.189	0.0287
Florida	0.165	0.0490
New York	0.113	0.003

(b) statistics on all roadmaps

Figure 4.8. Time of route requests on roadmaps

Section 4.5.3.1 studies the temporal stability of the methods along the timeline. Section 4.5.3.2 examines the effect of our proposed optimizations. Section 4.5.3.3 tests the scalability of the methods with respect to various parameters. Section 4.5.3.4 evaluates the performance of RS with parallelization.

4.5.3.1 Temporal Stability

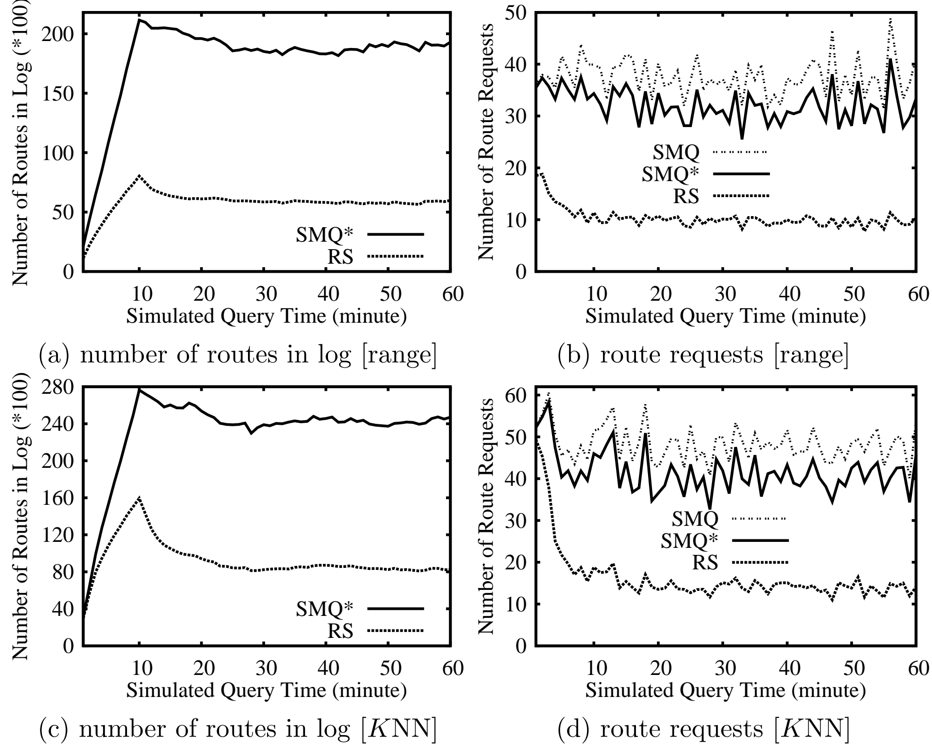
In this section, we simulate the arrival of queries along a 60-minute (1-hour) timeline, while fixing all parameters to default. Thus, each test uses $60 \cdot \lambda = 3600$ queries. The route log $\mathcal{L}$ is initially empty. To report temporal behavior, we measure (i) the route log size and (ii) the number of route requests of each query.

We first conduct experiments with uniformly distributed queries and datasets. Fig. 4.9a shows the number of routes in $\mathcal{L}$ of RS and SMQ* versus the timeline, for range queries. SMQ is not plotted here as it does not utilize the log $\mathcal{L}$. The log size rises steadily in the first $\delta = 10$ minutes (the warm-up period) and then

the expiration mechanism starts its effect. Observe that the drop in the log size during the $[10, 20)$ minutes matches with the drop in the number of route requests during the $[0, 10)$ minutes (see Fig. 4.9b). After that, the log size remains stable in subsequent minutes because $\mathcal{L}$ contains only the routes requested by the latest $\lambda \cdot \delta$ queries. SMQ* has a larger log size because it incurs more route requests than RS.

Fig. 4.9b illustrates the number of route requests of each query versus the timeline, for range queries. The performance of SMQ remains constant since it does not utilize the route log. In the first $\delta = 10$ minutes, as the log size of RS rises, it could exploit more information, like deriving exact values and tight lower/upper bounds for travel times, to reduce the number of route requests. After that, its log size keeps stable so its performance also keeps stable. The trend of SMQ* is similar to RS, except that SMQ* incurs much more route requests than RS. That is because SMQ* uses only the optimal subpath property to derive exact travel times from the route log, but it does not use the lower/upper bounds applied in RS. Experimental results on KNN queries are similar (see Fig. 4.9c,d).

As observed in the above experiments, the number of route requests converges to a stable value after the first δ minutes (the warm-up period). Thus, in subsequent experiments, we simulate the arrival of queries along 2δ-minute time interval. We view the first δ minutes as the *warm-up period*, and the last δ minutes as the *stable period*. We only measure the average performance in the stable period.



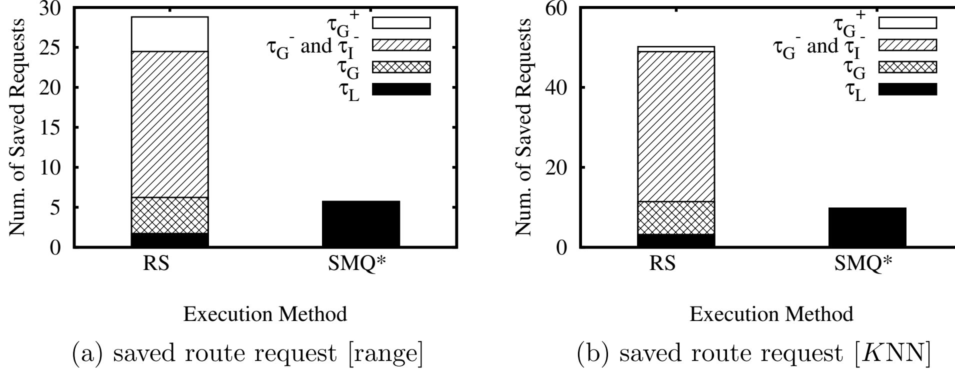
(a) number of routes in log [range] (b) route requests [range]

(c) number of routes in log [KNN] (d) route requests [KNN]

Figure 4.9. Temporal behavior, expiry time $\delta = 10$

4.5.3.2 Effect of Optimization Techniques

First, we investigate the effectiveness of our proposed lower/upper bound techniques. Recall that RS exploits the travel time information obtained from recent routes for three techniques: (i) retrieve the exact travel time of a point p, (ii) prune p by its lower bound $p.\tau_G^-, p.\tau_I^-$ (excluding cases using $p.\tau_c^-$), and (iii) detect p as a true hit by its upper bound $p.\tau_G^+$. We further divide technique (i) into two types: (i.a) existing technique using the optimal subpath property [52] on the route log $\mathcal{L}$, and (i.b) our proposed technique using Lemma 2 on the time-tagged network G. Note that SMQ* applies only technique (i.a), but not techniques (i.b), (ii), (iii).



(a) saved route request [range] (b) saved route request [KNN]

Figure 4.10. Effectiveness of techniques

Table 4.5. Effect of candidate ordering

	RS [range]		RS [KNN]		
	ASC	DESC	ASC	DESC	DIFF
route requests	17.12	11.57	21.29	20.29	18.39

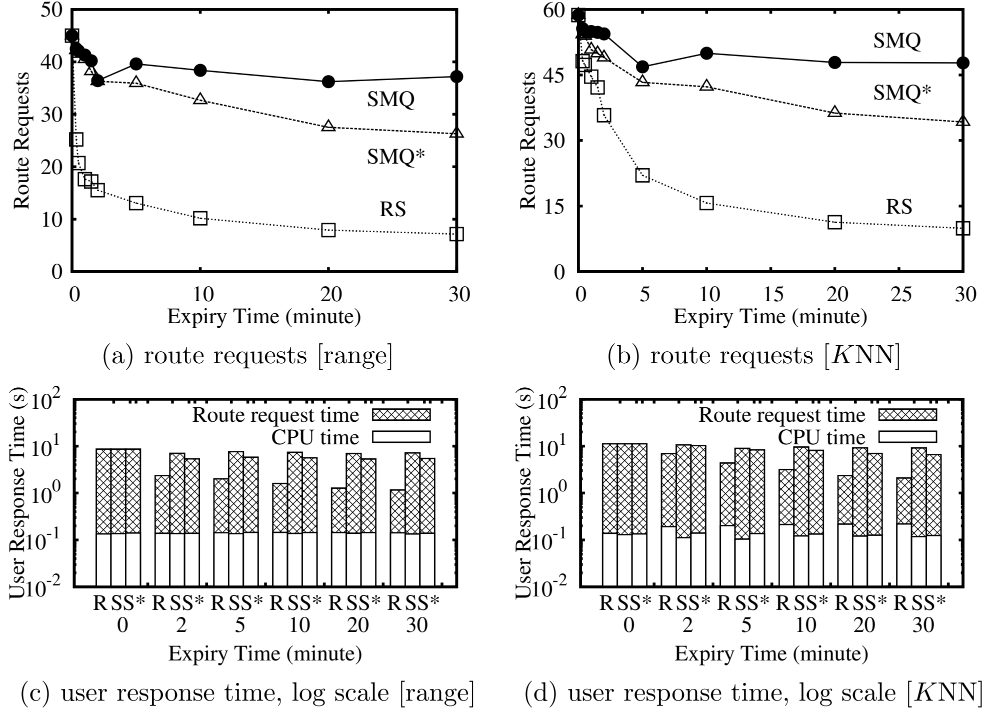
Fig. 4.10 depicts the statistics of applying these techniques in the methods, at the default setting. Observe that our proposed lower-bound technique (for computing $p.\tau_G^-, p.\tau_I^-$) saves the largest number of route requests, while the existing technique for computing exact travel time $p.\tau_{\mathcal{L}}$ (using optimal subpath property) saves the least. The reason for $p.\tau_G^-, p.\tau_I^-$ outperforming $p.\tau_G^+$ is that, RS has a higher chance to derive a tight $p.\tau_G^-, p.\tau_I^-$ for each data point, but a finite $p.\tau_G^+$ may not exist for a data point.

Next, we study the effect of candidate orderings on RS in terms of the number of route requests per query. It can apply the ASC / DESC orderings for range queries, and ASC / DESC / DIFF orderings for KNN queries. Table 4.5 shows that RS-DESC and RS-DIFF achieve the best performance for range and KNN queries, respectively.

4.5.3.3 Scalability Experiments

As discussed before, in this section, we simulate the arrival of queries along 2δ-minute time interval. And we measure the performance in terms of: (i) average number of route requests per query in the stable period, and (ii) average user response time per query in the stable period. Furthermore, we also plot the breakdown of user response time into server CPU time and the time spent on route requests, as illustrated in Section 4.4. The server CPU time already includes the overhead of maintaining the structures in Section 4.3.1.

Effect of expiry time δ. Fig. 4.11a shows the average number of route requests for range queries with respect to various δ. To illustrate the trend of route requests for smaller expiry times, we add the result for four more δ (20, 30, 60, 90 seconds) apart from the values listed in Table 4.4. Since SMQ does not use the log, its cost remains constant and much higher than that of RS and SMQ*. When δ increases, the route log of RS and SMQ* accumulates routes requested from more warm-up queries ($\lambda \cdot \delta$). Thus, RS and SMQ* could exploit more information in the log to reduce the cost. Fig. 4.11c illustrates the decomposition of the user response time for various δ. Here, 'R', 'S', 'S*' refer to RS, SMQ, SMQ*, respectively. To make the server CPU time visible, we plot the y-axis in log scale. Clearly, the time on route requests dominates the user response time. RS achieves a low server CPU time (0.1s) and user response time (1s). As a remark, the LBS's query throughput is decided only by its CPU time because it remains idle while issuing route requests. Fig. 4.11b,d depict the performance for KNN queries. The trends are similar to those for range queries. Due to the overhead on using route log, RS and SMQ* incur slightly higher server CPU time



(a) route requests [range] (b) route requests [KNN]

(c) user response time, log scale [range] (d) user response time, log scale [KNN]

Figure 4.11. Effect of expiry-time δ

than SMQ.

Since the user response time is mostly spent on route requests, we only report the number of route requests in experiments below.

Effect of query rate λ. As shown in Fig. 4.12a,b, the effect of query rate λ on the performance is similar to that of expiry time δ as discussed above. The reason is that, the route log of RS and SMQ* accumulate routes requested from more warm-up queries ($\lambda \cdot \delta$), as λ increases.

