

M.Sc. Engg. (CSE) Thesis

**AN EFFICIENT TECHNIQUE FOR ONLINE
CROWDSHIPPING BASED PACKAGE DELIVERY
WITH DAILY COMMUTERS**

Submitted by

Shadman Saqib Eusuf

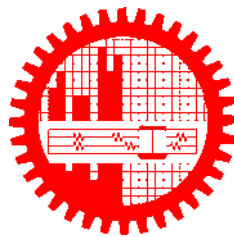
1017052009

Supervised by

Dr. Rifat Shahriyar

Co-Supervised by

Dr. Mohammed Eunos Ali



Submitted to

**Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh**

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

November 2022

Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "AN EFFICIENT TECHNIQUE FOR ONLINE CROWDSIPPING BASED PACKAGE DELIVERY WITH DAILY COMMUTERS", is the outcome of the investigation and research carried out by me under the supervision of Dr. Rifat Shahriyar, Professor, and co-supervised by Dr. Mohammed Eunos Ali, Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

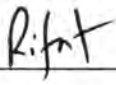
Shadman Saqib Eusuf

Shadman Saqib Eusuf

1017052009

The thesis titled "AN EFFICIENT TECHNIQUE FOR ONLINE CROWDSHIPPING BASED PACKAGE DELIVERY WITH DAILY COMMUTERS", submitted by Shadman Saqib Eusuf, Student ID 1017052009, Session October 2022, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on November 29, 2022.

Board of Examiners

1. 

Dr. Rifat Shahriyar
Professor
Department of CSE, BUET, Dhaka
Chairman
(Supervisor)
2. 

Dr. Mohammed Eunus Ali
Professor
Department of CSE, BUET, Dhaka
Member
(Co-Supervisor)
3. 

Dr. Mahmuda Naznin
Professor and Head
Department of CSE, BUET, Dhaka
Member
(Ex-Officio)
4. 

Dr. Md. Saidur Rahman
Professor
Department of CSE, BUET, Dhaka
Member
5. 

Dr. Md. Shamsuzzoha Bayzid
Associate Professor
Department of CSE, BUET, Dhaka
Member
6. 

Dr. M. Kaykobad
Distinguished Professor
Department of CSE
Brac University
Dhaka
Member
(External)

Acknowledgement

First of all, I am thankful to the Almighty Allah for keeping me healthy and active so that I could finish the work.

I am thankful to my thesis supervisors Prof. Dr. Rifat Shahriyar and Prof. Dr. Mohammed Eunos Ali for their support and encouragement. Specially, my sincere gratitude goes to Prof. Dr. Mohammed Eunos Ali, for his continuous supervision and guidance that directed me throughout this bittersweet long journey of M.Sc. Engg. thesis. His immense knowledge, valuable advices and constant motivation helped me overcome countless hurdles. I am really thankful to him for his exceptional patience with me.

I would also like to thank my teachers and colleagues for their helpful attitude throughout this difficult time and my friends for their selfless help numerous times. Finally, my deep gratitude goes to my parents, my wife and all my family members for their support, encouragement, attention and patience during my studies.

Dhaka
November 29,
2022

Shadman Saqib Eusuf
1017052009

Contents

Candidate’s Declaration	i
Board of Examiners	ii
Acknowledgement	iii
List of Figures	vi
List of Tables	vii
	viii
Abstract	ix
1 Introduction	1
1.1 Motivation	1
1.1.1 A Map Based Example	2
1.2 State of the Art	3
1.3 Modeling	4
1.4 Solution Overview	4
1.4.1 Challenges	4
1.4.2 Pruning with Two Indexes	5
1.5 Contributions	5
1.6 Thesis Organization	5
2 Background and Related Works	7
2.1 Background	7
2.1.1 Introduction to Spatio-temporal Database	7
2.1.2 Traditional Spatial Indexes	8
2.2 Related Works	12
2.2.1 Crowdsourced Package Delivery	12
2.2.2 Spatio-temporal Indexes	14
2.2.3 Trajectory Matching	16

3	Problem Formulation	18
3.1	Terms and Notations	18
3.2	Delivery Cost Function	19
3.3	Formal Definition of Best Packet Delivery Queries	20
4	Modeling and Baseline	21
4.1	Modeling User Trajectories	21
4.2	Base Solution using Path Finding Algorithm	23
4.3	Best First Informed Search	24
4.4	Processing m -BDPT	25
5	An Efficient Index Based Solution	26
5.1	SQ-index	26
5.2	Best First Exploration with SQ-index	29
5.3	Spatio Temporal Co-Visit Index (STCoV-index)	30
5.3.1	Quadtree Based Partition	33
5.3.2	Spatio-temporal Transformation	34
5.3.3	R-tree Based Grouping	34
6	Experimental Evaluation	36
6.1	Experimental Settings	36
6.1.1	Trajectory Datasets	37
6.1.2	Performance Evaluation and Parameterization	38
6.1.3	Experimental Results on Myki Dataset	39
6.1.4	Experimental Results on NYC Dataset	42
7	Conclusion	47
	References	48

List of Figures

1.1	An Example of Crowd-source Package Delivery using Daily Commuters	2
2.1	Spatial objects distributed in the space	9
2.2	R-tree constructed with spatial objects of Figure 2.1	9
2.3	Quadtree constructed with spatial objects of Figure 2.1	10
2.4	Morton ordering of the quadtree nodes constructed with spatial objects of Figure 2.1	11
4.1	Packet delivery example using <i>TrajGraph</i>	22
5.1	SQ-index: Summary Quadtree	27
5.2	SQ-index: (a) Outgoing Blocks (b) Incoming Blocks	27
5.3	(a) Spatial Partitions, and (b) Temporal Buckets of STCoV-index	31
5.4	Trajectory Grouping using STCoV-index	32
5.5	Index Structure of STCoV-index	32
6.1	Evaluating BDPT for varying number of trajectories in MM dataset	39
6.2	Evaluating BDPT for varying % of packet keepers in MM dataset	40
6.3	Evaluating BDPT for varying detour distance (δ) in MM dataset	41
6.4	Evaluating BDPT for varying packet duration in MM dataset	41
6.5	Evaluating packet delivery rate of BDPT for varying the percentage of keepers, the detour threshold δ , and the packet duration in MM dataset	42
6.6	Evaluating cost of successful deliveries of BDPT for varying the percentage of keepers, the detour threshold δ , and the packet duration in MM dataset	42
6.7	Evaluating BDPT for varying number of trajectories in NYC dataset	43
6.8	Evaluating BDPT for varying % of packet keepers in NYC dataset	43
6.9	Evaluating BDPT for varying detour distance (δ) in NYC dataset	44
6.10	Evaluating BDPT for varying packet duration in NYC dataset	44
6.11	Evaluating packet delivery rate of BDPT for varying the number of trajectories, the detour threshold δ , and the packet duration in NYC dataset	45
6.12	Evaluating cost of successful deliveries of BDPT by varying the number of trajectories, the detour threshold δ , and the packet duration in NYC dataset	45

List of Tables

6.1 Parameters for Experiments on Myki and NYC Datasets	38
---	----

List of Algorithms

1	<code>findDeliverer(P_{req}, $TrajGraph$)</code>	23
2	<code>minWA*Path($start$, $nodeList$, $TrajGraph$)</code>	24
3	<code>SQReduce (P_{req}, $SQ-index$)</code>	30

Abstract

The advancement of location-aware technologies enables generating an unprecedented amount of trajectories representing the daily commuting patterns of dwellers in a city. A wide variety of location-based services have started capitalizing on these spatio-temporal footprints of users in enhancing existing services and developing new services. In this thesis, we propose a new location-based service, namely *crowdshipping*, that enables a service delivery company to exploit users' daily commuting patterns to deliver a packet from one place to another using crowd. In particular, our proposed service engages users in shipping goods near their regular itinerary (with a small detour) while minimizing the total cost of the delivery. We take into account the commuters' choice of transport and the involvement of multiple commuters in delivering a package. A major challenge in solving such a query is to select a set of candidate trajectories (i.e., users) from a large trajectory database that can deliver a packet with minimum cost. To address this challenge, we propose a solution based on two indexes. We first build a summary index to capture the overall commuting patterns of the users in the space. This index sets up a regional connectivity network with the trajectories passing through them, which helps us to identify the initial search space for a package to be delivered. We then use a second index by grouping the trajectories based on their spatio-temporal co-visiting patterns. It helps prune the trajectories in temporal domain while searching for an answer. Besides, it helps group the trajectories with spatial and temporal locality together in the physical disk pages. To evaluate our proposed approach we compare it with a baseline based on a traditional spatial index (quadtree) on large real-world trajectory datasets. Experiments show that our efficient index performs an order of magnitude better than the baseline on the real data both in terms of runtime and I/O cost.

Chapter 1

Introduction

1.1 Motivation

With the widespread use of the GPS technology over the last decade, millions of personalized trajectories (i.e., spatio-temporal footprints) are produced everyday from various platforms like transportation companies, map-based services, social media, and so on. For instance, people nowadays share their daily commuting trips of Uber and Bikely besides sharing real-time location using map services like Google Maps. This generates huge volumes of trajectories, representing the daily commuting patterns of millions of individuals, specially in large cities. The overwhelming amount of trajectories can potentially facilitate many interesting location-based services including ridesharing [1], public transport route construction [2], driving route recommendation [3], crowdsourcing based task assignment [4, 5] and so on. One of the key location-based services that can be benefited from these daily commuting trajectories is the package or parcel delivery service. It has vast application encompassing but not limited to online shopping, meal delivery service, courier service and so on.

We have seen an unprecedented explosion of package delivery services in recent years. Amazon shipped 1.9 billion packages in 2019 followed by 4.2 billion packages in 2020 in US only [6]. Besides, FedEx, one of the leading logistic companies in US, shipped approximately 2.1 million packages in the last fiscal year [7] making a revenue of 93.51 billion dollars [8]. Likewise, Uber made a revenue of approximately 5.2 billion dollars from its delivery services only in the first half of 2022 [9, 10]. These evidences indicate that the target market of parcel delivery is huge and thus an efficient approach to it has a commercially good potential. In this thesis, we propose a new location-based service, namely, *crowdshipping*, that enables a service delivery company to exploit users' daily commuting patterns to deliver a package (also often called packet in this literature).

Suppose a customer wants to deliver a package from an origin to a destination through a delivery service provider that is using the crowd (daily commuters) to ship packages. Commuters may

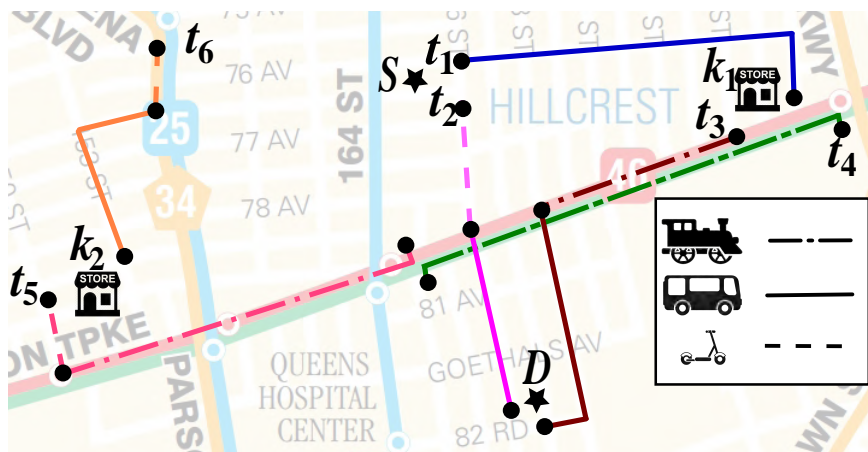


Figure 1.1: An Example of Crowd-source Package Delivery using Daily Commuters

have a regular travel pattern and can take several transports for traveling from one place to another. A commuter may only be interested in participating in the package delivery if she does not need to deviate more than a certain distance from her usual route. Also since it may not be possible for a single commuter to complete the end-to-end delivery of the package, engaging multiple commuters in the delivery of a single package may be required. A package, in its delivery life cycle, can be stored temporarily by a keeper at an intermediate location (e.g., in a smart locker at a train station or in a shop that chooses to participate in the delivery process), where keepers can receive the package from one deliverer, store it for some time, and hand it over to another deliverer or the receiver. In this process, the delivery service provider needs to find the route for the package combining one or more deliverers and keepers that minimizes the cumulative delivery cost and delivers the package within a stipulated time.