Effect of dataset size $|\mathcal{P}|$. In this experiment, we vary dataset size $|\mathcal{P}|$ and plot the number of route requests for range queries in Fig. 4.12c. The number of route requests rises proportionally to $|\mathcal{P}|$ as more objects are covered by the

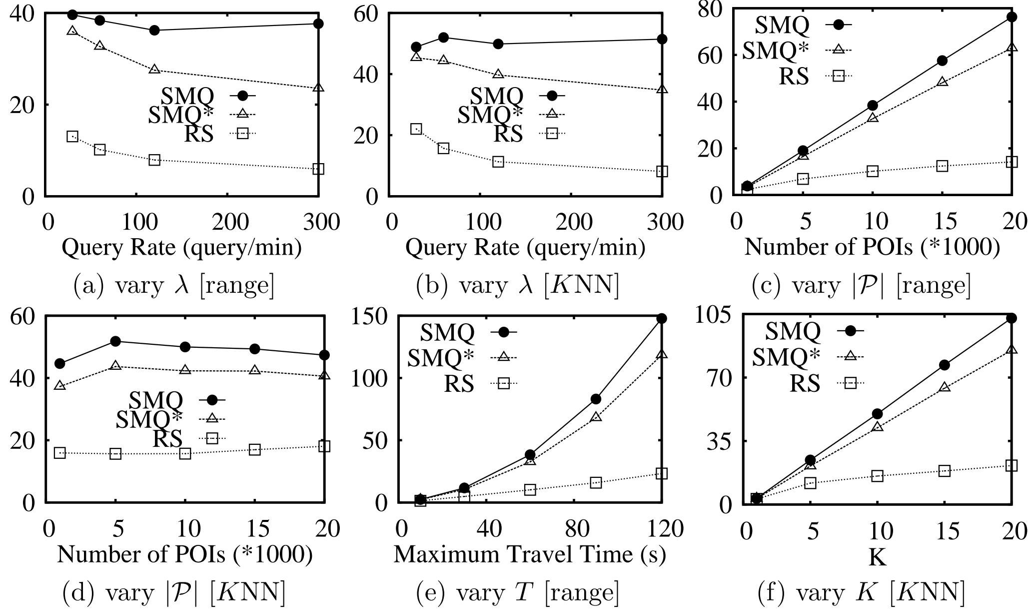


Figure 4.12. Effect of various parameters [y-axis: route requests]

query range. The performance gap between SMQ/SMQ* and RS widens because RS applies effective bounding techniques. In contrast, for KNN queries, the performance is insensitive to $|\mathcal{P}|$, as depicted in Fig. 4.12d. When $|\mathcal{P}|$ increases, the travel time from q to its KNN decreases. This enables pruning more candidates, canceling out the effect of $|\mathcal{P}|$.

Effect of time limit T and result size K. Fig. 4.12e shows the performance of the methods on range queries versus the travel time limit T. As T increases, the number of query results increases and so does the number of route requests. RS outperforms SMQ/SMQ* by a wide margin. As a remark, the average number of query results rises from 1.5 to 35.6 when T increases from 10s to 120s. Fig. 4.12f depicts the performance of the methods by varying K. Again, when the result

Table 4.6. Effect of roadmap on range and KNN queries

Roadmap	route requests [range]			route requests [KNN]		
	SMQ	SMQ*	RS	SMQ	SMQ*	RS
Chowan	37.81	18.76	2.37	44.03	20.38	3.29
Erie	40.53	36.1	11.92	49.23	41.36	16.8
Florida	44.7	40.07	18.71	55.31	54.51	25.01

size K increases, so does the number of route requests.

Effect of roadmaps. We then examine the effect of the roadmaps on the performance of the methods. Table 4.6 lists the roadmaps in ascending sizes (Chowan, Erie, Florida), together with the average number of route requests of the methods, for range queries and KNN queries. We follow the experimental methodology in [134] and fix the *object density* (i.e., the ratio of $|\mathcal{P}|$ over the number of nodes in the network) to 10%. That is, we have $|P| =$ 1.4K, 10.6K and 104.9K for Chowan, Erie and Florida, respectively. As the log routes have fewer intersections in larger road networks, the derived lower/upper bounds become looser, and thus the number of route requests increases in larger networks.

Effect of query distribution. This experiment illustrates the effect of query distribution on the performance of the methods. For each query set 'Gau_x%', we select 30 Gaussian bells randomly, set the standard deviation of each Gaussian to be x% of the map domain length [39], and generate points in these bells following such distribution. For comparison, we also use an uniformly generated query set ('Uniform').

Table 4.7 shows that, SMQ is insensitive to the distribution of the queries since it does not utilize the logs obtained from recent queries. For Gaussian queries, when the standard deviation is small, the current query is likely to be

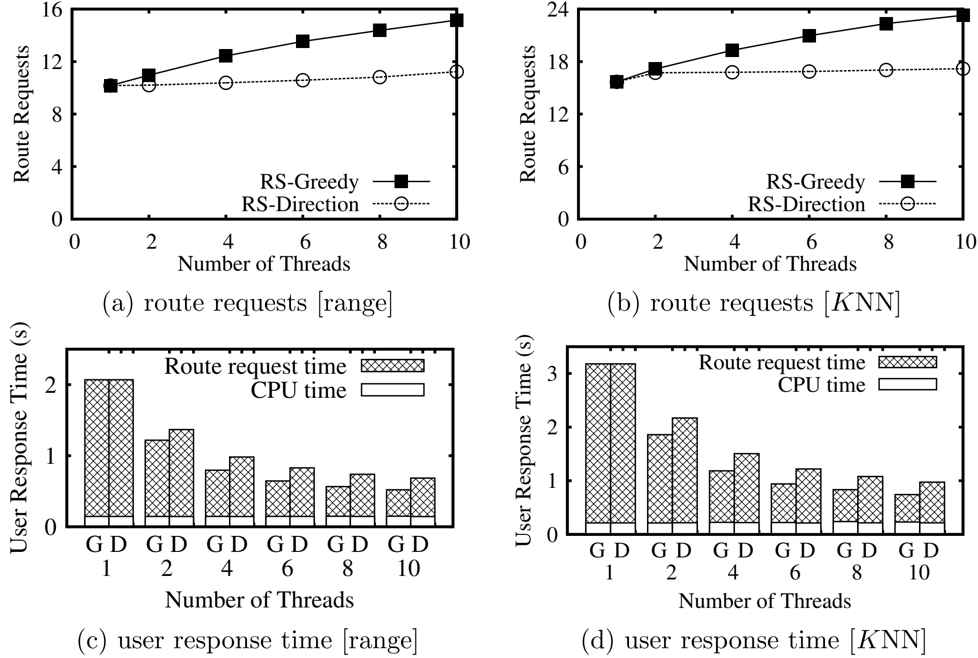
Table 4.7. Effect of query distribution

Distribution of query q	route requests [range]			route requests [KNN]		
	SMQ	SMQ*	RS	SMQ	SMQ*	RS
Gau_2.5%	42.43	17.44	3.83	48.61	15.33	5.51
Gau_5.0%	40.84	23.52	5.91	51.02	25.52	8.25
Gau_10%	41.42	28.53	8.51	51.24	28.43	11.52
Gau_20%	41.73	32.00	10.31	50.81	33.33	14.74
Gau_50%	40.92	33.21	10.98	48.85	36.53	15.65
Uniform	39.37	34.67	11.57	49.97	42.29	18.39

near to some recent queries, and thus recent routes provide valuable information for RS and SMQ* to save route requests. Observe that uniform query distribution, i.e., our default query distribution, leads to the the worst-case performance because the current query can be located far from recent queries and reuse less information from their routes.

4.5.3.4 Effect of Route Request Parallelization

This section studies the user response time of parallelization variants of RS: (i) RS-Greedy using greedy parallelization, and (ii) RS-Direction using direction-based parallelization. Fig. 4.13a,c show their average number of route requests and user response time versus the number of threads m, for range queries. At $m = 1$, both variants are the same as RS which issues route requests sequentially and incurs the longest user response time. As expected in Section 4.4, RS-Direction results in fewer route requests but a slightly longer user response time than RS-Greedy. We obtain similar experimental results for KNN queries in Fig. 4.13b,d.



(a) route requests [range] (b) route requests [KNN]

(c) user response time [range] (d) user response time [KNN]

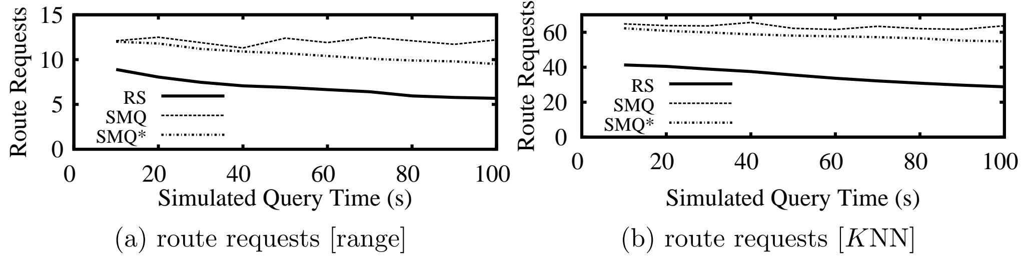
Figure 4.13. Effect of the number of threads m

4.5.4 Experiments on Google Directions API

We have implemented SMQ, SMQ* and RS with Google Directions API [16], whose request/response format has been described in Section 4.1.2. Due to the daily request limit (2,500) for evaluation users [17], we conduct this experiment on the Manhattan region (see Section 4.5.1). We randomly select 100 POIs[4.5] in this region, and generate 100 queries (along a 100-second time period).

Fig. 4.14 depicts the number of route requests of each query versus the timeline, for range queries and KNN queries. RS outperforms SMQ and SMQ* on both range queries and KNN queries. Also, the performance gap between them widens with the timeline. The number of route requests is still decreasing

[4.5] E.g., There are about 50 ATMs in the Manhattan region http://locators.bankofamerica.com/locator/locator/ListLoadAction.do



(a) route requests [range] (b) route requests [KNN]

Figure 4.14. Temporal behavior vs. timeline, $\delta = 10$ minutes, Manhattan region (in New York), on Google Directions API

as the timeline has not yet reached the (default) expiry time $\delta = 10$ minutes.

4.6 Chapter Summary

In this chapter, we propose a solution for the LBS to process range/KNN queries such that the query results have accurate travel times and the LBS incurs few number of route requests. Our solution Route-Saver collects recent routes obtained from an online route API (within δ minutes). During query processing, it exploits those routes to derive effective lower-upper bounds for saving route requests, and examines the candidates for queries in an effective order. We have also studied the parallelization of route requests to further reduce query response time. Our experimental evaluation shows that Route-Saver is 3 times more efficient than a competitor, and yet achieves high result accuracy (above 98%).

Chapter 5

Route Recommendation for Spatial Crowdsourcing Workers

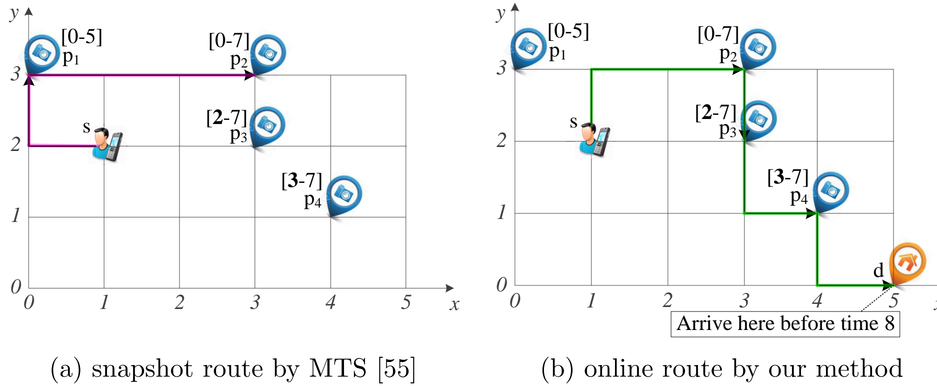
Spatial crowdsourcing platforms[5.1] [5.2] publish crowdsourcing tasks that are associated with rewards and tagged with spatial / temporal attributes (e.g., location, release time and deadline). To complete a task, a worker must reach the task's location before its deadline. Popular tasks include taking photos, reporting activities / accidents, and verifying data on-site, etc.

Regarding the matching between tasks and workers, existing approaches on spatial crowdsourcing can be divided into: (i) the *server-centric* mode [75, 76], where the server assigns tasks to workers based on their reported locations /

[5.1] www.clickworker.com/en/mobile-crowdsourcing
[5.2] features.en.softonic.com/mobile-crowdsourcing-does-it-work

regions, or (ii) the *worker-centric* mode [31, 40, 55], where the server publishes its tasks and let workers to choose any task freely. In this chapter, we adopt the worker-centric mode as it protects the location privacy of the worker [55] and enables the worker to choose tasks autonomously from different crowdsourcing platform which he has registered in.