1.1.1 A Map Based Example

Consider the example in Figure 1.1. There are six commuter trajectories denoted as t_1 to t_6 and two intermediate packet storage locations (i.e., keepers), k_1 and k_2 . The transport modes along the trajectory segments are represented by lines in different styles (see figure legend). For instance, commuter t_2 travels by e-scooter followed by bus along her trajectory. Suppose that a customer wants to send a packet from S to D within a deadline of two hours. So, we need to find a path comprising of a set of trajectories (commuters) and packet keepers to deliver the packet in the minimum possible cost without violating the delivery deadline. For simplicity, assume we model the cost in terms of total delivery duration. In this example, there are two possible paths for the packet to reach its destination: i) t_2 can pick it up from S and drop it off at D since she travels close to both the packet source and the destination; ii) t_1 can pick up the packet from S , drop it off at k_1 , and, t_3 can pick it up from k_1 and take it to D . In the latter case, the keeper k_1 stores the packet temporarily until t_3 picks it up from there. This relaxes the constraint

of both the commuters to be at the same location at the exact same time for packet handover. Now between the choices of $\{t_2\}$ and $\{t_1, k_1, t_3\}$ to deliver the packet, we choose the one with the minimum duration meeting the delivery deadline. Assume that t_2 will start her journey 30 minutes after the packet is available at S and takes 20 minutes of e-scooter ride followed by a 10 minutes bus ride totalling to **60 minutes (30 + 20 + 10)** for delivering the packet from S to D . On the other hand, say, t_1 starts 5 minutes after the packet is available at S and takes 10 minutes bus ride to ship the packet from S to k_1 . So, the total time t_1 takes to ship the packet is 15 minutes (5 + 10). Then k_1 keeps it for 5 minutes before t_3 picks it up from there. Finally, t_3 takes a 5 minutes train ride followed by a 10 minutes bus ride to drop it at D , totalling to 15 minutes. So, the total delivery time in this path is **35 minutes (15 + 5 + 15)** for the packet delivery. As t_2 incurs higher cost taking 60 minutes compared to 35 minutes via $\{t_1, k_1, t_3\}$ for the packet delivery, the best option is to choose $\{t_1, k_1, t_3\}$.

1.2 State of the Art

The traditional package delivery services usually involve dedicated delivery persons to achieve a delivery task. For instance, Uber, Amazon and Foodpanda make contract with deliverers who work solely for delivering parcels of their customers [11–13]. This requires the delivery persons to make extra movements according to the delivery locations. Although Amazon Flex and Walmart have recently introduced crowdshipped delivery services [14] besides their dedicated delivery wing, they do not match their delivery requests with the commuting paths of the participants. On the other hand, our goal is to engage daily commuters in parcel delivery by reusing their regular itinerary pattern and requiring little extra movement. Here, we briefly discuss existing works on crowdshipping and their limitations.

Existing works on crowdshipping either process the query offline or solve it in a unimodal transportation scenario, e.g., taxi, for a few OD (origin-destination) pairs. Most of the studies that address the crowdshipping problem usually formulate it as an integer linear programming (ILP) problem [15, 16] and provide offline solutions. For instance, given a set of delivery tasks, a set of OD pairs for the vehicles, and a set of constraints, Arslan et al. [16] aim at maximizing the total profit using an ILP formulation and solve the ILP using a batch-wise recursive approach of assigning sets of tasks to the vehicles. Similarly, Chen et al. [15] formulates the ILP given a set of agent routes and task nodes subject to two sets of constraints. They solve it using a greedy heuristic that assigns a task to an agent having the least travel time for detour. Likewise, some works [14, 17] model crowdshipping as a vehicle routing problem with occasional drivers and solve it by adapting a mixed integer programming formulation.

Recently, Chen et al. [18] have modeled the package delivery problem as an arriving on time problem where they construct a package transport network, consisting of interchange stations and edges connecting them, as a subset of the available road network from taxi pick-up drop-off

pairs only. The major limitations of existing works are as follows: i) Integer Linear Programming based solutions (e.g., [16, 17]) are offline in nature and are not scalable for large user base and thus are not suitable for many real-life applications; and ii) the query-based solution such as the approach in [18] only consider taxi pick-up and drop-off points to design a package delivery network. More importantly none of the existing approaches exploits user commuting trajectories to include a wide user base in crowdshipping, which is the main focus of this thesis.

1.3 Modeling

To alleviate the constraints of limited spatial coverage and the offline nature of the solution, we formulate the crowdshipping problem as an online location-based query service. Given the available commuter trajectories, our proposed crowdshipping query aims at finding a minimum cost delivery path for a package given its source, destination and a delivery time window which spans from the earliest time the package is available at the source to the deadline of shipping it to the destination.

We model the query as a trajectory matching problem. As a baseline solution, we first construct a graph by mapping the trajectory points spatially to the road network and augmenting with their temporal attributes. We also map the keepers to the road network nodes and add bidirectional edges to support detour from the nearby trajectories. We then run an informed search (e.g., A* search [19]) to find a path for a given package delivery request. Since the number of nodes may be large for their spatio-temporal representation and the number of edges may be large for retaining travel information like time, duration, transport, detour etc., the graph can be huge compared to the static road network graph. We cannot efficiently prune the candidates for this graph construction with traditional spatial indexes. As a result, path finding in this graph may be intractable. Moreover, the graph needs to be updated with the addition of new trajectories.

1.4 Solution Overview

1.4.1 Challenges

A key challenge in efficiently answering the crowdshipping query lies in quickly finding, from a large trajectory database, the trajectories that have orientation and spatio-temporal locality similar to the package delivery request. Since there can be millions of commuters in a city, we need to process millions of trajectories while answering the crowdshipping query, which is a major obstacle to a scalable and practical solution. We need to match the trajectories with the packet origin-destination with a cost minimization objective and join the overlapping ones via the keepers allowing a detour threshold. Thus, we need to organize them in such a way that not

only helps prune the irrelevant trajectories, but also ensures minimum disk access to answer the query.

1.4.2 Pruning with Two Indexes

To alleviate the challenge of pruning trajectories, we want to retrieve only those trajectories which can contribute to a delivery task. We design two different indexes: i) the first index helps to summarize the spatial connectivity using a quadtree to identify a set of candidate trajectories that most likely contribute to delivering a packet; and ii) the second index groups the user trajectories based on spatio-temporal co-visiting patterns using morton-code-based quadtree, which enables us to further prune trajectories. By combining the above two indexes, we achieve an order of magnitude performance improvement over the baseline solutions both in terms of processing time and I/O costs.

1.5 Contributions

The contributions of the thesis are summarized as follows:

- We first formulate the package delivery query using a large trajectory database comprising of user daily commuting routes, which helps us to deliver packages between any source-destination locations in a city by exploiting user trajectories.
- We propose two novel index structures, summarizing the trajectory orientation with one index and capturing their co-visiting patterns with the other one. They both help us to prune a large number of irrelevant trajectories.
- We introduce a best-first exploration to prune spaces from their connectivity through the trajectories and use an informed search technique for matching the pruned trajectories with the package delivery request.
- We conduct experiments on real-world datasets to demonstrate the efficiency and effectiveness of our proposed indexes and algorithms in terms of processing time testifying the online nature of query processing, I/O cost and success rate of package delivery.

1.6 Thesis Organization

We organize the rest of the writing as follows. Chapter 2 presents a background of spatio-temporal databases and traditional indexing schemes and highlights the distinctions with some related works. Chapter 3 formally defines the crowdshipping query as best packet delivery path

query and extends it to m best paths. Then, Chapter 4 outlines modeling the query to a trajectory matching problem and providing a baseline solution by mapping it to path finding in a graph. Next, Chapter 5 describes the our two proposed index structures and how we exploit them for spatio-temporal pruning of the solution candidates from a large trajectory database. Chapter 6 experimentally shows the efficiency and effectiveness of our approach on real world datasets by comparing with a baseline. Finally, Chapter 7 concludes our works and presents future research directions relating to our problem.

Chapter 2

Background and Related Works

2.1 Background

We introduce the readers to some basic concepts related to spatio-temporal databases. For example, we highlight spatial, temporal and spatio-temporal data, some traditional indexing schemes and so on in this section.

2.1.1 Introduction to Spatio-temporal Database

Spatial Data

Spatial data refers to any attribute of an object in terms of its location, position, boundary, direction and so on. It deals with information related to the geography or the geometry of objects [20]. For example, points, polylines, polygons etc. are used to denote different spatial objects like a particular location, connecting roads, district or country boundaries etc. on a map. Besides, latitude, longitude, elevation corresponding to different objects in the geographic coordinate system is also considered as spatial data and sometimes called geospatial data [21,22].

Spatial Database

A spatial database is one that is optimized for storing and querying on spatial data [20]. If spatial data is stored using the traditional B-tree indexes in RDBMS, finding nearby spatial objects or those within a spatial range is inefficient. This is because traditional indexes are not optimized to deal with spatial queries. Besides, retrieval of spatial objects from physical disks is easier if they are stored based on spatial locality. Spatial databases provide these functionalities.

Temporal Data

Data containing timestamps indicative of the its other attributes being valid based on the specific time instance or time window is referred to as temporal data [20]. Traditional DBMS provide datatypes for working with temporal data. For example, Oracle provides datatype to work with temporal data [23].

Spatio-temporal Data

Spatio-temporal data has both spatial and temporal attributes [24] as the name suggests. When spatial data is augmented with temporal attribute, e.g. timestamps, indicating the spatial data to be applicable at its temporal counterpart, it is called spatio-temporal data. For example, movement trajectory of a vehicle is captured with spatio-temporal data. The trajectory is basically a sequence of timestamped locations besides other optional attributes. So it can be stored as a sequence of points, each having a location (e.g., latitude, longitude when we consider geospatial data) at a particular time.

Spatio-temporal Database

Spatio-temporal database stores and processes spatio-temporal data (example: trajectory database). These databases requires index structures for both the spatial and the temporal dimensions of the spatio-temporal objects [20]. It retains the spatio-temporal locality of the objects and processes both spatial and temporal queries efficiently. The locality may depend on the nature of the query being processed. Trajectory database is a good example of a spatio-temporal database.

2.1.2 Traditional Spatial Indexes

A spatial index is the underlying data structure of a spatial database. It facilitates fast access to spatial data [25] by grouping the objects with spatial locality. Thus objects that are expected to be retrieved together from a spatial query are stored together. Usually trees of different structures are used in spatial indexing. R-tree and Quadtree are examples of two such trees.