The closest work to ours is the *maximum task scheduling* (MTS) problem [55]. It returns a route that covers the maximum number of tasks (in a worker's specified region, e.g., his city). Since [55] considers the MTS problem at a snapshot, it would not update the worker's route when new tasks arrive. We illustrate it in Figure 5.1a. Assume that we use the Manhattan distance and each grid takes a time unit to travel. Each task p_i is tagged with its release time and deadline. Suppose that the worker starts from s at time 0. The MTS route is $s \rightarrow p_1 \rightarrow p_2$. The solution in [55] would not update the route when new tasks are released (e.g., p_3, p_4).



(a) snapshot route by MTS [55]

(b) online route by our method

Figure 5.1. Route recommendation for the worker: each task p_i with [release time - deadline]

In this chapter, we wish to support two extra requirements compared to [55]:

($\mathcal{R}1$) update the worker's route online with respect to newly released tasks and ($\mathcal{R}2$) align with the worker's trip, i.e., reaching a destination before expected time. It is important to support $\mathcal{R}1$ in order to assign a worker as many tasks as possible. New spatial crowdsourcing tasks are indeed being released continuously in real systems[5.3]. We also consider requirement $\mathcal{R}2$ as the worker may have planned his own activities, e.g., reaching a specified destination by an expected time [90]. Such worker is willing to take crowdsourcing tasks along his trip provided that he can arrive at his destination on time.

To this end, we study the online route recommendation problem for spatial crowdsourcing workers, by taking requirements $\mathcal{R}1$ and $\mathcal{R}2$ into consideration. Figure 5.1b illustrates the route recommended by our method. Suppose that the worker starts from s at time 0 and plans to arrive at home $(5, 0)$ at time 8. At time 0, the worker is recommended to take the task p_2. When new tasks are released (e.g., p_3, p_4), the worker is recommended to take them. In summary, our recommended route is $s \to p_2 \to p_3 \to p_4 \to d$, which covers 3 tasks and reaches the destination d on time.

To the best of our knowledge, this chapter is the first on tackling the online route recommendation problem for spatial crowdsourcing workers with destination and arrival time constraints. We contribute the followings:

- We show that no algorithm can achieve a non-zero competitive ratio [30] in our online problem, meaning that the number of tasks found by any online algorithm may be arbitrarily small compared to the optimal offline solution.

[5.3] www.clickworker.com/en/clickworkerjob
www.lionbridge.com

– We propose two categories of heuristics (GetNextTask and Re-Route) that offers trade-offs between the response time and the number of tasks. GetNextTask greedily selects the next task to complete so it incurs a short response time. On the other hand, Re-Route produces a route with more tasks as it conducts a complete search to update the optimal route with respect to newly released tasks.

– We further propose pruning rules to reduce the response time of Re-Route.

Experiments on real datasets show that our methods take less than 1 second to update the route, and return routes that contain 82–91% of the optimal number of tasks.

The remainder of this chapter is organized as follows. We formally define our problem in Section 5.1. Then, we illustrate our proposed heuristics in Section 5.2 and present optimization techniques in Section 5.3. After that, some discussions are made on the Route Recommendation in Section 5.4. In Section 5.5, we test the performance of our proposed techniques on both real and synthetic datasets. Section 5.6 highlights the related work. Finally, we conclude our chapter in Section 5.7.

5.1 Problem Statement

We first introduce some terminology and then define our problem formally.

Definition 4 (Task p). *We denote a task by* $p_{sid,kid} = (loc, [t_p^-, t_p^+])$*, where loc is the task's location,* t_p^-, t_p^+ *are the release time and deadline of the task,*

respectively. The subscripts sid and kid denote the task's server ID and task ID, respectively. A worker may complete p and collect the reward[5.4] *if he can reach $p.loc$ before t_p^+.*

Definition 5 (Query q). *We denote a query q by $q = (s, d, [t_q^-, t_q^+])$. s and d are the worker's start and destination locations, respectively. t_q^- and t_q^+ are the start time from s and expected arrival time at d, respectively.*

Definition 6 (Travel Time τ). *We denote the travel time as $\tau(v, u) = \frac{dist(v,u)}{speed_q}$, where $dist(v, u)$ is the distance*[5.5] *between v and u, and $speed_q$ is the (constant) travel speed of the worker for q. $\tau(R)$ denotes the travel time along a route R (via vertices on R).*

With the above terminology, we are ready to define our problem formally below.

Problem 1 (Oriented Online Route Recommendation (OnlineRR)). *Let a worker's query be $q = (s, d, [t_q^-, t_q^+])$.* OnlineRR *aims to find a route such that it covers* the maximum number of tasks *and the worker can arrive at d by t_q^+. It may update the route according to the worker's live location and the new tasks released by crowdsourcing servers.*

We adopt the system architecture as depicted in Figure 5.2. Spatial crowdsourcing servers publish new spatial crowdsourcing tasks. A worker may install our *route recommender* on his mobile device (smartphone). The route recommender is responsible for: (i) collecting task information from different servers

[5.4] The reward of a task can be collected by the same worker for only once. Similar to [55], we assume that each task has a unit reward and can be completed immediately.

[5.5] Our method can be applied to any distance function provided that it satisfies the triangle inequality, such as Euclidean distance, Manhattan distance, and road network distance.

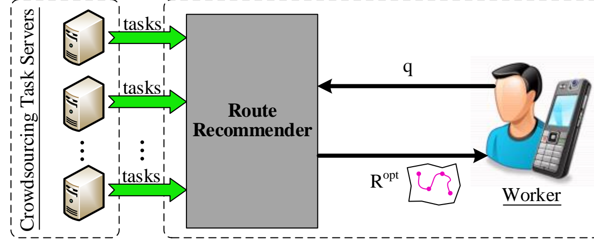


Figure 5.2. System architecture

continuously, (ii) recommending / updating a route based on the worker's current location and available tasks.

5.2 Online Route Recommendation

First, we prove in Section 5.2.1 that no online algorithm can achieve a non-zero competitive ratio in OnlineRR. Then, we propose two categories of heuristic approaches for OnlineRR in Sections 5.2.2 and 5.2.3.

5.2.1 Competitive Analysis

We use the competitive ratio [30] to measure the performance of online algorithms. Since OnlineRR is a maximization problem, the competitive ratio $\mathcal{CR}$ is defined as:

$$\mathcal{CR} = \min_{e \in E} \frac{count(R_{alg}(e))}{count(R_{opt}(e))} \tag{5.1}$$

where E denotes the set of all problem instances, $R_{alg}(e)$ is the route recommended by an online algorithm alg for instance e, $R_{opt}(e)$ is the optimal route R_{opt} for instance e (cf. Definition 7), and $count(R_*(e))$ means the number of tasks on $R_*(e)$.

Definition 7 (Optimal route $R_{opt}(e)$ for OnlineRR). *Given a problem instance e, we denote its optimal route by $R_{opt}(e)$, which is obtained under assumption that the information of all tasks are known in advance (even before their release times).*

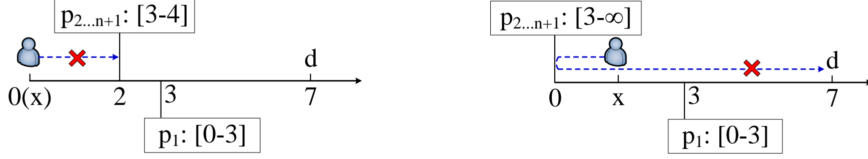
We show our competitive analysis below. It applies to any online algorithm, including both deterministic algorithms and randomized algorithms.

Theorem 4. *No online algorithm has a non-zero competitive ratio for* OnlineRR.

Proof. Since $\mathcal{CR} = \min_{e \in E} \frac{count(R_{alg}(e))}{count(R_{opt}(e))}$, it suffices to find a specific instance (i.e., the adversary) that makes $\mathcal{CR}$ as low as possible. Without loss of generality, in the following proof, we consider only locations on the positive half line $[0, +\infty)$. For the query, we set $t_q^- = 0$, $s = 0$, $t_q^+ = 10, d = 7$. Assume that $speed_q = 1$, that is $\tau_e(v, u) = |v - u|$. We simply denote a task p by $(p.loc, [t_p^-, t_p^+])$.

At time 0, the adversary releases a task $p_1 = (3, [0, 3])$. At time $m = 3$, the adversary will check the worker's current location (say x), and then decides to further release n tasks accordingly. There are two cases: (1) $x = 0$, or (2) $x > 0$. We show that the adversary can release those n tasks to make $\mathcal{CR}$ arbitrarily small.

Case 1: $x = 0$. In this case, the adversary will release tasks $p_{2 \leq i \leq n+1} = (2, [3, 4])$ (see Figure 5.3a). The worker cannot complete these tasks, since he cannot reach them before their deadlines, and thus $count(R_{alg}) = 0$. But if all tasks are known in advance, the worker can wait at position 2 until all tasks are released and finish them on time $m = 3$. In this case, the competitive ratio is: $\mathcal{CR} = 0/n = 0$.



(a) Case 1: the worker cannot reach the location of $p_{2\leq i\leq n+1}$ before their deadline (i.e., time 4)

(b) Case 2: the worker cannot proceed to $p_{2\leq i\leq n+1}$ and arrive at d on time

Figure 5.3. At time $m = 3$, adversaries release tasks $p_{2\leq i\leq n+1}$ with [release time - deadline]

Case 2: $x > 0$. In this case, the adversary would release n tasks $p_{2\leq i\leq n+1} = (0, [m, \infty])$ (see Figure 5.3b). As $m+x+d > m+d = 10 = t_q^+$, the worker cannot proceed to position 0 at time m; otherwise, he cannot reach d before t_q^+. So, the worker can finish at most the task p_1 only if he moves directly to m at time 0. However, if all tasks are known in advance, the worker could stay at 0 until time $m = 3$ to finish tasks $p_{2\leq i\leq n+1}$, and thus $count(R_{opt}) = n$. Therefore, $\mathcal{CR} \leq 1/n \to 0$ because n can be an arbitrary large value. □

5.2.2 Greedy Task Approach

In this section, we present a greedy approach that incurs low response times.

The greedy approach works as follows. Initially, it calls GetNextTask (cf. Algorithm 5.1) to find the first task for the worker. Given the set of available[5.6] tasks P and the worker's location s_{now} at current time t_{now}, GetNextTask greedily selects the task with the highest score ψ_p. Upon reaching the chosen task, GetNextTask is involved to get the next task repeatedly until reaching d.

[5.6] Available tasks are tasks released before the current time t_{now}.

Algorithm 5.1 Get next best task

```
    algorithm GetNextTask (Query q = (s_now, d, [t_now, t_q^+]), Set of available tasks P)
1: Cand ← compute the set of feasible tasks from P                ▷ apply Equation 5.2
2: if Cand ≠ ∅ then
3:     p_next ← choose p ∈ Cand with best score ψ_p           ▷ ψ_p is a heuristic function
4:     Return p_next
5: else
6:     Apply policy P_stay or P_go      until Cand ≠ ∅ or t_now + τ(s_now, d) = t_q^+
```

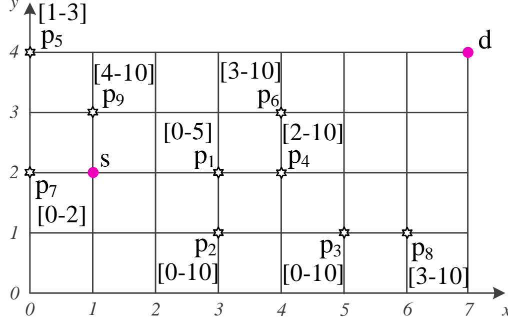
Due to the tasks' deadlines and the worker's expected arrivial time (cf. Definitions 4, 5), the worker may complete a task p if: (i) he can reach $p.loc$ before t_p^+, and (ii) he can reach d no later than t_q^+. Therefore, we call a task to be *feasible* if it satisfies:

$$\tau(s_{now}, p) + \tau(p, d) \leq t_q^+ - t_{now} \qquad \text{and} \qquad t_{now} + \tau(s_{now}, p) \leq t_p^+ \tag{5.2}$$

If there is no feasible task for q, the worker may stay or move based on a predefined policy (cf. Line 6 in Algorithm 5.1). In the policy $\mathcal{P}_{go}$, the worker simply moves towards the destination d. In the policy $\mathcal{P}_{stay}$, the worker waits at s_{now} until $t_{now} + \tau(s_{now}, d) = t_q^+$. When new feasible tasks are released, we resume the search and invoke GetNextTask to obtain the next task.

We illustrate several heuristics for computing the score ψ_p. Figure 5.4a shows the map of tasks which are labeled with release times and deadlines, and Figure 5.4b shows the result route of each heuristic. In this example, we use the query $q = (s, d, [0, 10])$, the policy $\mathcal{P}_{stay}$, and the Manhattan distance.

Nearest Neighbor Heuristic (G-NN). It chooses the nearest feasible task to the worker's current location s_{now}, and thus setting $\psi_p = \tau(s_{now}, p)$. In Figure 5.4, G-NN produces the route $\langle s, p_7, p_5, d \rangle$.