R-tree

The R-tree is one of the primitive spatial indexes proposed by Antonin Guttman in 1984 [26]. It groups the spatial objects in rectangles, further considers them as spatial objects and groups these rectangles hierarchically. These rectangles are called minimum bounding rectangle (MBR) as they encompass the spatial objects with minimum possible bounding area. Each node of R-tree corresponds to a list of MBRs and the leaf nodes hold the MBRs containing the original spatial

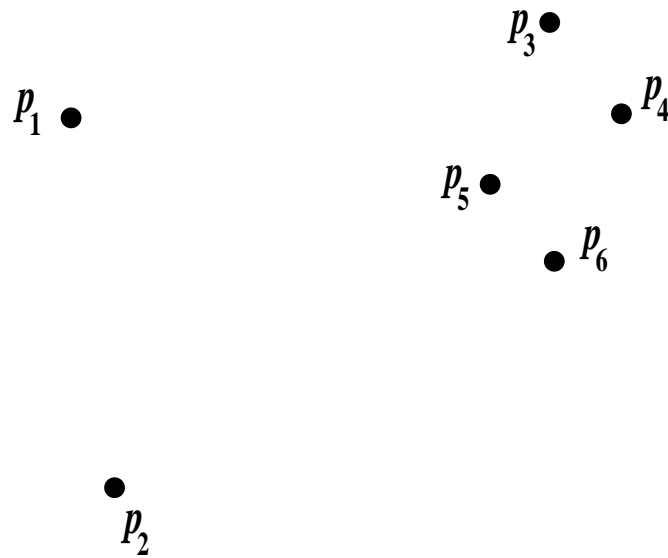


Figure 2.1: Spatial objects distributed in the space

objects. These MBRs at the leaf nodes are directly mapped to physical disk blocks. Each MBR can hold at most a threshold number of objects. The threshold is determined from the size of the spatial objects and the disk page size (typically 4096 bytes for example).

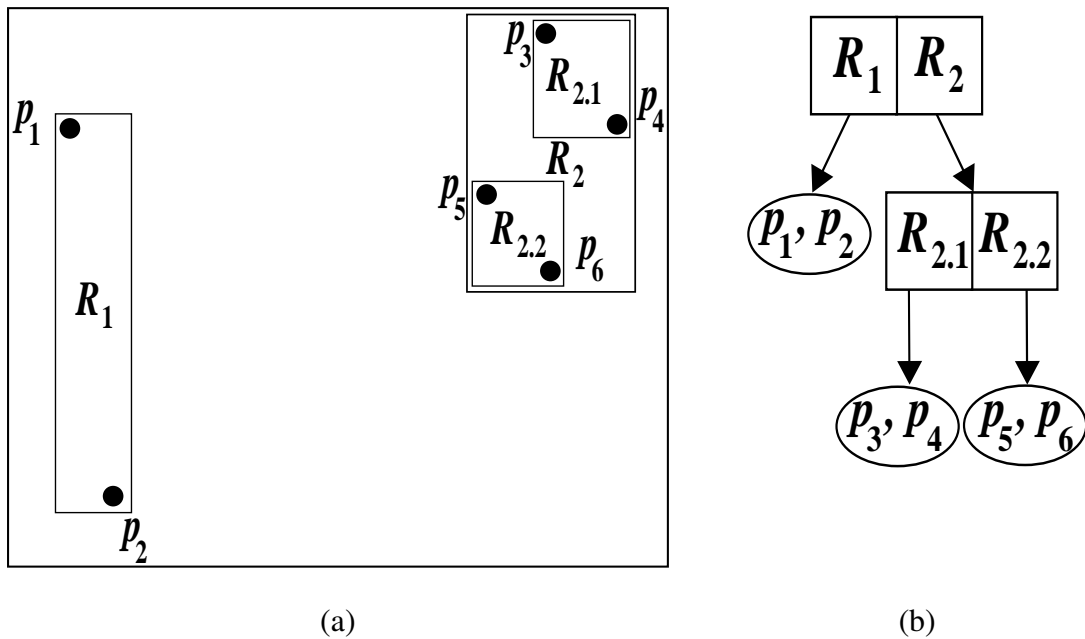


Figure 2.2: R-tree constructed with spatial objects of Figure 2.1

Let us consider an example with six spatial objects. We denote them as points, p_1, \dots, p_6 , distributed over the space, as shown in the figure 2.1. Suppose we want to index them using an R-tree where each node of the R-tree can hold at most two spatial objects. The resulting R-tree groupings are shown in the figure 2.2(a). In order to keep the area of the bounding rectangles

minimum, $\{p_1, p_2\}$, $\{p_3, p_4\}$, $\{p_5, p_6\}$ are grouped in three MBRs, R_1 , $R_{2.1}$, $R_{2.2}$ respectively. Then the MBRs of $R_{2.1}$, $R_{2.2}$ are again grouped into a larger MBR R_2 . The resulting structure of the R-tree is shown in the figure 2.2(b).

Quadtree

Quadtree is another renowned traditional spatial index proposed by Hanan Samet [27] in 1984. It is a hierarchical data structure that works by partitioning the available space into four equal-sized quadrants. Each leaf node of a quadtree can hold a threshold number of spatial objects. Each intermediate node holds the pointers to its four child nodes. The capacity threshold of a quadtree leaf depends on the size of a disk page and the size of the spatial objects. The objects grouped in a single quadtree leaf are typically stored in the same physical disk page while those of the nearby quadtree leaves either share the same disk page or occupy the adjacent ones.

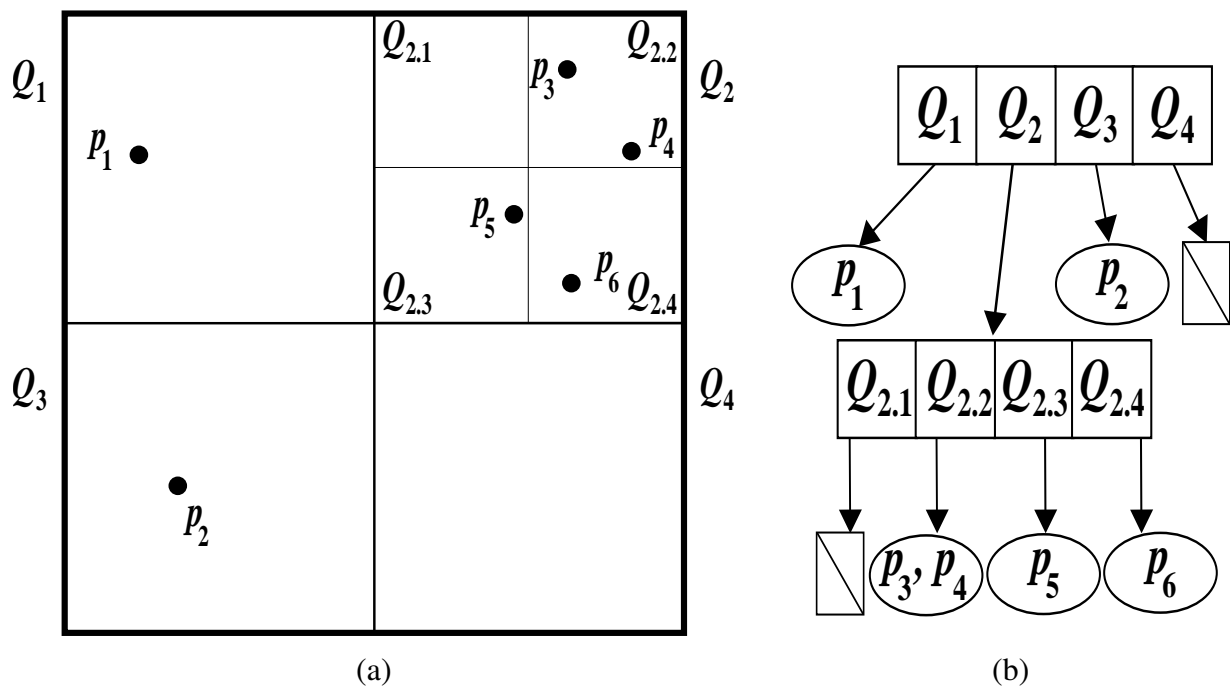
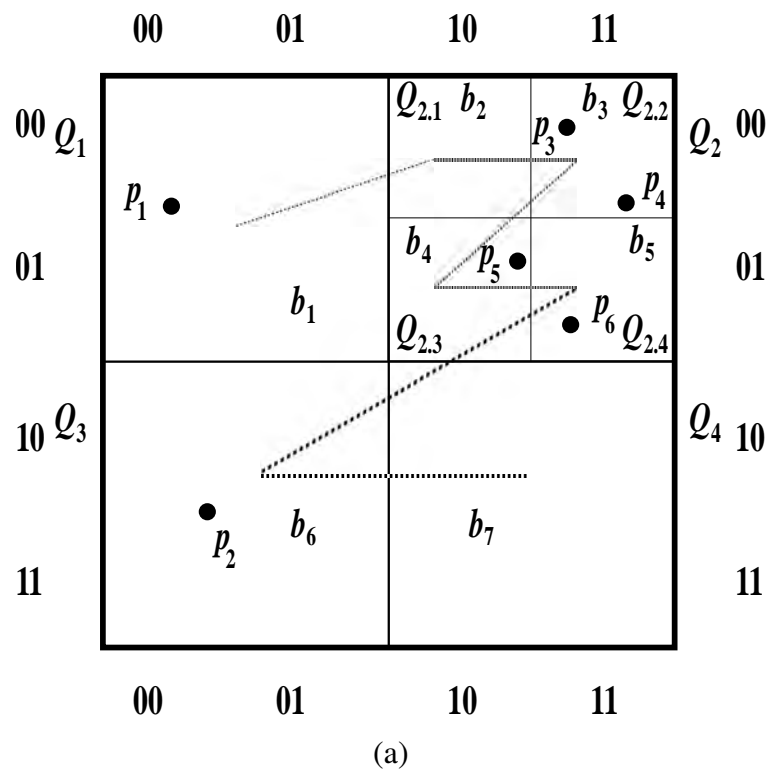


Figure 2.3: Quadtree constructed with spatial objects of Figure 2.1

Let us again consider the example of figure 2.1. Assume we now want to index them using a quadtree where the capacity of each node is limited to two spatial objects. The partitioning of space is shown in the figure 2.3(a). The space is initially divided into Q_1 , Q_2 , Q_3 , Q_4 quadrants resulting in p_1 to be in Q_1 node, p_2 in Q_3 node and the rest in Q_2 node. Since Q_2 now contains more than two points, it is further divided into its four children node, namely, $Q_{2.1}$, $Q_{2.2}$, $Q_{2.3}$, $Q_{2.4}$. Now $Q_{2.2}$ holds two points while the other child nodes of Q_2 holds at most one point. So no further partitioning is required. Note that, unlike R-tree leaves, quadtree leaves can be empty (e.g., $Q_{2.1}$ and Q_4). Figure 2.3(b) shows the structure of the resulting quadtree.