(a) map of tasks with [release time - deadline]

Heuristic	Route
G-NN	$\langle s, p_7, p_5, d \rangle$
G-SD	$\langle s, p_7, p_5, d \rangle$
G-MCS	$\langle s, p_1, p_4, p_6, d \rangle$
Re-Route	$\langle s, p_1, p_2, p_3, p_8, d \rangle$

(b) result routes (with $\mathcal{P}_{stay}$)

Route	# of tasks	Route	# of tasks	Route	# of tasks
$\langle s, p_1, d \rangle$	1	$\langle s, p_2, d \rangle$	1	$\langle s, p_3, d \rangle$	1
$\langle s, p_7, d \rangle$	1	$\langle s, p_1, p_2, d \rangle$	2	$\langle s, p_1, p_3, d \rangle$	2
$\langle s, p_2, p_3, d \rangle$	2	$\langle s, p_7, p_1, d \rangle$	2	$\langle s, p_1, p_2, p_3, d \rangle$	3
$\langle s, p_3, p_1, d \rangle$	not feasible	$\cdots$	not feasible	$\cdots$	not feasible

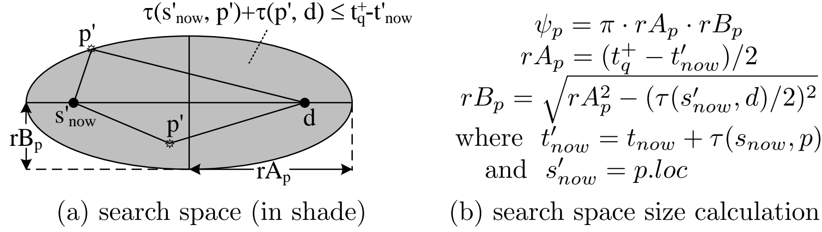
(c) all possible routes known at $t_{now} = 0$

Figure 5.4. Example of query $q = (s, d, [0, 10])$ in OnlineRR (using Manhattan Distance)

Earliest Deadline Heuristic (G-SD). It chooses the task with the earliest deadline, and thus setting $\psi_p = t_p^+$. In Figure 5.4, G-SD recommends the route $\langle s, p_7, p_5, d \rangle$.

Maximum Candidate Space Heuristic (G-MCS). It chooses the task p that can maximize the search space of feasible tasks (Equation 5.2) in future. The search space in future is obtained under the assumption that p is just completed. The space shape differs for different distance metrics, but we can use a general approach Monte Carlo [107] to compare it. If a specific distance metric is used, then the exact candidate space size can be calculated. Take Euclidean distance for example, the space size is the area of the ellipse shown in Figure 5.5a, and

thus we can calculate the score ψ_p using equations in Figure 5.5b for Euclidean distance metric.



(a) search space (in shade) (b) search space size calculation

Figure 5.5. Feasible candidates search space for Euclidean distance metric

We illustrate how G-MCS works in Figure 5.4. At time 0, the feasible tasks are p_1, p_2, p_3, p_7. Since p_1 has the highest score (ψ_{p_1}), p_1 is chosen to be visited. When the worker reaches p_1, a new task p_4 is released while p_7 expires, so the set of feasible tasks becomes $\{p_2, p_3, p_4\}$. Then p_4 is chosen as it has the highest score (ψ_{p_4}). Upon reaching p_4, the algorithm selects p_6 as it has the best score among $\{p_3, p_6, p_8\}$. After completing task p_6, there are no more feasible tasks. After waiting for two more time units, the worker moves toward d. In summary, G-MCS obtains the route $\langle s, p_1, p_4, p_6, d \rangle$.

5.2.3 Complete Search for Route Approach

In this section, we present a complete search approach that tends to find more tasks than the heuristics in Section 5.2.2.

Specifically, we formulate the following SnapshotRR problem, which takes the current query and the set of available tasks as input. Then, we solve SnapshotRR by enumerating all possible routes and obtain the one with the maximum number of tasks.

Problem 2 (Snapshot Route Recommendation (SnapshotRR)). *Given a query*

$q = (s_{now}, d, [t_{now}, t_q^+])$ *at the current snapshot* t_{now}*,* SnapshotRR *aims to find a route such that it covers* the maximum number of tasks *and the worker can arrive at* d *by* t_q^+*.*

We illustrate this approach for the query $q = (s, d, [0, 10])$ in Figure 5.4. At time 0, we apply Equation 5.2 and obtain the set of feasible tasks: $P = \{p_1, p_2, p_3, p_7\}$. Figure 5.4c shows all possible routes (known at time 0). The optimal route at time 0 is $\langle s, p_1, p_2, p_3, d\rangle$.

We propose a simple optimization to solve SnapshotRR in Algorithm 5.2. At Line 3, we check whether there exists a new feasible task p (that was not available in the previous call of Algorithm 5.2). If such p exists, we must solve SnapshotRR again. Otherwise, the best route remains the same as in the previous call, so we need not solve SnapshotRR again.

Algorithm 5.2 Complete search the result route

```
    algorithm Re-Route (Query q = (s_now, d, [t_now, t_q^+]), Set of available tasks P)
1: Let P_prev be the set of available tasks in the previous call
2: if P ≠ ∅ then
3:     if ∃p ∈ P − P_prev such that p is feasible then          ▷ Equation 5.2
4:         R ← Solve SnapshotRR(q, P)                        ▷ conduct complete search
5: else
6:     Apply policy 𝒫_stay or 𝒫_go     until P ≠ ∅ or t_now + τ(s_now, d) = t_q^+
```

We proceed to illustrate how Re-Route works in the example in Figure 5.4. At time 0, Re-Route computes the route $R_0 = \langle s, p_1, p_2, p_3, d\rangle$, and then the worker moves along R_0 to p_1. Upon reaching p_1, a new feasible task p_4 is found, so Re-Route re-calculates the route as $R_1 = \langle p_1, p_2, p_3, d\rangle$. When the worker reaches p_2, a new feasible task p_8 is found, so Re-Route updates the route to $R_2 = \langle p_2, p_3, p_8, d\rangle$. After reaching p_8, a new task p_9 is found but it is not

feasible. Thus, Re-Route would not computes the route again (cf. Line 3 in Algorithm 5.2). Eventually, the worker moves to d. In summary, the actual route traveled by the worker is: $\langle s, p_1, p_2, p_3, p_8, d\rangle$. It covers more tasks than other heuristics (cf. Figure 5.4b).

Since it is expensive to solve SnapshotRR by enumerating all possible routes, we will present optimizations to solve SnapshotRR efficiently in Section 5.3.

5.3 Optimization for SnapshotRR

We adapt the *bi-directional search* algorithm for the Orienteering Problem with Time Windows (OPTW) problem [106] to solve our problem. For brevity in discussion, we use $q = (s, d, [t_q^-, t_q^+])$ instead of $q = (s_{now}, d, [t_{now}, t_q^+])$. We will conduct bi-directional search for SnapshotRR in three steps:

Step 1: Search sub-routes in the forward direction (from s) and store in $\overrightarrow{\mathbb{R}}$

Step 2: Search sub-routes in the backward direction (from d) and store in $\overleftarrow{\mathbb{R}}$

Step 3: Join sub-routes between $\overrightarrow{\mathbb{R}}$ and $\overleftarrow{\mathbb{R}}$

According to Pruning Rule 1, the bi-directional search can reduce the search space. However, the method in [106] does not exploit spatial properties in our problem. In this section, we develop more effective pruning rules to accelerate bi-directional search on SnapshotRR.

Pruning Rule 1 (Half travel time bound property proved in [106]). *In the forward (or backward) route searching from vertex s (or d), only routes R with $\tau(R) \leq \tau_{max}/2$ are maintained and extended, where $\tau_{max} = t_q^+ - t_q^-$.*

5.3.1 Forward Search and Backward Search

In this section, we elaborate the forward search (Step 1) and discuss adaptations for the backward search (Step 2) at the end. In the following discussion, we use R instead of $\overrightarrow{R}$ to represent a sub-route found in forward search (which will be stored in $\overrightarrow{\mathbb{R}}$) for simplicity.

We first introduce the sub-route concept and its extension operation. Then, we propose a pruning rule and a search strategy to speedup the computation. In the following, we denote the set of vertices as $V = P \cup \{s, d\}$, where P is the set of available tasks.

Sub-route Extension.

We denote a path from s to $v \in V$ as a sub-route R_v, which contains four attributes $R_v = (\tau(R_v), B_{R_v}, C_{R_v}, v)$.

- $\tau(R_v)$ represents the travel time along R_v (i.e., from s to v).
- B_{R_v} stores a sequence of tasks visited before on the sub-route R_v. We denote the profit of R_v as $|B_{R_v}|$ because all tasks have the same reward.
- C_{R_v} is a set of candidate vertices (that are feasible for visiting in future), and its calculation is discussed in Equation 5.5.

During route search, for each vertex v, we store all sub-routes of the form R_v into a set $\mathbb{R}_v$. In addition, we only consider *feasible routes.* Recall that $\tau(R_v)$ represents the travel time (along R_v) from s to v. According to Equation 5.2, a

sub-route R_v is said to be *feasible* if:

$$\tau(R_v) \leq t_v^+ - t_q^- \qquad \text{and} \qquad \tau(R_v) \leq t_q^+ - t_q^- \tag{5.3}$$

where t_v^+ is the deadline for vertex v when v is a task, or ∞ when $v \in \{s, d\}$.

For each vertex $u \in C_{R_v}$, we can extend R_v with an arc (v, u) to form a new sub-route R_u. The component of $R_u = (\tau(R_u), B_{R_u}, C_{R_u}, u)$ is calculated as follows:

$$B_{R_u} \leftarrow \langle B_{R_v},\ v\rangle \quad \text{and} \quad \tau(R_u) \leftarrow \tau(R_v) + \tau_e(v, u) \tag{5.4}$$

The set C_{R_u} contains each candidate vertex p that satisfies:

$$\begin{gathered} p \in C_{R_v}(\heartsuit) \quad \text{and} \quad p \notin B_{R_u}(\diamondsuit) \\ \tau(u,p) \leq t_p^+ - t_q^- - \tau(R_u) \quad \text{and} \quad \tau(u,p) \leq (t_q^+ - t_q^-)/2 - \tau(R_u)(\clubsuit, \spadesuit, \blacklozenge) \\ \tau(u,p) + \tau(p,d) \leq t_q^+ - t_q^- - \tau(R_u)(\clubsuit, \blacktriangledown) \end{gathered} \tag{5.5}$$

which involve the constraints in Equation 5.4 ($\clubsuit$), Equation 5.3 ($\spadesuit$), triangle inequality ($\heartsuit$), the constraint that each task can be visited only once ($\diamondsuit$), the worker's arrival time t_q^+ ($\blacktriangledown$) and Pruning Rule 1 ($\blacklozenge$).

We illustrate sub-route extension in Figure 5.6. Assume that $q = (s, d, [0, 10])$ and $P = \{p_1, p_2, \cdots, p_7\}$. We consider Manhattan distance in this example. First, we compute the candidate set of s. By Pruning Rule 1, we only consider tasks within $10/2 = 5$ units from s (i.e., tasks in the dotted diamond in Figure 5.6). Thus, tasks p_3, p_7 are not feasible. The tasks p_4 and p_5 are not feasible as they violate constraints on the task's deadline and the worker's arrival time,

respectively. Thus, we obtain the candidate set of s as $C_s = \{p_1, p_2, p_6\}$, and compute the sub-route for s as $R_s = (0, \emptyset, C_s, s)$. Next, we append arcs (s, p_1), (s, p_2), (s, p_6) into R_s to generate three new sub-routes: $R_1 = (1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)$, $R_2 = (3, \langle p_2 \rangle, \{p_6\}, p_2)$, $R_6 = (5, \langle p_6 \rangle, \emptyset, p_6)$.

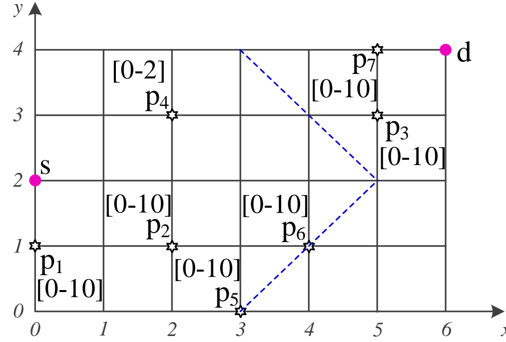


Figure 5.6. Example query $q = (s, d, [0, 10])$ for SnapshotRR problem (using Manhattan distance)

Dominate Test Pruning.

We develop the following pruning rule to further reduce the search space.

Pruning Rule 2 (Dominating Pruning)**.** *Let $R_v = (\tau(R), B_{R_v}, C_{R_v}, v)$ and $R'_v = (\tau(R'_v), B_{R'_v}, C_{R'_v}, v)$ be two feasible routes associated with v. We can prune R'_v if:*

$$\tau(R_v) \leq \tau(R'_v) \quad and \quad |C_{R'_v} \cap B_{R_v}| \leq |B_{R_v}| - |B_{R'_v}|$$

Proof. Among all full routes with R'_v as the prefix, let $R'_{opt} = \langle s, B_{R'_v}, R'_{tail}, d \rangle$ be the maximum reward route. With the given condition $\tau(R_v) \leq \tau(R'_v)$, after traveling along R_v, we can still follow all tasks in R'_{tail} and arrive at d by t^+_q. There exists a route $R_{exist} = \langle s, B_{R_v}, R_{tail}, d \rangle$ where $R_{tail} = R'_{tail} - B_{R_v}$. R_{exist}

ensures that the reward of each task is gained at most once as B_{R_v} and R_{tail} have no common tasks.

Since $R'_{tail} \subseteq C_{R'_v}$, we have $|R'_{tail}| = |R_{tail}| + |R'_{tail} \cap B_{R_v}| \leq |R_{tail}| + |C_{R'_v} \cap B_{R_v}|$.

By combining the above with the given condition $|C_{R'_v} \cap B_{R_v}| \leq |B_{R_v}| - |B_{R'_v}|$, we derive: $|B_{R_v}| + |R_{tail}| \geq |B_{R'_v}| + |C_{R'_v} \cap B_{R_v}| + |R_{tail}| \geq |B_{R'_v}| + |R'_{tail}|$. As the reward of R_{exist} (extended from R_v) is greater than or equal to that of R'_{opt} (extended from R'_v), we can prune the subroute R'_v. □

Search Strategy.

Our strategy is to identify sub-routes with better reward values in order to utilize pruning rule 2. To do so, we introduce the concept of upper bound reward:

Definition 8 (Vertex upper bound reward $\$^+_v$). *Given a sub-route $R_v = (\tau(R_v), B_{R_v}, C_{R_v}, v)$, we define its upper bound reward as:*
$\$^+_{R_v} = |B_{R_v}| + |C_{R_v}|$.
The upper bound reward of vertex $v \in V$ is defined as:
$\$^+_v = \max\{\$^+_{R_v} \mid R_v \in \overrightarrow{\mathbb{R}_v}\}$.

Initially, we begin the search from a sub-route at s. We iteratively extend sub-routes found so far and apply pruning rule 2 to discard unpromising sub-routes. During the search, we employ a heap H to process vertices in descending order of $\$^+_v$.

We illustrate this method on the example in Figure 5.6 and show the running

steps in Table 5.1. Iteration 1 corresponds to the extension of the sub-route R_s at s, which we have discussed before. We obtain three new subroutes R_1, R_2, R_6, insert them in their corresponding route sets $\overrightarrow{\mathbb{R}_p}$, and also enheap p_1, p_2, p_6 into H. In each subsequent iteration, we deheap the vertex $v \in H$ with the largest $\$_v^+$, and extend its sub-routes R_v in the descending order of $|B_{R_v}|$.

In iteration 2, we generate a new sub-route $(3, \langle p_1, p_2 \rangle, \{p_6\}, p_2)$ and apply Pruning Rule 2 to discard the previous subroute at p_2, i.e., $(3, \langle p_2 \rangle, \{p_6\}, p_2)$. Similarly, the previous sub-routes for p_6: $(5, \langle p_6 \rangle, \emptyset, p_6)$ and $(5, \langle p_1, p_6 \rangle, \emptyset, p_6)$ are pruned in iterations 2 and 3, respectively.

The forward search terminates when H becomes empty, i.e., no sub-routes can be extended. It returns the set $\overrightarrow{\mathbb{R}}$ of all surviving sub-routes.

Algorithm 5.3 illustrates the pseudo code of route search in forward direction. It is self-explanatory and summarizes what we have discussed above.

Backward Search. Route space search in backward direction is similar to that in forward direction. The pruning rules, searching strategies, and dominating testing discussed for forward search can be modified for backward search directly.

Table 5.1. Forward space search

Iteration	Selected Vertex	Extended Route R	Modified $\mathbb{R}$	Heap H
1	s	$(0, \emptyset, \{p_1, p_2, p_6\}, s)$	$\overrightarrow{\mathbb{R}_{p_1}} = \{(1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)\}$ $\overrightarrow{\mathbb{R}_{p_2}} = \{(3, \langle p_2 \rangle, \{p_6\}, p_2)\}$ $\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_6 \rangle, \emptyset, p_6)\}$	$(p_1, 3)$ $(p_2, 2)$ $(p_6, 1)$
2	p_1	$(1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)$	$\overrightarrow{\mathbb{R}_{p_2}} = \{(3, \langle p_1, p_2 \rangle, \{p_6\}, p_2)\}$ $\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_1, p_6 \rangle, \emptyset, p_6)\}$	$(p_2, 3)$ $(p_6, 2)$
3	p_2	$(3, \langle p_1, p_2 \rangle, \{p_6\}, p_2)$	$\overrightarrow{\mathbb{R}_{p_6}} = \{(5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)\}$	$(p_6, 3)$
4	p_6	$\emptyset$	$\emptyset$	$\emptyset$
$\overrightarrow{\mathbb{R}}$	$(5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)$, $(3, \langle p_1, p_2 \rangle, \{p_6\}, p_2)$, $(1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)$, $(0, \emptyset, \{p_1, p_2, p_6\}, s)$			

Algorithm 5.3 Forward Search

function RouteSearchFW(Query $q = (s, d, [t_q^-, t_q^+])$, Vertex set $V = P \cup \{s, d\}$)
▷ Initialization
1: Create an empty set $\overrightarrow{\mathbb{R}_v}$ for each vertex $v \in V$ to store sub-routes associated with v
2: Calculate the candidate vertex set C_s of s ▷ Equation 5.5
3: $\overrightarrow{\mathbb{R}_s} \leftarrow \{(0, \emptyset, C_s, s)\}$
4: Create a max-heap $H \leftarrow \{(s, |C_s|)\}$ to store vertices whose routes will be extended
▷ Repeatedly generate feasible sub-routes
5: **while** $H \neq \emptyset$ **do**
6: $(v, v.ub) \leftarrow$ Extract-Max(H) ▷ Searching strategy
7: Sort routes $R \in \overrightarrow{\mathbb{R}_v}$ in the descending order of $|B_R|$ ▷ Searching strategy
8: **for all** $R_v \in \overrightarrow{\mathbb{R}_v}$ **do**
9: **for all** $u \in C_{R_v}$ **do**
10: $R_u \leftarrow$ Extend(R_v, q, u) ▷ Equation 5.4, 5.5, Pruning Rule 1
11: RemoveDominate($\overrightarrow{\mathbb{R}_u}, R_u$) ▷ Pruning Rule 2
12: **if** $R_u \in \overrightarrow{\mathbb{R}_u}$ **then** ▷ R_u not pruned
13: **if** $(u, u.ub) \notin H$ **then**
14: Insert $(u, \$^+_{R_u})$ into H
15: **else**
16: $u.ub \leftarrow \max\{u.ub, \$^+_{R_u}\}$
17: Return $\overrightarrow{\mathbb{R}} \leftarrow$ all routes in each nonempty $\overrightarrow{\mathbb{R}_v}$

5.3.2 Route Join

In this section, we elaborate on how to join sub-routes obtained in the forward search and the backward search. Let $\overrightarrow{R_v} = (\tau(\overrightarrow{R_v}), B_{\overrightarrow{R_v}}, C_{\overrightarrow{R_v}}, v)$ and $\overleftarrow{R_u} = (\tau(\overleftarrow{R_u}), B_{\overleftarrow{R_u}}, C_{\overleftarrow{R_u}}, u)$ be two sub-routes in the forward and the backward directions, respectively. They are *feasible* to be joined if:

$$\tau(\overrightarrow{R_v}) + \tau(v, u) \leq u.t_p^+ \qquad \text{and} \qquad \tau(\overrightarrow{R_v}) + \tau(\overleftarrow{R_u}) + \tau(v, u) \leq t_q^+ - t_q^-$$

$$B_{\overrightarrow{R_v}} \cap B_{\overleftarrow{R_u}} = \emptyset \tag{5.6}$$

We denote the joined route as $R^{join} = \langle s, B_{\overrightarrow{R_v}}, rev(B_{\overleftarrow{R_u}}), d\rangle$, where $rev(B_{\overleftarrow{R_u}})$ refers to a list of vertices in $B_{\overleftarrow{R_u}}$ but in the reversed order. Its reward is: $|B_{\overrightarrow{R_v}}| + |B_{\overleftarrow{R_u}}|$.

Table 5.2. Route join

	sub-routes sorted in the descending order of $\lvert B_R \rvert$
$\overrightarrow{\mathbb{R}}$	$(5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)$, $(3, \langle p_1, p_2 \rangle, \{p_6\}, p_2)$, $(1, \langle p_1 \rangle, \{p_2, p_6\}, p_1)$, $(0, \emptyset, \{p_1, p_2, p_6\}, s)$
$\overleftarrow{\mathbb{R}}$	$(5, \langle p_7, p_3, p_6 \rangle, \emptyset, p_6)$, $(2, \langle p_7, p_3 \rangle, \{p_6\}, p_3)$, $(1, \langle p_7 \rangle, \{p_3, p_6\}, p_7)$, $(0, \emptyset, \{p_3, p_6, p_7\}, d)$

route join iterations

iteration	candidate join pairs		join result	$\$^{best}$
	$\overrightarrow{R}$	$\overleftarrow{R}$	R^{join}	
1	$(5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)$	$(5, \langle p_7, p_3, p_6 \rangle, \emptyset, p_6)$	not feasible (Equation 5.6)	0
2	$(5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)$	$(2, \langle p_7, p_3 \rangle, \{p_6\}, p_3)$	$R = \langle s, p_1, p_2, p_6, p_3, p_7, d \rangle$	5
...	...	...	skipped (Pruning Rule 3)	5
optimal route for this snapshot			$R = \langle s, p_1, p_2, p_6, p_3, p_7, d \rangle$	

We develop two optimization techniques to accelerate the join procedure. First, we apply pruning rule 3 to skip the *feasible* checking (cf. Equation 5.6) for pairs of sub-routes. Second, we sort sub-routes in the descending order of their $|B_R|$. This helps us find a tigher $\$^{best}$ earlier, and in turn boosts the power of Pruning Rule 3.

Pruning Rule 3 (Reward bound pruning)**.** *Let* $\$^{best}$ *be the maximum reward on all joined routes found so far. If* $|B_{\overrightarrow{R}}| + |B_{\overleftarrow{R}}| \leq \$^{best}$*, then we need not join* $\overrightarrow{R}$ *and* $\overleftarrow{R}$*.*

Continuing with the example in Figure 5.6, we illustrate the join procedure in Table 5.2. First, we sort forward sub-routes $\overrightarrow{R} \in \overrightarrow{\mathbb{R}}$ and backward sub-routes $\overleftarrow{R} \in \overleftarrow{\mathbb{R}}$ in descending order of $|B_R|$. For each pair of $\overrightarrow{R}$ and $\overleftarrow{R}$, if it survives Pruning Rule 3, then we conduct feasible checking and then join the pair. After joining the forward sub-route $\overrightarrow{R} = (5, \langle p_1, p_2, p_6 \rangle, \emptyset, p_6)$ with the backward sub-route $\overleftarrow{R}$ $=(2, \langle p_7, p_3 \rangle, \{p_6\}, p_3)$, we update $\$^{best}$ to 5. All remaining pairs are pruned according to Pruning Rule 3. The best route (known at this snapshot) is $\langle s, p_1, p_2, p_6, p_3, p_7, d \rangle$.

5.4 Discussion

In this section, we discuss three enhancements in our proposed Route Recommendation methods: (1) supporting different output formats, (2) recommending routes for multiple workers, and (3) supporting tasks that have non-zero execution times.

Output format of Route Recommender. The best route returned by the Route Recommender is a sequence of spatial crowdsourcing tasks. The worker is advised to conduct tasks in the sequence one by one. Therefore, instead of a complete route, it is also fine to return the next task on the best route each time the spatial crowdsourcing worker finishes a task.

Route recommendation for multiple workers. When recommending routes for multiple spatial crowdsourcing workers, a global assignment will be conducted before recommending the best route for each individual worker. The global assignment will assign tasks to workers fairly. In detail, for each new published task, the global assignment will find a set of workers such that the new task is a feasible candidate for them (according to Equations 5.2 and 5.5). Then, taking fairness into consideration, the new task will be assigned to the worker whose retrieved reward is minimum so far. Each task, after being assigned to a worker w, is treated as locked by w, and will not be assigned to other workers unless it becomes in-feasible for w. With the global assignment, each worker can have a distinct set of tasks, which can avoid conflicts in conducting tasks. Then, the Route Recommender can find the best route for each worker over his assigned

tasks.