Morton Code	Binary Number Range
b_1	0000 to 0011
b_2	0100
b_3	0101
b_4	0110
b_5	0111
b_6	1000 to 1011
b_7	1100 to 1111

(b)

Figure 2.4: Morton ordering of the quadtree nodes constructed with spatial objects of Figure 2.1

Morton Code

While organizing the spatial objects under different quadtree leaves in the physical disk pages, we need a linear ordering of the quadtree nodes. Morton ordering is such a linear ordering technique using a space filling curve of Z shape, and therefore, alternatively called the Z-order [28]. In this approach, each quadtree node is filled continuously using a Z-shaped curve and assigned a range of unique codes, also known as the morton code. The code is generated by enumerating the coordinate axes with binary numbers and interleaving them bit by bit. For example, let us consider figure 2.4(a) where the Z-pattern starts from Q_1 , visits the children of Q_2 recursively (i.e., $Q_{2.1}$, $Q_{2.2}$, $Q_{2.3}$, $Q_{2.4}$), goes to Q_3 and finishes at Q_4 . These nodes are assigned morton numbers b_1 to b_7 sequentially. Since Q_2 is divided into its children, we need two bits per

coordinate axis to represent each smallest quadtree nodes uniquely. As a result, we get the binary number range 0000 to 0011 for b_1 by interleaving 00 with 00 to get 0000, 00 with 01 to get 0001, 01 with 00 to get 0010, and 01 with 01 to get 0011. Similarly, we get the binary number range for each morton code as shown in figure 2.4(b). Note that, the linear ordering of the quadtree nodes helps store the spatial objects they contain to nearby disk pages. For example, assume each disk page can hold two points. Since b_1 contains only p_1 , b_3 contains p_3 and p_4 , we can store p_1 in a single disk page and p_3 and p_4 in the following page. Next, as b_4 contains p_5 and b_5 contains p_6 , we can store them both in the next page. Finally as we can store p_2 (contained in b_6) in the following disk page. Thus we exploit the benefits of morton code.

2.2 Related Works

The related body of works encompasses package or parcel delivery, particularly in crowdsourced context, spatio-temporal trajectory indexing, trajectory matching and join, and so on. In this section we discuss the relevant studies on each of these topics and distinguish with our work.

2.2.1 Crowdsourced Package Delivery

We find two categories of approaches towards addressing the crowdsourced package delivery problem. The first one is offline formulation of the problem using integer linear programming and the second one is online query oriented. In the former approach, the problem is mostly based on advanced planning given all the information while in the latter approach, it needs to address the problem on the fly based on the users' demand. Next we elaborate them followed by some miscellaneous works from orthogonal perspectives.

Integer Linear Programming Formulation

Most of the studies on the crowdsourced package delivery solve different variants of the problem offline using integer linear programming (ILP) techniques [14–17, 29]. Chen et al. [15] formulate the problem as ILP, given a set of agent routes and task nodes imposing two sets of constraints. They propose a solution based on greedy heuristic by allocating a task to an agent who needs a minimum detour to do the job. Arslan et al. [16] construct ILP with the objective of profit maximization given some delivery tasks and vehicle OD pairs subject to a set of constraints. They provide an exact recursive algorithm for assigning sequences of tasks to the vehicles. Besides, they introduce rolling horizon framework that performs batch processing of offline ILP solution providing an abstraction for an online solution. But as the solution neither works with multipoint trajectories, nor shows scalability in terms of runtime, it is not amenable to the purely online nature of the queries that we study in this work. Besides, Archetti et al. [14] model

crowdshipping as a vehicle routing problem with occasional drivers and solve it by adopting a mixed integer programming formulation. Their motivation is to involve the in-store customers to deliver goods to online customers on their way home. They assume the model to be static, i.e., the delivery requests and the availability of the drivers are known beforehand. Macrina et al. [17] extend the work of Archetti et al. [14] by augmenting time windows for the occasional drivers and delivery requests, and further develop their work by introducing transshipment nodes [29]. However, they address crowdshipping as a route planning problem, model it on a complete directed graph and impose constraints on time, flow, capacity to formulate and solve an integer programming problem.

Online Query Formulation

To the best of our knowledge, online formulation of the crowdshipping query is an uncommon research problem. Only a recent study has addressed the crowdshipping problem as an online query. Chen et al. [18] have formulated package delivery as an arriving on time problem. They use historic taxi OD pairs to build a package transport network in a unimodal private transport setting. They design an adaptive taxi scheduling algorithm that finds a maximum arriving-on-time probability path for each packet request. They generate the packet source and destination randomly at the interchange stations and assign random deadlines. Their method is based on the decision of whether to choose a taxi to send the packet to the next interchange station or to wait for an upcoming ride. They introduce several heuristics to make this decision. We, on the other hand, consider commuting trajectories on both public and private transports on the whole road network space. As a result, our approach to the problem is more robust, generic and trajectory oriented.

Miscellaneous

Crowdshipping is also considered as a challenging problem from the supply chain management, economic and environmental perspectives. For instance, Le et al. [30] reviews current practice in crowdshipping from the perspective of supply, demand and operations and management and find out the scopes of improvement. Besides, Gatta et al. [31] evaluates the environmental and economic impacts of public transport based crowdshipping in urban areas to assess the potential of involving the crowd in this service. We, on the other hand, focus on how we can match the commuter trajectories in a spatio-temporal database to accomplish the crowdshipping task from an orthogonal perspective.

2.2.2 Spatio-temporal Indexes

Variants of the traditional spatial indexing data structures like R-tree and Quadtree are usually extended for temporal queries to form spatio-temporal indexes. The R-tree groups the nearby spatial objects together in rectangles, referred to as Minimum Bounding Rectangles (MBRs), in a hierarchical pattern. The MBR of the root node spans the whole space while those of child nodes cover sub-regions of the entire space. The Quadtree, in contrast, divides the available space recursively into four disjoint equally sized quadrants until a quadrant contains at most a given threshold number of spatial objects.

Introducing spatio-temporal trajectory indexes based on the variants of R-tree (e.g., [32,33]), and Quadtree (e.g., [34]) has been widely researched. These studies can be broadly categorized into two directions: (i) The first approach represents a trajectory as a set of independent points. Then it aims to design data structure for indexing the independent points. In this indexing approach, points that are spatially co-located but are not necessarily from the same trajectory, are stored together. An additional auxiliary data structure stores the link among different points of an object at the cost of extra storage. (ii) In the second approach, the connectivity between consecutive points of a trajectory is taken into account during the index structure construction. Thus it is expected that all/most of the points of the same trajectory are stored together.

Independent Spatial and Temporal Index

Segregation of the spatial and the temporal attributes has been addressed in several R-tree variants, namely MR-tree [35], HR-tree [36], MV3R-tree [37]. These studies focus on constructing separate R-trees based on spatial attributes of the trajectories for different temporal windows. In another study, the authors augment B-tree with time window boundaries and construct the Start-End timestamp B-tree (SEB-tree) [38]. The SEB-tree partitions the search space into different zones and assigns a B-tree index for each zone, where the spatio-temporal objects in a zone are sorted by their timestamps. All these indexes belong to the first category of indexes.

Trajectory Locality Preservation

The second group of indexes emphasizes on the connection between points of a trajectory while indexing them. For instance, the temporal dimension is augmented with the 2D spatial counterpart in a 3D R-tree [32] that can handle the spatial and the temporal queries similarly. The trajectories, in this approach, are plotted in 3D comprising of two spatial and one temporal dimensions. They are then bounded by spatio-temporal MBRs that are indexed by the 3D R-tree. The spatio-temporal R-tree (STR-tree) [33] is an R-tree variant that partially conserves trajectory locality trying to keep the trajectory segments together. Yet, the segments of a trajectory may be distributed over several STR-tree nodes and nearby trajectories may be stored in different groups.

On the other hand, the trajectory-bundle tree (TB-tree) [33] chooses trajectory preservation over spatial locality. It is also an R-tree based index where each leaf node holds only the segments belonging to the same trajectory. But different trajectory segments that are spatially close, are stored separately to uphold trajectory preservation. Trajectory retrieval during query processing requires multiple iterations in both STR-tree and TB-tree indexes. The segments are incrementally fetched as the query processing interacts with different nodes. As a result, we cannot adopt them to our packet delivery query since a deliverer cannot carry a packet on some disjoint segments of her trajectory.

Trajectory Segmentation

The Scalable and Efficient Trajectory Index (SETI), proposed in [39] comprises of a multilevel index to handle the scalability issues more efficiently. SETI basically deals with the spatial attributes and the temporal properties in two levels of index. The first level divides the spatial domain into uniform non-overlapping cells of a fixed size. The coordinates of the endpoints of a trajectory segment help determine which cell it belongs to. In case a segment crosses cell boundaries, it is split into multiple segments so that each split belongs to one cell only. The second level of index uses R-trees to index the temporal spans. Each cell of the first level index contains a second level R-tree. Each R-tree stores the minimum and maximum timestamps of a cell, i.e., the minimum and maximum timestamps of all the trajectory segments belonging to the current cell as per their spatial orientation. Thus, spatial and temporal dimensions are processed independently in the SETI indexing approach. So the spatio-temporal range queries are answered quickly using this approach through independent filtering at two levels of the index. However, we cannot use this index since whole trajectory retrieval is costly for segment-wise storing. Many R-trees associated with the spatial cells may need to be traversed for retrieving all the segments of a trajectory. In our package delivery query, whole trajectories are important as they indicate the involvement of different deliverers.

Some recent indexing techniques also relies on the trajectory segmentation approach. For instance, TrajStore, TrajTree [40], TQ-tree [34] uses this idea for indexing trajectories. TrajStore partitions trajectories into segments and clusters the segments with spatio-temporally locality on the disk page. In TrajTree, sub-trajectories with their bounding boxes are stored at the leaf nodes while an intermediate node maintain a sequence of the bounding boxes of their child nodes. Ali et al. [34] propose a two-level quadtree index to store co-located trajectory segments hierarchically, so that longer and shorter trajectories are grouped separately in addition to their colocated pattern. This index is suitable for a special group of trajectory coverage query.

2.2.3 Trajectory Matching

Besides trajectory construction and indexing, trajectory matching or similarity search with respect to query trajectory, set of POIs or time windows has been addressed in several studies. These studies aim at finding one or more trajectories based on their similarity or overlap with another trajectory, their cumulative proximity with a set of point locations, their intersection with query time windows or their distance-wise ranking from neighboring trajectories. The similarity, proximity, ranking metrics depend on the applications addressed in the studies. We omit the studies on trajectory construction and mining in this literature as it is out of the scope.

Trajectory Search by Trajectory Similarity

Studies have defined trajectory similarity in different ways. For example, Chen et al. [41] uses edit distance between two trajectories as their similarity metric. Besides, Shang et al. [42] consider both spatial and textual attributes of the trajectories to determine their similarity. Frentzos et al. [43] propose a metric to identify the dissimilarity between two trajectories and find top k similar trajectories with respect to a query trajectory by adopting a best first technique. Besides, six similarity measures on a real taxi trajectory dataset and their comparative review is presented in [44].

Besides, unlike the unweighted assumption for similarity metrics in the aforementioned studies, several studies consider the metric to be weighted. For instance, Shang et al. [45] assigns weights to each point along the query trajectory based on the choice of the users. They find the top k similar trajectories by considering the weighted trajectory points in the similarity function. They perform the matching by iterating over the query trajectory points and checking if circles around them intersect with the available trajectories. The notion of the weight of the points is modeled by circles of different radii around them. Points with greater weight have circles with larger radii. The similarity is calculated by considering how many points are touched by a trajectory. However, since we are interested in matching only two spatial points, packet source and destination by means of possibly joining the indexed trajectories, our main challenge lies in efficient retrieval of trajectories that may possibly be joined to connect them. So adopting the similarity search for the source-destination pair is not an efficient alternative to our solution.