Route recommendation of tasks that have non-zero execution times. Let $\tau_{exe}(p)$ be the execution time of task p. Methods discussed in above sections assume that each task can be completed immediately, that is $\tau_{exe}(p) = 0$. In fact, our methods can be easily extended to support the spatial crowdsourcing tasks with $\tau_{exe}(p) > 0$. In detail, the modifications are:

1. When detecting candidate tasks, the Equation 5.2 is modified to:

$$\tau(s_{now}, p) + \tau_{exe}(p) + \tau(p, d) \leq t_q^+ - t_{now}$$
$$t_{now} + \tau(s_{now}, p) + \tau_{exe}(p) \leq t_p^+ \tag{5.7}$$

2. When finding the candidate set in Re-Route, the Equation 5.5 is modified to:

$$p \in C_{R_v} \quad \text{and} \quad p \notin B_{R_u}$$
$$\tau(u, p) + \tau_{exe}(p) \leq t_p^+ - t_q^- - \tau(R_u)$$
$$\tau(u, p) + \tau_{exe}(p) \leq (t_q^+ - t_q^-)/2 - \tau(R_u)$$
$$\tau(u, p) + \tau_{exe}(p) + \tau(p, d) \leq t_q^+ - t_q^- - \tau(R_u) \tag{5.8}$$

3. When extending a route R_v by appending a vertex u into it, the calculation of the travel time for the new generated sub-route R_u in Equation 5.4 is modified to:

$$\tau(R_u) \leftarrow \tau(R_v) + \tau_e(v, u) + \tau_{exe}(u) \tag{5.9}$$

5.5 Experiment

This section studies the effectiveness and efficiency of our proposed methods on both real and synthetic datasets.

5.5.1 Experimental Setting

We first introduce the datasets used in experiments, and then describe the performance measures for algorithms.

Datasets.

Real datasets. Similar to [55], we obtain real check-in data in Foursquare[5.7] and convert them to crowdsourcing tasks in our problem. Specifically, we collect check-in data for New York city (NYC) and Los Angeles County (LA) in a month (September 2012). For each day in that month, we use all check-in items within a 90-minute duration. We take check-in items at the same location as a single task, set its release time and deadline to the earliest and the latest check-in time respectively[5.8]. We measure the travel time $\tau(v, u)$ as the Euclidean distance between two locations divided by the average speed. We use a walking speed 6 km/h for NYC (whose map size 789 km^2 is small), and use a driving speed 60 km/h for LA (whose map size 10,570 km^2 is large).

Synthetic datasets. As NYC and LA have similar result trends (see Figure 5.7), we use the map domain of LA to generate synthetic datasets. For each synthetic task, we randomly choose its release time t_p^- randomly in $[t_q^-, t_q^+]$ and

[5.7] https://foursquare.com/

[5.8] For each location with only one check-in item (say, at time t), we choose its deadline randomly in $[t, t_q^+]$, where t_q^+ refers to the query's deadline.

then choose its deadline t_p^+ in range $[t_p^-, t_q^+]$, as we consider queries of the form $q = (s, d, [t_q^-, t_q^+])$ in our experiments. We generate two types of datasets. In each uniform dataset (UNI), task locations are randomly chosen within the map domain. In each Gaussian dataset (GAU), task locations are generated based on four Gaussian bells, with the standard deviation of Gaussian bell as x times of the map domain length. The parameter values for the number of tasks and Gaussian standard deviation x are shown in Table 5.3.

Platform and Performance Measures. We implemented our methods (G-NN, G-MCS, G-SD, Re-Route) in C++, and conducted experiments on an Ubuntu 11.10 machine with a 3.4 GHz Intel Core i7-3770 processor and 16 GB RAM.

We use queries of the form $q = (s, d, [t_q^-, t_q^+])$, where $t_q^+ - t_q^- = 90$ minutes by default. We randomly choose s, d in the map domain such that $\tau(s, d) = 45$ minutes. The parameter values for $t_q^+ - t_q^-$ are given in Table 5.3.

In each experiment, we run a set Q of 50 queries and report (i) the quality ratio for Q, and (ii) the average response time per call of a method. Specifically, we define the *quality ratio* of a method as:

$$quality \ \ ratio \ = \frac{1}{|Q|} \cdot \sum_{q \in Q} \frac{count(R_{method}(q))}{count(R_{opt}(q))}$$

where q is a query in Q, $R_{method}(q)$ is the route for q found by our method, $R_{opt}(q)$ is the optimal route for q found by an offline method that knows all tasks in advance[5.9].

[5.9] As mentioned in Definition 7, $R_{opt}(q)$ is obtained with assumption that all tasks' information are known in advance at time t_q^-. With this assumption, OnlineRR becomes a special case of SnapshotRR where tasks can have release time larger than t_q^- and the approach for SnapshotRR can be used to find $R_{opt}(q)$ then.

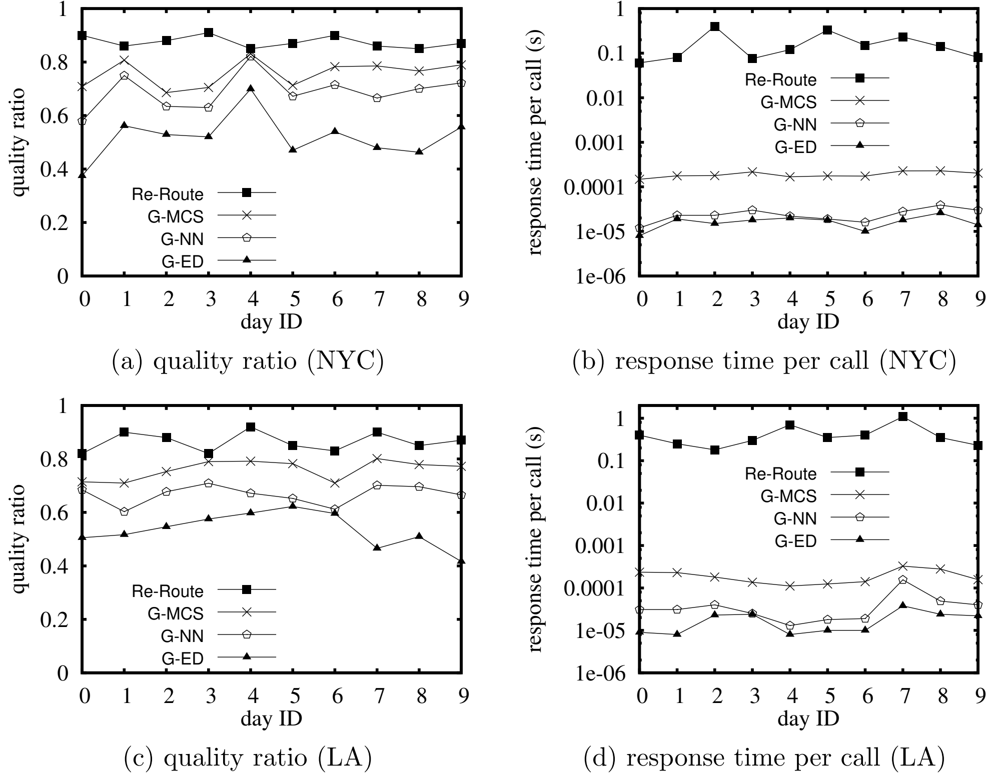
Table 5.3. Experiment parameters

Parameter	Default	Range
total number of tasks	100	$20, 50, 100, 200, 500$
$t_q^+ - t_q^-$ [minutes]	90	$30, 60, 90, 120, 150$
Gaussian x	0.1	$0.05, 0.1, 0.25, 0.5$

We have tested the effects of policies $\mathcal{P}_{stay}$ and $\mathcal{P}_{go}$ (cf. Section 5.2.2) on our methods. For the same method, the quality ratios between $\mathcal{P}_{stay}$ and $\mathcal{P}_{go}$ differ only by $0.01 - 0.02$. Thus, we take the default policy in our methods as $\mathcal{P}_{stay}$.

5.5.2 An Experiment on Real Datasets

We plot the performance of methods on real datasets (LA and NYC) on each day from Sep/21/2012 to Sep/30/2012 in Figure 5.7. Within the query period, LA and NYC contain 60 and 40 tasks on average, respectively. The optimal routes R_{opt} in LA and NYC cover 10 and 5 tasks on average, respectively. Figures 5.7a,c show the quality ratio of the methods on NYC and LA, respectively. Re-Route outperforms other methods and achieves 0.82–0.91 quality. G-MCS is the second best and obtains 0.70–0.84 quality. Although Re-Route incurs higher response time, it takes less than 1 second per call, as depicted in Figure 5.7b,d. We consider such time acceptable for crowdsourcing workers. For example, for the LA dataset, Re-Route is called for 10 times (on average) during the query period (90 minutes). Observe that the time per call (1 second) is negligible compared to the average travel time between two tasks ($90/10 = 9$ minutes).



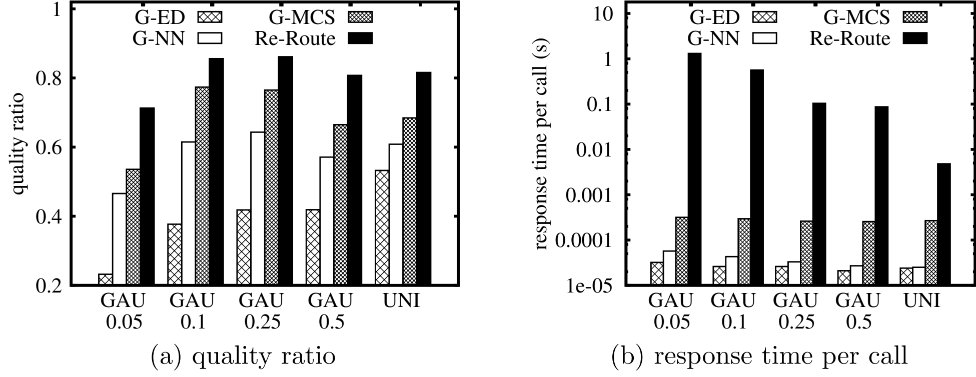
(a) quality ratio (NYC) (b) response time per call (NYC)

(c) quality ratio (LA) (d) response time per call (LA)

Figure 5.7. Performance on real datasets

5.5.3 Scalability Experiments on Synthetic Datasets

Effect of task distribution. Figure 5.8 depicts the performance of methods on GAU datasets with standard deviation x and on a UNI dataset. As illustrated in Table 5.4a, a more skewed dataset (i.e., with smaller x) leads to an optimal route with higher reward because tasks in the same cluster are close together. Since our methods can also find routes with higher reward on a more skewed dataset, the quality ratio does not vary much (See Figure 5.8a). Re-Route again outperforms other methods on the quality ratio. On the other hand, a more skewed dataset induces more feasible candidate tasks in Re-Route, and thus it



(a) quality ratio (b) response time per call

Figure 5.8. Effect of task distribution

Table 5.4. Reward on the optimal route

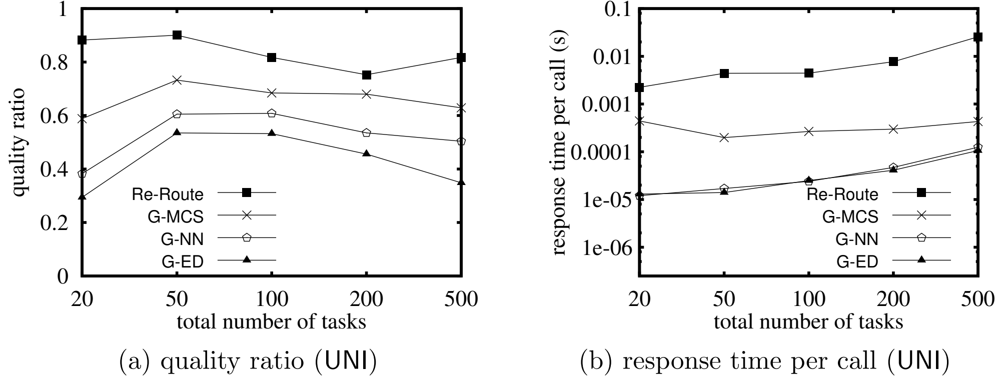
Task distribution	Gaussian	Uniform	
Parameter	**(a)** standard deviation x	**(b)** total number of tasks	**(c)** query period $t_q^+ - t_q^-$
values	0.05, 0.1, 0.25, 0.5	20, 50, 100, 200, 500	30, 60, 90, 120, 150
Reward of R_{opt}	12.57, 9.39, 6.84, 4.72	1.7, 3.14, 5.26, 7.94, 13.2	1.62, 3.26, 5.26, 6.92, 8.92

incurs higher response time. Nevertheless, Re-Route takes at most around 1 second per call in Figure 5.8b, which is acceptable for crowdsourcing workers.