Trajectory Search by Point Location

Tang et al. [46], Han et al. [47] address the problem of finding nearby trajectories given a set of points. While Tang et al. [46] matches the trajectories to the given points modeling the cost as a sum of distances from the points, Han et al. [47] calculates the closeness with respect to travel time. Both of them index the available trajectory points using R-tree independently without preserving their trajectory locality variants and find k nearest trajectories with respect

to the query points. However, in our proposed BDPT query, trajectory locality is important because two different trajectory segments indicate two different commuters. Once a commuter receives a packet, she cannot hand it over until there is a packet keeper to facilitate this exchange process. Therefore, matching different points from different trajectories with the packet source and destination does not serve our purpose.

Chapter 3

Problem Formulation

We formally define the crowdshipping query in this chapter. So, we first introduce some key terms and concepts. Next we mathematically express the cost metric. Then we formulate the queries using the defined terms and mathematical concepts.

3.1 Terms and Notations

Let, T be a set of trajectories where each trajectory represents the daily travel path of a commuter. A trajectory t is defined as a sequence of spatio-temporal points, i.e., $t = \{p_1, p_2, \dots, p_{|t|}\}$. Each point is represented as $p(l, \tau)$, where l is the location and τ is the time when location l is visited by t . The trajectory t , therefore, has $|t|-1$ segments where the i -th ($1 \leq i < |t|$) segment is denoted by $s_i = (p_i, p_{i+1}, m_i)$. Here, m_i is the transport mode used by t to go to p_{i+1} from p_i .

Packet Delivery Request is a quadruple (S, D, τ_e, τ_l) , where S is the source, D is the destination, τ_e is the earliest packet pickup time, and τ_l is the latest packet delivery time. We denote this request as P_{req} .

Packet Requester, denoted by r , is a user who requests a packet delivery.

Packet Deliverer is a commuter who delivers a packet from one place to another. We denote a deliverer as d . Each deliverer has an associated trajectory t . We use the terms a deliverer d and her trajectory t interchangeably throughout the thesis. The set of all deliverers available for delivering a packet is denoted by T .

Packet Keeper is a storage space or a user who keeps the packet in the middle of a multi-hop delivery. We denote a keeper as $k = (l, \tau_o, \tau_c)$ where l represents its location, τ_o and τ_c represent its opening and closing times, respectively. In other words, the keeper is not available before τ_o or after τ_c . The set of all keepers is denoted by K .

Detour Constraint (δ): A deliverer may not want to deviate too much from her original travel route. The distance threshold δ denotes the maximum distance any deliverer is willing to deviate from her route. Since the travel time depends on the distance δ , it is not considered separately.

Besides, she may only be interested in the deviation at exchange junction points. For example, let a deliverer trajectory be $t_1 = \{p_1, p_2, p_3, p_4\}$, where she goes from p_1 to p_2 on foot, p_2 to p_3 by train and p_3 to p_4 by bus. Now, since she may only want to deviate her journey from any of its junction points to pickup a delivery packet and deliver it to the destination or to an intermediate stop, the source of the packet should preferably be somewhere near p_1 or p_2 or p_3 , and the destination of the packet should preferably be somewhere near p_2 or p_3 or p_4 .

Delivery Path: Given a packet request, P_{req} , a delivery path is an ordered sequence of packet deliverers and keepers who can collaboratively deliver the packet from the source to the destination of P_{req} while satisfying the detour constraint δ . We represent a delivery path as $\pi(P_{req}) = \{d_{i_1}, k_{j_1}, d_{i_2}, k_{j_2}, \dots, k_{j_{n-1}}, d_{i_n}\}$ where the amount of detour for a deliverer d_{i_x} to fetch a packet from keeper $k_{(j-1)_x}$ and drop it at keeper k_{j_x} ($1 \leq x \leq n$, $d_{i_x} \in T$ and $k_{j_x} \in K$) is no more than δ . We denote the set of delivery paths for a packet request as $\Pi(P_{req})$.

Best Delivery Path: Our goal is to serve a packet delivery request, P_{req} , with the minimum *cost* subject to the detour constraint δ . The cost is a generic term that may depend on the application or the user. Essentially, we need to find a delivery path π , i.e., a sequence of packet deliverers and keepers to deliver packet incurring the minimum possible cost.

3.2 Delivery Cost Function

The cost of a delivery path can be defined in terms of delivery time or path length. So the fastest or the shortest delivery is possible based on the choice of the cost function. Besides the detour constraint, δ is considered in cost computation to introduce a binary notion that defines cost as a finite quantity if and only if detour distance is within the allowed limit δ .

The cost of a packet delivery consists of three components: (i) a fixed cost for the distance; (ii) a cost for detour i.e., deviation from the regular trajectory of a deliverer; and (iii) the cost for storing the packet at the keeper.

Cost of a Delivery Path, π : If a packet delivery request P_{req} is delivered via a delivery path, $\pi(P_{req}) = \{d_{i_1}, k_{j_1}, d_{i_2}, k_{j_2}, \dots, k_{j_{n-1}}, d_{i_n}\}$, then we can define cost of the delivery path π as follows.

$$Cost(\pi) = \sum_{x=1}^{n-1} Cost(d_{i_x}, k_{j_x}) + Cost(k_{j_x}) + Cost(k_{j_x}, d_{i_{x+1}}) \quad (3.1)$$

$$Cost(d, k) = \sum_{i=a}^{b-1} Cost(d.s_i) + Cost_{\delta}(d.p_b, k) \quad (3.2)$$

Here $Cost(d.s)$ is the measure of cost along each trajectory segment of the packet deliverer d , usually in terms of time or distance. Deliverer d receives the packet at its $d.p_a$ point and hands it over going to keeper k from $d.p_b$ point ($1 \leq a \leq b \leq |d|$). The corresponding detour cost is

denoted as $Cost_\delta(\cdot)$. We assume the deliverer always returns to the same point after a detour on her trajectory. The detour distance is calculated for going to the keeper from a trajectory point and returning to the trajectory and is represented as follows.

$$Dis_{detour}(d.p, k) = Dis(d.p, k.l) + Dis(k.l, d.p) \quad (3.3)$$

$$Cost_\delta(d.p, k) = \begin{cases} Dis_{detour}(d.p, k) & \text{if } Dis_{detour}(d.p, k) \leq \delta \\ \infty & \text{otherwise} \end{cases} \quad (3.4)$$

The $Dis(\cdot)$ function is generic. It can accommodate both Euclidean and road network distances.

$$Cost(k, d') = Cost_\delta(d'.p_{a'}, k) \quad (3.5)$$

Here, $Cost(k, d')$ consists of only the detour cost from a point $p_{a'}$ of the deliverer d' . It is calculated in the same way as finding $Cost_\delta(d.p, k)$ using Equations 3.3 and 3.4. Finally, $Cost(k)$ of Equation 3.1 is required to store the packet at keeper k to facilitate packet handovers.

3.3 Formal Definition of Best Packet Delivery Queries

Now we formally define the best and the m -best delivery path queries for a packet request as follows.

Definition 1. *BDPT (Best Delivery Path Query).* Given a set T of deliverers' trajectories, a set K of keepers at different locations of the city, a packet delivery request P_{req} , and a detour distance threshold δ , the best delivery path query finds a delivery path π such that $Cost(\pi) \leq Cost(\pi')$ for every other delivery path π' .

Definition 2. *m -BDPT (m -Best Delivery Paths Query).* Given a set T of deliverers' trajectories, a set K of keepers at different locations of the city, a packet delivery request P_{req} , a detour distance threshold δ , and an integer m ($1 \leq m$) the m -BDPT query finds the top- m delivery paths $\Pi_m(P_{req})$ from the set of all delivery paths $\Pi(P_{req})$ such that $\forall \pi \in \Pi_m, \forall \pi' \in \Pi \setminus \Pi_m, Cost(\pi) \leq Cost(\pi')$.

Chapter 4

Modeling and Baseline

To answer the BDPT query, we need to identify the commuter trajectories, that can participate in delivering a packet, from a large trajectory database. These trajectories should be spatio-temporally conforming to the packet delivery request. Specifically, they should pick the packet from the source, exchange it from one commuter to another via the keepers, and drop it at the destination without requiring to deviate from their regular itineraries by more than a pre-specified amount.

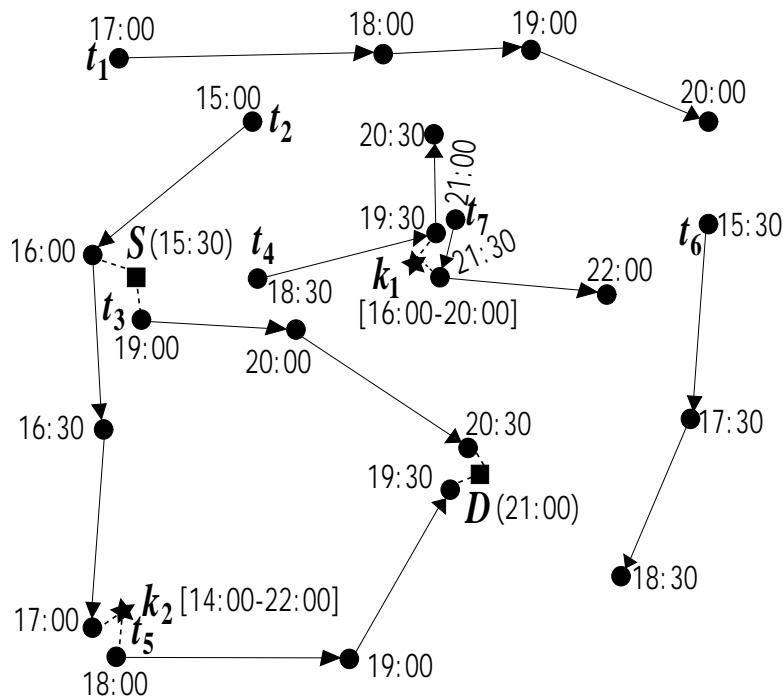
In this chapter, we first formulate the problem as a path finding problem on the graph constructed from commuter trajectories and packet keepers, where we can adopt an informed search based baseline solution.

4.1 Modeling User Trajectories

To answer the BDPT query, we construct a graph, hereafter referred to as *TrajGraph*, with commuter trajectories. The trajectories are spatially mapped to the existing road network and augmented with temporal attributes. When a packet delivery is requested, our goal is to find a best delivery path for it in the *TrajGraph*.

Suppose a packet has to be delivered from S to D as shown in Figure 4.1. t_1, t_2, \dots, t_7 represents trajectories of some packet deliverers and k_1, k_2 denote two packet keepers. Each user trajectory consists of a sequence of timestamped locations indicating her regular itinerary profile. A keeper is denoted by a single timestamped location only. Both the trajectory points and the keepers are represented as nodes in the *TrajGraph* while the edges i.e. trajectory segments holds the cost of traveling from one point to the next one, based on the transportation mode.

The nodes of *TrajGraph* are represented spatio-temporally, i.e., with both location and timestamps. The spatial attribute holds two coordinates while the temporal attribute consists of time window indicating the availability of the user at the specific location. The time window can be expressed as either the start and end of window $[\tau_s, \tau_e]$ or the midpoint and range on either

Figure 4.1: Packet delivery example using *TrajGraph*

side $(\tau_{mid}, \tau_{range})$. In the latter case (as shown in Figure 4.1), the start-end time window can be calculated as $[\tau_{mid} - \tau_{range}, \tau_{mid} + \tau_{range}]$. The duration of the availability of a keeper at her specific *TrajGraph* node is supposed to be significantly higher than that of a trajectory node, as explicitly shown in the Figure 4.1. The edges hold travel cost along a trajectory taking the transportation mode and additional detour cost (if applicable) into consideration. Attributes like time, distance which can directly be derived from adjacent nodes, are not stored to save memory requirement. Note that, a key challenge of modeling multiple transport modes used by the same user to travel between a particular source destination pair is resolved by adding parallel edges with appropriate costs.