Since the trend on quality is consistent across different task distributions, we only use UNI datasets in the remaining experiments.

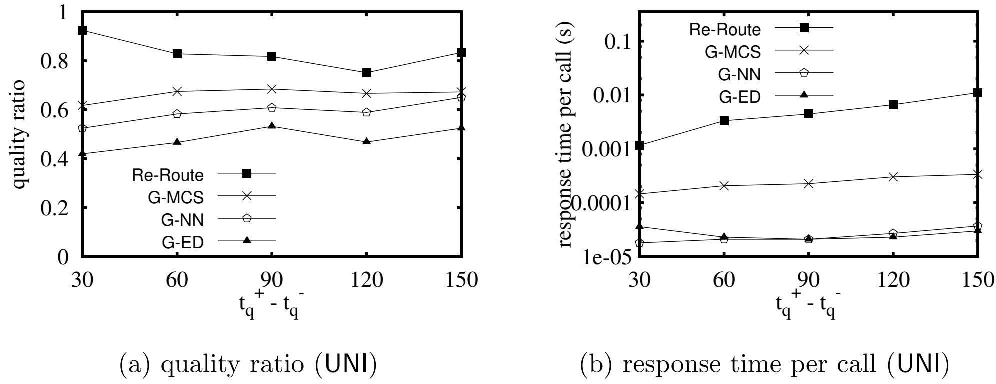
Effect of total number of tasks. When the total number of tasks increases, both the optimal route (cf. Table 5.4b) and our methods' routes would cover more tasks. Thus, the quality ratio is independent of the total number of tasks, as shown in Figure 5.9a. The response time of Re-Route increases slightly with the total number of tasks (see Figure 5.9b), but it is still within 0.1 seconds per call.

Effect of the query period $t_q^+ - t_q^-$. As the query period $t_q^+ - t_q^-$ widens, more



(a) quality ratio (UNI) (b) response time per call (UNI)

Figure 5.9. Effect of the total number of tasks

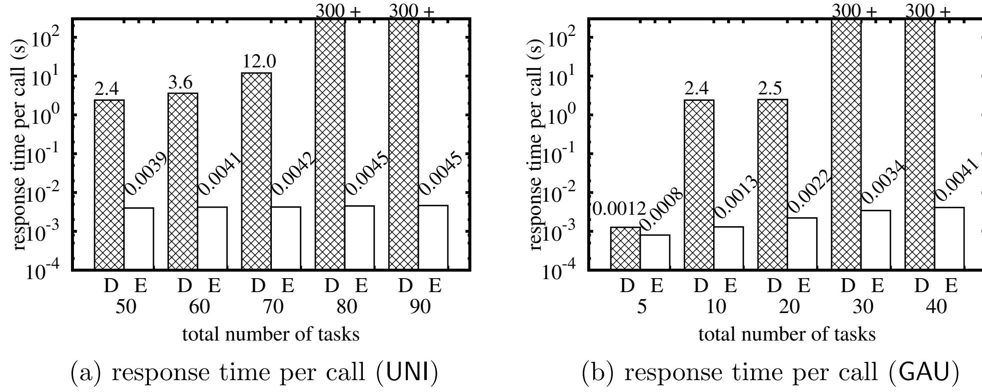


(a) quality ratio (UNI) (b) response time per call (UNI)

Figure 5.10. Effect of the query period $t_q^+ - t_q^-$

tasks become feasible and thus the optimal route contains more tasks, as shown in Table 5.4c. We plot the performance of the methods with respect to $t_q^+ - t_q^-$ in Figure 5.10. The quality ratio is independent of $t_q^+ - t_q^-$ as our methods are also able to find routes with more tasks. The response time per call in Re-Route remains acceptable.

Effect of Pruning Rules on Re-Route. We proceed to test the effect of optimization techniques (cf. Section 5.3) on the response time per call of Re-Route. We consider two variations of Re-Route: (i) DISABLE applies only pruning rule 1



(a) response time per call (UNI) (b) response time per call (GAU)

Figure 5.11. Effect of pruning rules on Re-Route **(E for** ENABLE**, D for** DISABLE**)**

(in Ref. [106]), and (ii) ENABLE applies all three pruning rules in Section 5.3.

As DISABLE is very slow, we scale down the total number of tasks in this experiment, and terminate it if it takes more than 300 seconds per call. We show the response time per call of DISABLE and ENABLE on both UNI and GAU datasets in Figure 5.11. Observe that ENABLE runs much faster than DISABLE, implying that our pruning rules are able to shrink the search space significantly.

5.6 Related Work

The related work relevant to this chapter is described in Section 2.3. This section will present the differences between this chapter and the related work presented in Chapter 2.

Spatial crowdsourcing is an emerging topic in crowdsourcing research. Existing researches are divided into the *server-centric* mode [49,75,76,102,122] and the *worker-centric* mode [31,40,55]. We focus on the latter one as discussed in the introduction. However, [31,40] do not consider the influence of the worker's

travel time, which is critical in our OnlineRR problem. The closest work to ours is [55], which selects a route with the maximum number of tasks for a worker. However, [55] does not discuss how to update a route with respect to online task arrivals. Also, it does not consider the worker's destination and deadline.

Our OnlineRR problem is related to the orienteering problem [66, 126]. The orienteering problem is a variant of the selective traveling salesman problem [60], where (i) not all requests need to be completed, and (ii) the cost is the sum of the total travel time and the penalty of rejected requests. The orienteering problem is well studied [66, 126], but only several works [36, 48, 64, 106] consider the Orienteering Problems with each request having a Time Window (OPTW). Those works focus on the offline scenario but not the online scenario. While there exist approximation algorithms for OPTW offline [36, 48, 64], OnlineRR is an online problem and does not permit any online algorithm to achieve a non-zero competitive ratio.

Righini et al. [106] propose an exact bi-directional search algorithm for OPTW, which can be adapted to solve our SnapshotRR problem. Unlike our solution, this algorithm does not exploit spatial properties to prune unpromising sub-routes. In Section 5.3, we have developed two pruning rules and a search strategy that are specific for SnapshotRR.

Other related route planning problems include the trip planning problem [90] and the optimal sequenced route problem [113]. They require finding the shortest route that passes through specific types of points-of-interests. On the other hand, our problem needs to maximize the number of tasks on a route subject to the tasks' deadlines and the worker's deadline.

OnlineRR problem is also related to online traveling salesman problem (OL-TSP) [34, 72]. Few works have studied OL-TSP with each request having a deadline [37, 128]. While OL-TSP aims to minimize the travel distance, our OnlineRR problem aims to maximize the number of tasks on a route. Moreover, the above works on OL-TSP do not consider the worker's destination and deadline. Finally, our problem is similar to an online job-scheduling problem whose tasks have dependent setup costs [29]. However, this problem does not exploit the spatial properties as in OnlineRR.

5.7 Chapter Summary

In this chapter, we study the oriented online route recommendation (OnlineRR) problem for spatial crowdsourcing task workers. We prove that no online algorithm can achieve a non-zero competitive ratio for OnlineRR. Then we propose several heuristics for OnlineRR and optimizations to speedup the computation. According to our experimental findings, Re-Route produces routes with the highest quality (0.82–0.91) with acceptable response time per call (0.1–1 s), whereas G-MCS returns routes with the second highest quality (0.70–0.84) at real-time (below 1 ms). Workers who prefer to save smartphone battery power may choose to use G-MCS rather than Re-Route as G-MCS has less computation cost and thus costs less battery power.

Chapter 6

Conclusion and Future Work

Web data services provide rich categories of data, which attract more and more consumers. Queries over different web data services have different purposes. In this thesis, we study how to optimize three types of queries over web data services.

6.1 Contributions

The first contribution is that, for SQL queries over data markets, we present a system to help data buyers to freely query against any dataset in the data market and walk away from that dataset anytime. Our experimental results verify that, with our system the data buyers do not need to worry whether it is worth or not to download the whole dataset in the beginning.

The second contribution is that, for the location based services, we propose a solution for them to process spatial range/KNN queries such that the query

results have accurate travel times and the LBS incurs few number of route requests. Our experimental evaluation shows that our method is 3 times more efficient than a competitor, and yet achieves high result accuracy (above 98%).

The third contribution is that, for a spatial corwdsourcing worker, we propose several heuristics to find the optimal route for a worker which can guarantee the worker to arrive at his destination on time as well as enable the worker to finish the maximum number of tasks. According to our experimental findings, our prosed method produces routes with high quality (0.82–0.91) with acceptable response time per call (0.1–1 s).

6.2 Future Research Directions

We proceed to outline several future research directions as follows.

Currently, our use-case in Chapter 3 does not cover many end users using PayLess simultaneously. In the future, we will incorporate multi-query optimization in PayLess if users are willing to defer their queries to become a batch.

Our Route-Saver in Chapter 4 for LBS utilizes δ to control the freshness of the route log. As a future work, we plan to investigate automatic tuning the expiry time δ based on a given accuracy requirement. This would help the LBS guarantee its accuracy and improve their users' satisfaction.

For the route recommendation OnlineRR in Chapter 5, a possible research direction is to consider the diversity and novelty of tasks during the path selection, and another direction is to extend it to support road networks with dynamic traffic.

Bibliography

[1] 2011 Census TIGER/Line Shapefiles. http://www.census.gov/cgi-bin/geo/shapefiles2011/main.

[2] 9th DIMACS Implementation Challenge on Shortest Paths. http://www.dis.uniroma1.it/challenge9/data/tiger/.

[3] Amazon mechanical turk. https://www.mturk.com/mturk/.

[4] Azure data marketplace. https://datamarket.azure.com.

[5] Bing Data Suppliers. http://windows.microsoft.com/en-HK/windows-live/about-bing-data-suppliers/.

[6] Bing Maps API. http://www.microsoft.com/maps/developers/web.aspx.

[7] Bing Maps Licensing and Pricing Information. http://www.microsoft.com/maps/product/licensing.aspx.

[8] Data markets compared. http://radar.oreilly.com/2012/03/data-markets-survey.html.

[9] data.gov. https://www.data.gov/.

[10] Datamarket. https://datamarket.com/.

[11] Discussing data markets in new york city. http://cloudofdata.com/2013/02/discussing-data-markets-in-new-york-city/.

[12] Environmental hazard rank data. http://datamarket.azure.com/dataset/edr/environmentalhazardrank.

[13] Esri. http://www.esri.com/.

[14] Factual. http://www.factual.com/.

[15] Google Directions & Bing Maps: Live Traffic Information. http://support.google.com/maps/bin/answer.py?hl=en&answer=2549020&topic=1687356&ctx=topic http://msdn.microsoft.com/en-us/library/aa907680.aspx.

[16] Google Directions API. https://developers.google.com/maps/documentation/directions/.

[17] Google Directions API Usage Limits. https://developers.google.com/maps/faq#usagelimits.

[18] Google Map Maker Data Download. https://services.google.com/fb/forms/mapmakerdatadownload/.

[19] Infochimps. http://www.infochimps.com.

[20] Is infochimps running from the data market business? http://cloudofdata.com/2013/02/is-infochimps-running-from-the-data-market-business/.

[21] OpenStreetMap. http://www.openstreetmap.org/.

[22] Statistics of Usage. http://www.quantcast.com.

[23] US Maps from Government. http://www.usgs.gov/pubprod/.

[24] Wolfram alpha. http://www.wolframalpha.com/.

[25] World bank. http://www.worldbank.org/.

[26] Worldwide historical weather data. https://datamarket.azure.com//dataset//weathertrends//worldwidehistoricalweatherdata.

[27] Xignite data market. http://www.xignite.com.

[28] Samer Al-Kiswany, Hakan Hacigümüs, Ziyang Liu, and Jagan Sankaranarayanan. Cost exploration of data sharings in the cloud. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 601–612, 2013.

[29] Ali Allahverdi, C. T. Ng, T. C. Edwin Cheng, and Mikhail Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, 2008.

[30] Borodin Allan and El-Yaniv Ran. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[31] Florian Alt, Alireza Sahami Shirazi, Albrecht Schmidt, Urs Kramer, and Zahid Nawaz. Location-based crowdsourcing: extending crowdsourcing to the real world. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction 2010, Reykjavik, Iceland, October 16-20, 2010*, pages 13–22, 2010.

[32] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 821–831, 2003.

[33] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 493–504. IEEE Computer Society, 2003.

[34] Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, 29(4):560–581, 2001.

[35] Magdalena Balazinska, Bill Howe, and Dan Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 4(12):1482–1485, 2011.

[36] Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Adam Meyerson. Approximation algorithms for deadline-tsp and vehicle routing with time-windows. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 166–174, 2004.

[37] Michiel Blom, Sven O. Krumke, Willem E. De Paepe, and Leen Stougie. The online TSP against fair adversaries. *INFORMS Journal on Computing*, 13(2):138–148, 2001.

[38] Daniele Braga, Stefano Ceri, Florian Daniel, and Davide Martinenghi. Optimization of multi-domain queries on the web. *Proc. VLDB Endow.*, 1(1):562–573, 2008.

[39] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 211–222, 2001.

[40] Muhammed Fatih Bulut, Yavuz Selim Yilmaz, and Murat Demirbas. Crowdsourcing location-based queries. In *Ninth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2011, 21-25 March 2011, Seattle, WA, USA, Workshop Proceedings*, pages 513–518, 2011.

[41] Andrea Calì, Diego Calvanese, and Davide Martinenghi. Dynamic query optimization under access limitations and dependencies. *J. UCS*, 15(1):33–62, 2009.

[42] Andrea Calì and Davide Martinenghi. Querying data under access limitations. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 50–59, 2008.

[43] Andrea Calì and Davide Martinenghi. Querying the deep web. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426, pages 724–727, 2010.

[44] Edward P. F. Chan and Yaya Yang. Shortest path tree computation in dynamic graphs. *IEEE Trans. Computers*, 58(4):541–557, 2009.

[45] S. Chaudhuri and V. Narasayya. Program for tpc-d data generation with skew. 2012.

[46] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43, 1998.

[47] Surajit Chaudhuri and Kyuseok Shim. Query optimization in the presence of foreign functions. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings.*, pages 529–542, 1993.

[48] Chandra Chekuri and Nitish Korula. Approximation algorithms for orienteering with time windows. *CoRR*, abs/0711.4825, 2007.

[49] Zhao Chen, Rui Fu, Ziyuan Zhao, Zheng Liu, Leihao Xia, Lei Chen, Peng Cheng, Caleb Chen Cao, Yongxin Tong, and Chen Jason Zhang. gmission: A general spatial crowdsourcing platform. In *PVLDB*, pages 1629–1632, 2014.

[50] Boris Chidlovskii and Uwe M. Borghoff. Semantic caching of web queries. *VLDB J.*, 9(1):2–17, 2000.

[51] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979.

[52] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[53] Shaul Dar, Michael J. Franklin, Björn THór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 330–341, 1996.

[54] Ugur Demiryurek, Farnoush Banaei Kashani, Cyrus Shahabi, and Anand Ranganathan. Online computation of fastest path in time-dependent spatial networks. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings*, volume 6849, pages 92–111, 2011.

[55] Dingxiong Deng, Cyrus Shahabi, and Ugur Demiryurek. Maximizing the number of worker's self-selected tasks in spatial crowdsourcing. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*, pages 314–323, 2013.

[56] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.

[57] Adam Dingle and Tomas Pártl. Web cache coherence. *Computer Networks*, 28(7-11):907–920, 1996.

[58] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 2009.

[59] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 778–784, 1997.

[60] Horst A. Eiselt, Michel Gendreau, and Gilbert Laporte. Location of facilities on a network subject to a single-edge failure. *Networks*, 22(3):231–246, 1992.

[61] Amr El-Helw, Ihab F. Ilyas, Wing Lau, Volker Markl, and Calisto Zuzarte. Collecting and maintaining just-in-time statistics. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 516–525, 2007.

[62] David Eppstein. Graph-theoretic solutions to computational geometry problems. In *Graph-Theoretic Concepts in Computer Science, 35th International Workshop, WG 2009, Montpellier, France, June 24-26, 2009. Revised Papers*, volume 5911, pages 1–16, 2009.

[63] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 311–322, 1999.

[64] Greg N. Frederickson and Barry Wittman. Approximation algorithms for the traveling repairman and speeding deliveryman problems. *Algorithmica*, 62(3-4):1198–1221, 2012.

[65] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[66] Damianos Gavalas, Charalampos Konstantopoulos, Konstantinos Mastakas, and Grammati E. Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *J. Heuristics*, 20(3):291–328, 2014.

[67] Anastasios Gounaris, Christos A. Yfoulis, Rizos Sakellariou, and Marios D. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 596–605, 2008.

[68] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[69] Haibo Hu, Dik Lun Lee, and Victor C. S. Lee. Distance indexing on road networks. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 894–905, 2006.

[70] Haibo Hu, Jianliang Xu, Wing Sing Wong, Baihua Zheng, Dik Lun Lee, and Wang-Chien Lee. Proactive caching for spatial queries in mobile environments. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 403–414, 2005.

[71] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. Location-dependent query processing: Where we are and where we are heading. *ACM Comput. Surv.*, 42(3), 2010.

[72] Patrick Jaillet and Michael R. Wagner. Online routing problems: Value of advanced information as improved competitive ratios. *Transportation Science*, 40(2):200–210, 2006.

[73] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

[74] Evangelos Kanoulas, Yang Du, Tian Xia, and Donghui Zhang. Finding fastest paths on A road network with speed patterns. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 10, 2006.

[75] Leyla Kazemi and Cyrus Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *SIGSPATIAL 2012 International Conference on Advances in Geographic Information Systems (formerly known as GIS), SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*, pages 189–198, 2012.

[76] Leyla Kazemi, Cyrus Shahabi, and Lei Chen. Geotrucrowd: trustworthy query answering with spatial crowdsourcing. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*, pages 304–313, 2013.

[77] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based K nearest neighbor search for spatial network databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 840–851, 2004.

[78] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[79] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 167–178, 2012.

[80] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Toward practical query pricing with querymarket. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 613–624, 2013.

[81] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, and Matthias Renz. Generalizing the optimality of multi-step k -nearest neighbor query processing. In *Advances in Spatial and Temporal Databases, 10th International Symposium, SSTD 2007, Boston, MA, USA, July 16-18, 2007, Proceedings*, volume 4605, pages 75–92, 2007.

[82] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Tim Schmidt . Hierarchical graph embedding for efficient query processing in very large traffic

networks. In *Scientific and Statistical Database Management, 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008, Proceedings*, volume 5069, pages 150–167, 2008.

[83] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Tim Schmidt. Proximity queries in large traffic networks. In *15th ACM International Symposium on Geographic Information Systems, ACM-GIS 2007, November 7-9, 2007, Seattle, Washington, USA, Proceedings*, page 21, 2007.

[84] Dongwon Lee and Wesley W. Chu. Towards intelligent semantic caching for web sources. *J. Intell. Inf. Syst.*, 17(1):23–45, 2001.

[85] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Jianliang Xu. Caching complementary space for location-based services. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, pages 1020–1038, 2006.

[86] Chen Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3):211–227, 2003.

[87] Chen Li and Edward Y. Chang. Query planning with limited source capabilities. In *IEEE ICDE*, pages 401–412, 2000.

[88] Chen Li and Edward Y. Chang. Answering queries with useful bindings. *ACM Trans. Database Syst.*, pages 313–343, 2001.

[89] Chen Li and Edward Y. Chang. On answering queries in the presence of limited access patterns. In *Database Theory - ICDT 2001, 8th International*

Conference, London, UK, January 4-6, 2001, Proceedings., pages 219–233. Springer, 2001.

[90] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, volume 3633, pages 273–290, 2005.

[91] Yu Li, Eric Lo, Man Lung Yiu, and Wenjian Xu. Query optimization over cloud data market. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 229–240, 2015.

[92] Yu Li and Man Lung Yiu. Route-saver: Leveraging route apis for accurate and efficient query processing at location-based services. *IEEE Trans. Knowl. Data Eng.*, 27(1):235–249, 2015.

[93] Yu Li, Man Lung Yiu, and Wenjian Xu. Oriented online route recommendation for spatial crowdsourcing task workers. In *submitted to SSTD.*

[94] Ziyang Liu and Hakan Hacigümüs. Online optimization and fair costing for dynamic data sharing in a cloud data market. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1359–1370, 2014.

[95] Ioana Manolescu, Luc Bouganim, Françoise Fabret, and Eric Simon. Efficient querying of distributed resources in mediator systems. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE*

2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings, volume 2519, pages 468–485, 2002.

[96] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 659–670, 2004.

[97] Alexander Muschalle, Florian Stahl, Alexander Löser, and Gottfried Vossen. Pricing approaches for data markets. In *Enabling Real-Time Business Intelligence - 6th International Workshop, BIRTE 2012, Held at the 38th International Conference on Very Large Databases, VLDB 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers*, volume 154, pages 129–144, 2012.

[98] Alan Nash and Bertram Ludäscher. Processing first-order queries under limited access patterns. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 307–318, 2004.

[99] Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992, pages 422–440, 2004.

[100] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[101] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial databases containing obstacles. *International Journal of Geographical Information Science*, 19:1091–1111, 2005.

[102] Layla Pournajaf, Li Xiong, Vaidy S. Sunderam, and Slawomir Goryczka. Spatial task assignment for crowd sensing with cloaked locations. In *MDM*, pages 73–82, 2014.

[103] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Trans. Knowl. Data Eng.*, 26(1):55–68, 2014.

[104] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 105–112, 1995.

[105] Qun Ren, Margaret H. Dunham, and Vijay Kumar. Semantic caching and query processing. *IEEE Trans. Knowl. Data Eng.*, 15(1):192–210, 2003.

[106] Giovanni Righini and Matteo Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Computers & OR*, 36(4):1191–1203, 2009.

[107] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2011.

[108] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 43–54, 2008.

[109] Jagan Sankaranarayanan and Hanan Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1158–1175, 2010.

[110] Fabian Schomm, Florian Stahl, and Gottfried Vossen. Marketplaces for data: an initial survey. *SIGMOD Record*, 42(1):15–26, 2013.

[111] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 154–165, 1998.

[112] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 435–446, 1996.

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

[114] David B. Shmoys, Alexander H. G. Rinnooy Kan, Jan K. Lenstra, and Eugene L. Lawler. *The Traveling salesman problem : a guided tour of combinatorial optimization*. J. Wiley and sons, 1987.

[115] Wouter Souffriau and Pieter Vansteenwegen. Tourist trip planning functionalities: State-of-the-art and future. In *Current Trends in Web Engineering - 10th International Conference on Web Engineering, ICWE 2010 Workshops, Vienna, Austria, July 2010, Revised Selected Papers*, volume 6385, pages 474–485, 2010.

[116] Utkarsh Srivastava, Peter J. Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. ISOMER: consistent histogram construction using query feedback. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 39, 2006.

[117] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query optimization over web services. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 355–366, 2006.

[118] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - db2's learning optimizer. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28, 2001.

[119] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB 2002, Proceedings of 28th International Confer-*

ence on Very Large Data Bases, August 20-23, 2002, Hong Kong, China, pages 287–298, 2002.

[120] Yufei Tao, Jimeng Sun, and Dimitris Papadias. Selectivity estimation for predictive spatio-temporal queries. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 417–428, 2003.

[121] Jeppe Rishede Thomsen, Man Lung Yiu, and Christian S. Jensen. Effective caching of shortest paths for location-based services. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 313–324, 2012.

[122] Hien To, Gabriel Ghinita, and Cyrus Shahabi. A framework for protecting worker location privacy in spatial crowdsourcing. *PVLDB*, 7(10):919–930, 2014.

[123] Goce Trajcevski and Peter Scheuermann. Triggers and continuous queries in moving objects database. In *14th International Workshop on Database and Expert Systems Applications (DEXA'03), September 1-5, 2003, Prague, Czech Republic*, pages 905–910, 2003.

[124] Goce Trajcevski, Ouri Wolfson, Klaus H. Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004.

[125] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu. How to price shared optimizations in the cloud. *PVLDB*, 5(6):562–573, 2012.

[126] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011.

[127] Pieter Vansteenwegen and Dirk Van Oudheusden. The mobile tourist guide: An OR opportunity. *OR Insight*, 20(3):21–27, 2007.

[128] Xingang Wen, Yinfeng Xu, and Huili Zhang. Online traveling salesman problem with deadline and advanced information. *Computers & Industrial Engineering*, 63(4):1048–1053, 2012.

[129] H. Z. Yang and Per-Åke Larson. Query transformation for psj-queries. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 245–254, 1987.

[130] Qi Yu and Athman Bouguettaya. Framework for web service query algebra and optimization. *ACM Trans. Web*, 2(1):6:1–6:35, 2008.

[131] Detian Zhang, Chi-Yin Chow, Qing Li, Xinming Zhang, and Yinlong Xu. Efficient evaluation of k-nn queries using spatial mashups. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings*, volume 6849, pages 348–366, 2011.

[132] Detian Zhang, Chi-Yin Chow, Qing Li, Xinming Zhang, and Yinlong Xu. Smashq: spatial mashup framework for k-nn queries in time-dependent road networks. *Distributed and Parallel Databases*, 31(2):259–287, 2013.

[133] Baihua Zheng and Dik Lun Lee. Semantic caching in location-dependent query processing. In *Advances in Spatial and Temporal Databases, 7th*

International Symposium, SSTD 2001, Redondo Beach, CA, USA, July 12-15, 2001, Proceedings, pages 97–116, 2001.

[134] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: an efficient index for KNN search on road networks. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 39–48, 2013.