Let us consider the example of Figure 4.1. Here, only the spatio-temporal attributes of each trajectory are shown. Transportation mode, breakdown of costs etc. are omitted from the figure for brevity. For instance, trajectory t_5 consists of 3 nodes with timestamps 18:00, 19:00, 19:30 respectively. Though there is an associated time range with each node to specify the time windows, we have omitted it in the example. We can assume a suitable value of τ_{range} (say, 10 minutes) based on the itinerary profile of a deliverer.

4.2 Base Solution using Path Finding Algorithm

In order to process the BDPT query, the source and destination of a packet request, P_{req} are connected to the existing spatio-temporal $TrajGraph$. This is done simply by mapping the source and destination to a set of nearby nodes. We achieve this by maintaining a traditional spatial index (e.g. a quadtree) for the trajectories and running a range query on it with the source and destination locations. A similar approach is applied for the keepers during $TrajGraph$ construction. Each keeper node is bidirectionally connected to its nearby nodes. Note that, these edges are crucial in joining trajectories to complete a packet delivery and contributes to detour cost if they exist in any packet delivery path. Different modes of transports between two $TrajGraph$ nodes via different trajectories are modeled by adding multiple parallel edges. Alternatively, a single edge for only public transport may suffice assuming both private and public transport user can travel along it.

Algorithm 1: findDeliverer($P_{req}, TrajGraph$)

Input: Packet request P_{req} , Trajectory graph $TrajGraph$
Output: A list of packet deliverers

- 1.1 $(L_S, L_D) \leftarrow \text{Filter}(P_{req}, now, TrajGraph)$
- 1.2 $bestPaths \leftarrow \emptyset$
- 1.3 **for** $node \in L_S$ **do**
- 1.4 $temp \leftarrow \text{shortestPath}(node, L_D, TrajGraph)$
- 1.5 $bestPaths.insert(temp)$
- 1.6 $bestPath \leftarrow \text{top}(bestPaths)$
- 1.7 $pktDeliverers \leftarrow \text{trajectories}(bestPath)$
- 1.8 **return** $pktDeliverers$

The $\text{Filter}(\cdot)$ function on line 1.1 maps the source and destination of the packet delivery request P_{req} to $TrajGraph$ nodes. It filters out the spatially distant and temporally disjoint nodes (i.e., those which have time window ending before $P_{req}.\tau_e$ or starting after $P_{req}.\tau_l$). It then assigns the potential source and destination nodes to two lists, L_S and L_D respectively. Note that, starting from a node of L_S and reaching one of L_D accomplishes a delivery task. In BDPT we seek for such a path incurring the minimum cost. The priority queue $bestPaths$, initialized at line 1.2 orders the possible delivery paths according to their costs. In the loop of line 1.3-1.5 the shortest paths i.e., those with smaller costs are found iterating over all source nodes. The $\text{shortestPath}(\cdot)$ function in line 1.4 is generic and thus can adopt to any cost associated with the $TrajGraph$ edges. In line 1.5, the paths (if found) are inserted into the priority queue. Finally the best path from the priority queue is obtained using a generic $\text{top}(\cdot)$ function (line 1.6) that extract the best element considering the chosen cost metric. It can adopt to the user choice to minimize path length or duration. The list of deliverers and keepers of the best path are extracted and returned on line 1.7-1.8.

Note that, this algorithm allows packet handover only via keepers and allows detour to reach the source, destination or intermediate packet keeper nodes. These edges have already been added

in the graph between heterogeneous entities, e.g., a packet deliverer node and a packet keeper node during construction, to facilitate detour. A packet can be transferred from one deliverer to another using these detour edges. The cost of a detour edge is considered more than that of a regular edge representing a trajectory segment because the deliverer needs to get back to her trajectory after handing over the packet. So, the cost of returning to her regular itinerary needs to be augmented. We use the notion of *detour edges* as an efficient and approximate alternative to processing a complex detour query. Here, we take advantage of the constraint that the detour distance is bounded by a spatial threshold, that we have denoted as δ . Thus we address the challenge of computing detour cost and incorporating it as a part of our solution.

4.3 Best First Informed Search

The performance of algorithm 1 depends on the size of the *TrajGraph* i.e., the number of trajectories used in its construction. We primarily use a traditional spatial index for organizing the trajectories, retrieve the relevant ones with an appropriate range query for a delivery request and use an A* search [19] based algorithm on them. Note that, this is the detailed algorithm mentioned on line 1.4 of Algorithm 1.

Algorithm 2: minWA*Path(*start*, *nodeList*, *TrajGraph*)

Input: Source node *start*, list of destinations *nodeList*, trajectory graph *TrajGraph*

Output: A best delivery path from *start* to *nodeList*

```

2.1 Initialize a min heap:  $H \leftarrow \emptyset$ 
2.2 Initial state:  $S : (node, path, w_g) \leftarrow (start, start, 0)$ 
2.3  $fweight(S) \leftarrow 0$ 
2.4  $H.insert(fweight(S), S)$ 
2.5 while  $H.notEmpty()$  do
2.6    $(w, S) \leftarrow H.pop()$ 
2.7   if  $S.node \in nodeList$  then return  $S.path$ 
2.8   for  $adjNode \in S.node.neighbors$  do
2.9     if  $adjNode.timeSlot > S.node.timeSlot$  then
2.10        $S'.node \leftarrow adjNode$ 
2.11        $S'.path \leftarrow append(S.path, adjNode)$ 
2.12        $w_g \leftarrow gweight(S.node, adjNode, timeSlot)$ 
2.13        $S'.w_g \leftarrow S.w_g + w_g$ 
2.14       for  $destination \in nodeList$  do
2.15          $w_h \leftarrow hweight(adjNode, destination)$ 
2.16          $fweight(S') \leftarrow S'.w_g + w_h$ 
2.17          $H.insert(fweight(S'), S')$ 
2.18  $path \leftarrow \emptyset$ 
2.19 return  $path$ 

```

Instead of exhaustively exploring nodes in all directions, the A* search based algorithm 2 proceeds in a specific direction, i.e., from source to destination of the packet request. We use a cost approximation for the unexplored part of a path using a generic heuristic function $hweight(.)$ (line 2.15). It can simply give the euclidean distance between current node and destination or

the time required using the fastest available mode of transport depending on the choice of cost function. Detour cost to be incurred along the path is ignored to make it admissible by avoiding overestimation of remaining cost. The algorithm augments temporal checking on line 2.9 to prune spatially compliant states that would otherwise be further processed.

4.4 Processing m -BDPT

The m -BDPT is an extension of the BDPT query. To answer the m -BDPT query, we generalize the solution of BDPT query. We compute the top- m paths comprising of commuter trajectories in Algorithm 2 instead of the best path only. For this purpose, we retrieve top- m paths in Algorithm 1. We can modify the $top(.)$ function at line 1.6 to pick top- m items from the list for this.

The algorithm 2 can be modified by adding the paths to a list instead of returning at line 2.7. When its cardinality reaches m we return the list of best m delivery paths.

Chapter 5

An Efficient Index Based Solution

Though the baseline solution described above provide a solution for the BDPT query by formulating the trajectory matching problem as a path finding problem, its complexity depends on the size of *TrajGraph*. The size of *TrajGraph* depends on number of trajectories used in its construction as trajectory points contribute to the nodes while trajectory segments contribute to the edges of this graph.

Unfortunately, trivially indexing the trajectories with a quadtree and running range query on it, we are unable to prune most of the non-promising trajectories. As a result, a large number of trajectories contributes to a large *TrajGraph*. However, only the trajectories having spatio-temporal conformity with a packet delivery request can take part in its delivery process. We need an efficient method for identifying and joining trajectories based on their spatio-temporal conformity for a packet delivery request.

Thus we design two indexes for quickly selecting candidate trajectories and pruning the trajectories based on their spatio-temporal conformity. The purpose of the first index is to summarize the spatial connectivity of the regions using commuter trajectories. It helps us quickly prune the irrelevant subspaces. The second index further helps us group the co-visiting trajectories together on the physical disk aiding temporal pruning and ensuring lower I/O overhead.

5.1 SQ-index

We observe that if some trajectories visit packet source but does not move towards the destination or vice versa, they have low potential of being on the delivery path. And if a region near the source contains many trajectories moving towards the destination, retrieving the trajectories in that region increases the chance of a successful delivery. The same is true for a region, near the destination, having many trajectories moving towards it from the source. These observations encourage us design an index structure where the search space can be pruned considering trajectory directions.

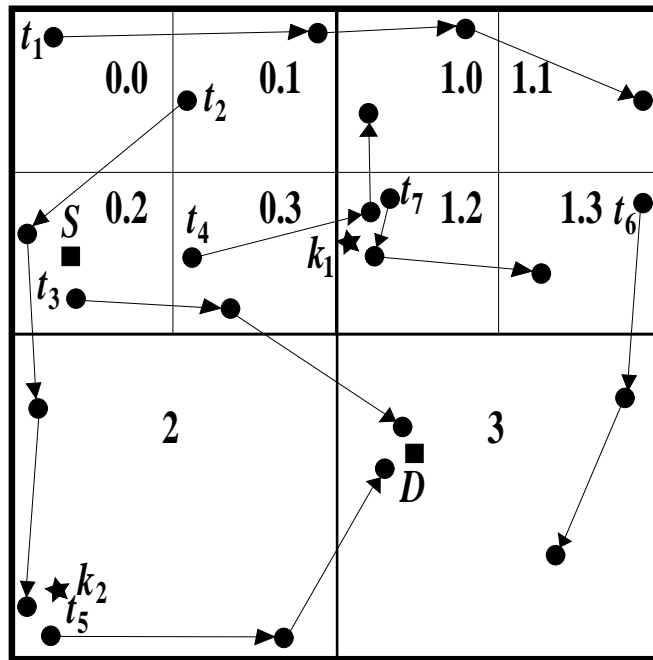


Figure 5.1: SQ-index: Summary Quadtree

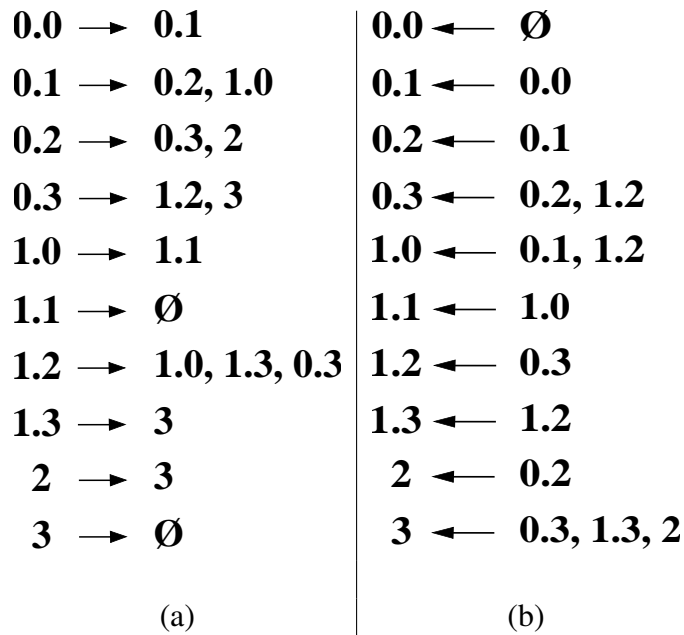


Figure 5.2: SQ-index: (a) Outgoing Blocks (b) Incoming Blocks

We propose an index based on the spatial directional property of the commuter trajectories. This index stores a summarized information of the regional connectivity by the trajectories they contain. It helps identify trajectories which are more likely to be joined to complete a packet delivery request, and thereby excludes the unnecessary ones. It helps in minimizing the volume of trajectories to be considered for a packet delivery request. So we can overcome the challenge of dealing with a huge *TrajGraph* with an overwhelming amount of trajectories, by filtering only the promising ones, more likely to be involved in the packet delivery.

The basic structure of this index is based on a quadtree. The available dataspace is partitioned using a quadtree structure and morton codes are used to enumerate its nodes. Each of the leaf nodes of the quadtree holds a triplet (*ID*, *Outgoing_Blocks*, *Incoming_Blocks*). Here *ID* represents the morton code used to identify the node. *Outgoing_Blocks* is a list of nodes which are directly reachable from the current node via some of the trajectories passing through it. Similarly *Incoming_Blocks* represent list of nodes from which the current node is directly reachable via the corresponding trajectory points they contain. So the leaves of the quadtree are interconnected and the connections are represented as a list. Note that, a large value, for example, 1000 points per node, is set as the threshold for partitioning the quadtree nodes to focus on a summary of the trajectory orientation and to limit the size of this graph.

For example, let us consider a quadtree with nodes enumerated with morton codes as shown in Figure 5.1. The nodes are linearly ordered following a Z-pattern. Accordingly, the top-left node is numbered 0, top-right one is numbered 1, bottom-left one is numbered 2 and the bottom-right one is numbered 3. When a node needs to be subdivided for having more than a threshold number of points, its children nodes get its morton codes as prefix. As a result, the top-left, top-right, bottom-left and bottom-right children of the node with morton code 0 are numbered 0.0, 0.1, 0.2 and 0.3 respectively. All the nodes in the quadtree are numbered in this manner and hereafter are referred to with their morton numbers. Now, the node 0.3 contains points from two trajectories, t_3 and t_4 . The trajectory t_4 goes to 1.2 and t_3 goes to node 3 from here. So the list of its outgoing blocks of 0.3 is {1.2, 3} (Figure 5.2(a)). Similarly, the incoming blocks of 0.3 are 0.2 via t_3 and 1.2 via keeper k_1 (Figure 5.2(b)). Note that, in case of presence of packet keepers near the boundary of any two adjacent nodes, it is assumed that the keeper connects them bidirectionally i.e. they both are added to each others' incoming and outgoing block list.

We can further augment time windows with the incoming and outgoing links or with the SQ-index blocks. We can store them as bitmap and prune irrelevant blocks in constant time with bitwise operations only. But we have skipped it for simplicity to emphasize mainly on spatial matching with the first index.

5.2 Best First Exploration with SQ-index

The performance of algorithm 2 directly depends on the number of trajectories used in the construction of the *TrajGraph*. The SQ-index, in fact, is introduced to facilitate the omission of irrelevant trajectories with respect to a packet delivery request.

Given a packet delivery request P_{req} , we can map its source and destination to the nearby blocks of the SQ-index. We explore outwards from the blocks close to the source. We do the same for the blocks close to the destination exploring inwards. If there is an overlap in the outgoing and the incoming blocks, we keep track of the overlapping block. Since overlap in the blocks may not always ensure trajectory join given the detour constraint δ , we incrementally add blocks to the pool of the outgoing and incoming ones from the current exploration state and repeat the process of overlap checking. When the overlap count crosses a threshold i.e., we have found enough overlapping blocks or the pools of the blocks are exhausted, the process stops. We then backtrack from the overlapping blocks to get a list of all potential SQ-index blocks and retrieve their underlying trajectories only. The count of overlapping blocks is important because in case of too few overlaps, we may not retrieve the trajectories that can be joined and in case of too many overlaps, some redundant trajectories may be retrieved. So the overlap count works like an estimation of whether some trajectories passing through them can be joined to serve the packet delivery request or not.

Algorithm 3 *SQReduce(.)* finds a reduced list of potential deliverer trajectories using SQ-index for constructing *TrajGraph*. The algorithm takes a packet request P_{req} and the SQ-index as input and gives a list of deliverers as output using algorithm 1. Line 3.1 - 3.4 initializes the lists of overlapping blocks, explored outblocks (i.e., outgoing blocks), explored inblocks (i.e., incoming blocks) and all SQ-index blocks to be retrieved. Line 3.5 - 3.6 finds nearby blocks of packet source and destination. The outblocks and inblocks are incrementally expanded (line 3.7 - 3.19) until a threshold number of overlapping blocks have been found. On line 3.9, a promising outblock is fetched from the current pool of outblocks denoted by O via a generic function *NextPotentialOutBlock(.)*. This function assesses the potential of a candidate outblock based on the application. For instance, in our problem the potential is measured by how close it moves to the destination. We calculate the distance from the center of a node to the destination to quantify this. The most potential outblock is then added to the potential block list and to the outblock set unless already explored (line 3.9 - 3.11). It is then matched with the already explored inblocks (line 3.12) and is added to the list of overlapping blocks in case of an overlap (line 3.13). Unless the count has reached the threshold value, a similar process is repeated by expanding the inblocks backwards (line 3.14 - 3.29). Note that, *OutNeighbors(.)* (line 3.11) returns the list of neighbors by moving forward along the trajectories from the current outblock while *InNeighbors(.)* returns those by moving backwards from the current inblock.

The potential trajectories from the list of potential blocks are retrieved (line 3.20-3.22) after the

Algorithm 3: SQReduce (P_{req} , SQ-index)

Input: Packet Delivery Request P_{req} , Summary Index SQ-index
Output: A list of packet deliverers

```

3.1 Initialize an empty list of overlapping blocks;  $L_{overlap} \leftarrow \emptyset$ 
3.2  $O_{explored} \leftarrow \emptyset$ 
3.3  $I_{explored} \leftarrow \emptyset$ 
3.4  $L_{sq} \leftarrow \emptyset$ 
3.5  $O \leftarrow \text{GetNearbyBlocks}(S, \text{SQ-index})$ 
3.6  $I \leftarrow \text{GetNearbyBlocks}(D, \text{SQ-index})$ 
3.7 while  $\text{Size}(L_{overlap}) < \text{threshold}$  do
3.8    $o \leftarrow \text{NextPotentialOutBlock}(O)$ 
3.9   if  $o \notin O_{explored}$  then
3.10      $O_{explored} \leftarrow O_{explored} \cup o$ 
3.11      $O \leftarrow O \cup \text{OutNeighbors}(o)$ 
3.12     if  $\text{Overlaps}(o, I_{explored})$  then
3.13        $L_{overlap} \leftarrow L_{overlap} \cup o$ 
3.14    $i \leftarrow \text{NextPotentialInBlock}(I)$ 
3.15   if  $\text{Size}(L_{overlap}) < \text{threshold}$  and  $i \notin I_{explored}$  then
3.16      $I_{explored} \leftarrow I_{explored} \cup i$ 
3.17      $I \leftarrow I \cup \text{InNeighbors}(i)$ 
3.18     if  $\text{Overlaps}(i, O_{explored})$  then
3.19        $L_{overlap} \leftarrow L_{overlap} \cup i$ 
3.20 for each block  $\in L_{overlap}$  do
3.21    $L_{sq} \leftarrow L_{sq} \cup \text{backtrack}(\text{block})$ 
3.22  $\text{trajectories} \leftarrow \text{Retrieve}(L_{sq})$ 
3.23  $\text{TrajGraph} \leftarrow \text{ConstructGraph}(\text{trajectories})$ 
3.24  $\text{pktDeliverers} \leftarrow \text{findDeliverer}(P_{req}, \text{TrajGraph})$ 
3.25 return  $\text{pktDeliverers}$ 

```

while loop terminates. The list of potential SQ-index blocks are obtained by backtracking from the overlapping blocks to the initial outblocks and inblocks (line 3.21). The reduced TrajGraph is constructed with only the potential trajectories and is passed to algorithm 1 (line 3.23-3.24) which returns the list of packet deliverers.

5.3 Spatio Temporal Co-Visit Index (STCoV-index)

The SQ-index helps us estimate a reduced set of trajectories that are good candidates of delivering a packet considering their spatial attribute. But in SQ-index we set a high threshold for the quadtree leaves resulting in too many trajectory points in each leaf. Besides they hold points of many different trajectories. As a result, we cannot map them to the physical disk blocks. Besides, the trajectories retrieved in the last step may not be joined if their timestamps are non-overlapping, specifically, with those of nearby keepers. So, for aiding the temporal matching for trajectory join and for further organizing them in the disk blocks to reduce the I/O overhead, we introduce a quadtree based flat time index. We observe that if some trajectories can participate in a packet delivery, their points are spatially co-located and have overlaps in the time-instant. This index combines the strengths of a quadtree, a linear temporal index and an R-tree.

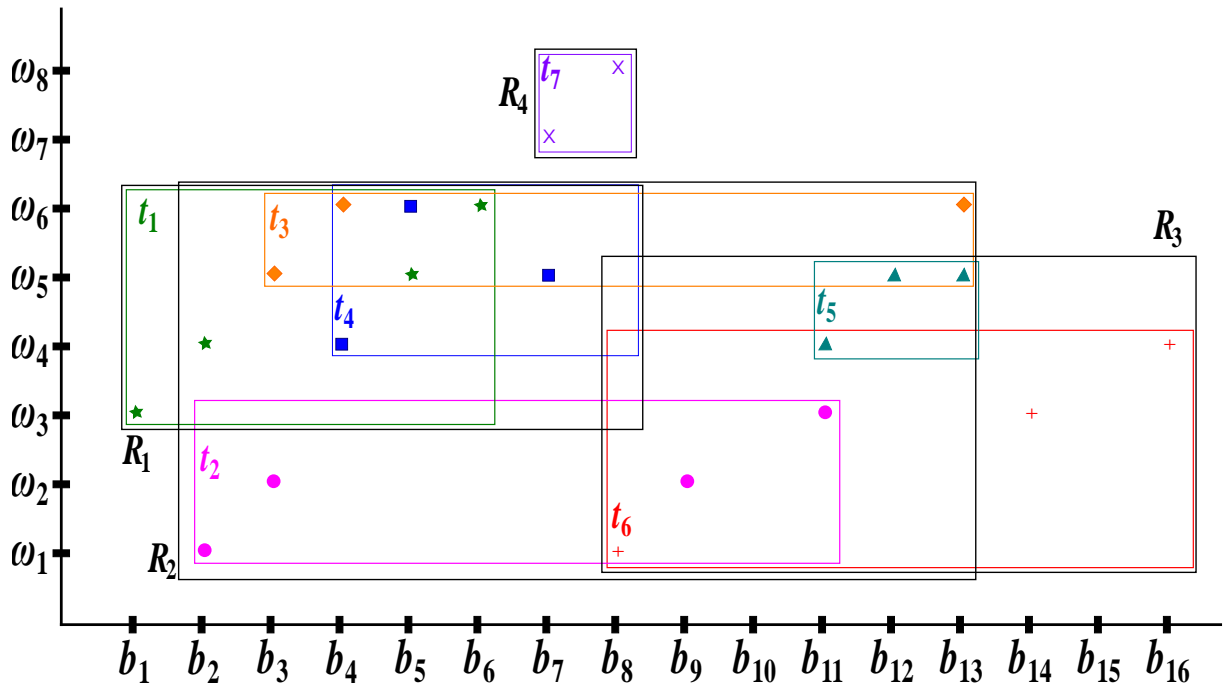
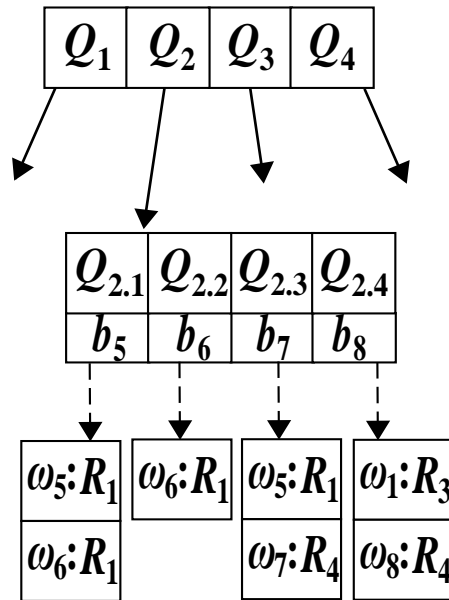


Figure 5.4: Trajectory Grouping using STCoV-index



(d)

Figure 5.5: Index Structure of STCoV-index

We use the STCoV-index to transform the trajectories to a new coordinate space. Firstly, we use a quadtree for spatially indexing the trajectory points, i.e., their latitude and longitude based coordinates and enumerate the leaves with a space filling curve (Morton code). We then map the timestamps of these points to a linear time index consisting of several time buckets. Each of the buckets in the time index gets an identifier. For instance, if the timestamp of a point is 09:00, it may be assigned to bucket 9. Thus, each trajectory point gets a spatial id from the morton code of the quadtree and a temporal id from the flat time index. Each trajectory is then represented as a sequence of tuples of spatial and temporal ids. We interpret the new representation as a transformation to new coordinates. The x -axis and y -axis denote the spatial and temporal dimensions respectively. Next we cluster the trajectories that are nearby in this transformed space. For a quick clustering technique, we use an R-tree whose leaves contain the trajectory points in the transformed coordinates. Each leaf is mapped to a disk page and we store this *disk-page id* in the corresponding time buckets of the respective quadtree leaves. Since the purpose of the R-tree is only to cluster the trajectories instead of its traditional usage in query processing, we do not store its hierarchical structure to save memory.

Example. Fig. 5.3, 5.4, 5.5 demonstrate how we construct the STCoV-index. Suppose there are seven commuter trajectories, $\{t_1, \dots, t_7\}$ Fig. 5.3(a). Each quadtree node contains at most, $\theta = 3$ points. The quadtree partitions the whole space into four quadrants Q_1, \dots, Q_4 . As each quadtree node contains 4 or more trajectory points, the blocks are further divided, e.g. Q_2 is divided into $Q_{2.1}, \dots, Q_{2.4}$. We assign a number to each quadtree block by applying a linear ordering (e.g., z-ordering). Assume, these numbers are b_1, b_2, \dots, b_{16} . Then the timestamps associated with trajectory points are assigned time-bucket number between ω_1 and ω_8 (Fig. 5.3(b)) based on which time bucket they fall into. Thus we get a new representation of the trajectories t_1, \dots, t_7 as a sequence of (b_i, ω_j) tuples. We then plot these points to a new coordinate system in a 2D space using a unique color for each trajectory (Fig. 5.4). The trajectory objects are expressed as MBRs in this transformed space. Finally, These MBRs are clustered using an R-tree. Each R-tree leaf node, R_1, \dots, R_4 maps to a physical disk-page. We store the disk-page references in different quadtree blocks of the STCoV-index, as shown in Fig. 5.5.

5.3.1 Quadtree Based Partition

In this step, we use a quad-tree to partition the points of the commuter trajectories. Each leaf-level quadrant can contain at most a threshold θ number of trajectory points. Let us consider seven trajectories $t_1 = \{t_1.p_1, t_1.p_2, t_1.p_3, t_1.p_4\}$, $t_2 = \{t_2.p_1, t_2.p_2, t_2.p_3, t_2.p_4\}$, \dots , $t_7 = \{t_7.p_1, t_7.p_2, t_7.p_3\}$ where each trajectory point $t_i.p_j = t_i.(l_j, \tau_j)$ denoting a tuple of timestamped location. The spatial points of these trajectories are recursively divided using a quadtree as shown in Figure 5.3(a), where we set $\theta = 3$. We can see quadtree blocks Q_1, Q_2, Q_3, Q_4 . Division into lower level is omitted for brevity.

5.3.2 Spatio-temporal Transformation

In the next step, we linearly order the quadtree leaves using a space filling curve. We specifically use a z-order curve [28], to number them. We call such a number, the spatial id s_{id} of the block. In a z-order space filling curve, we get an integer number s_{id} for each (x, y) block, where $id = x_1y_1x_2y_2\dots x_by_b$ is generated using bit-interleaving of the binary representation of $x = x_1x_2\dots x_b$ and $y = y_1y_2\dots y_b$. Thus, in the spatial domain each trajectory is represented as a list of spatial ids. Figure 5.3(a) shows the z-order of the quadtree blocks, which are numbered as b_1, b_2, \dots, b_{16} . Similarly, each timestamp of a trajectory is mapped to a time bucket and assigned a temporal id ω_{id} . Figure 5.3(b) shows $\omega_{id}: \omega_1, \omega_2, \dots, \omega_8$ of different timestamps of all trajectories. Thereby, each trajectory can be represented as a sequence of (s_{id}, ω_{id}) tuples. For instance, t_1 is represented as $(b_1, \omega_3), (b_2, \omega_4), (b_5, \omega_5), (b_6, \omega_6)$.

5.3.3 R-tree Based Grouping

After the trajectories are transformed spatio-temporally, we group them using an R-tree. Each set of points originating from the same trajectory in the new coordinates is represented as an MBR. The trajectories corresponding to the nearby MBRs have co-visiting property, i.e., they tend to travel to the same/nearby locations at similar times. We group such trajectories by employing R-tree to group their MBRs at its leaf. The leaf nodes of the R-tree corresponds to disk-pages. Figure 5.4 shows the grouping of different trajectories in the transformed space using an R-tree. Here we can see points of the trajectories in different colors, and each set of points of a single trajectory is represented as an MBR. These MBRs are grouped together to form an R-tree where each leaf level node, R_1, R_2 etc. corresponds to a disk-page.

Each trajectory now has a *disk-page id* where it is physically stored. This *disk-page reference* is stored at leaf-blocks of the quadtree containing any point of the trajectory. More specifically, at each quadtree block, we maintain a list of temporal ids denoting the time range of trajectory points and the *disk-page reference* is mapped to the appropriate temporal id so that during query processing, spatio-temporal pruning of trajectory blocks is possible.

Example:

Let us consider the example of Figure 4.1 to demonstrate our solution to the BDPT query. Assume we have constructed both the SQ-index (Figure 5.1, 5.2) and the STCoV-index (Figure 5.5) using these trajectories. Suppose, the packet delivery request source (S) and destination (D) spatially maps to the SQ-index blocks 0.2 and 3 respectively, (Figure 5.1). The outgoing blocks of 0.2 include block 0.3, 2 while the incoming blocks of 3 are 0.3, 1.3, 2 (Figure 5.2(a), (b)). Block 0.2 is added to the explored outblocks, $O_{explored}$ while block 3 is added to the explored inblocks, $I_{explored}$ after the first iteration. Suppose, the most potential outblock calculated next is 0.3 and

the most potential inblock is 2. So, these blocks are added to the respective explored block lists while their neighbors are enqueued for gradual exploration. The overlap count increases to 1 assuming block 2 is retrieved from the outblock pool in the next iteration. Assume, the process terminates when overlap count reaches 2. In the reverse expansion of incoming blocks, say, node 0.3 is retrieved from the outblock pool as the most potential candidate. As a result, overlap count reaches 2, and we have block 0.2, 2, 0.3, 3 from the SQ-index contributing to the packet delivery request. The potential of SQ-index blocks may depend on distance from source/destination, trajectory density and so on. We run range query on STCoV-index with the blocks 0.2, 2, 0.3, 3 found from SQ-index. They overlap with STCoV-index blocks $b_3, b_4, b_9 \dots b_{16}$. Since ω_1 and ω_6 spans from 15:00 to 16:00 and 20:00 to 21:00 containing earliest packet pickup time (15:30) and the latest delivery time (21:00) respectively, we search for time buckets ω_1 to ω_6 in each of these STCoV-index blocks. For example, b_3 block has two trajectory points, one from t_2 , falling in ω_2 time bucket and the other from t_3 , falling in ω_5 time bucket. As a result, (b_3, ω_2) holds the disk page id R_2 containing t_2 and (b_3, ω_5) also holds R_2 as it contains t_3 as well. Note that, if there are time buckets, in a STCoV-index node, that are temporally disjoint from the packet delivery time window, the disk pages associated with such tuples can safely be pruned. Finally, we retrieve the disk pages containing the trajectories and construct *TrajGraph* with them. In our example, R_1, R_2, R_3 disk pages, i.e., $t_1 \dots t_6$ trajectories are retrieved. We exclude any trajectory that may further be temporally in compliant and construct *TrajGraph* with the remaining ones. If we consider time as the cost metric for convenience, we see that, deliverer t_3 can take the packet to destination at 20:30 while t_2 and t_5 combinedly can take it to destination at 19:30. So the best scheme would be to send the packet via deliverer t_2 and t_5 .

It is important to note that in our proposed index, we only use R-tree to organize the spatio-temporally co-located objects in the disk, and then use the disk references in the quadtree index. Thus, during the query processing time, it does not need to intersect with large number of R-tree nodes. The quadtree based partitioning of the space can easily prune the space based on a given query, and when it comes to retrieve the candidate objects (that meets with the query object), the system can fetch it using from the disk page in a single pass.

Chapter 6

Experimental Evaluation

In this chapter, we present the experimental evaluation for our solution to answer the BDPT query. To the best of our knowledge, there is no prior work that answers the BDPT query in a whole road network space supporting multiple modes of transports considering commuter trajectories. We, therefore, compare our approaches with a baseline. Specifically, we compare the following three methods.

(i) Baseline (BL): In this method, the trajectories are indexed using only a traditional spatial index. Specifically, the trajectories visiting the bounding box of P_{req} considering the source and destination locations, are retrieved by executing a range query in a quadtree. Then *TrajGraph* is constructed with all the retrieved trajectories and processed for finding a sequence of best deliverers and keepers. (ii) SQ-index (SQ): We use our proposed SQ-index to summarize trajectories based on spatial orientation. Using Algorithm 3, we exploit the SQ-index for pruning disk blocks containing trajectories at the first level. We then execute a spatial range query on the traditional index to retrieve relevant trajectories and construct *TrajGraph* with them. (iii) SQ_STCoV-index (SQST): In addition to the SQ-index, we use STCoV-index instead of a traditional index. After the spatial pruning using SQ-index, the trajectories are further pruned with STCoV-index. This shows the merits of our two index structures.

6.1 Experimental Settings

We implemented our data structures and algorithms in Java (JDK 1.8) and conducted the experiments in a PC equipped with Intel core i5-3570K processor and 8 GB of RAM. We design our experiments to estimate performance in disk-based solution although we actually use in-memory data structures in the experiments.

6.1.1 Trajectory Datasets

We use a public transport trip dataset of Melbourne (Melbourne Myki) [48] and a taxi trip dataset along with the public transport network of New York City [49] to evaluate the performance of our methods. We perform the necessary preprocessings on them and pick a smaller temporal window to work with homogeneous travel patterns.

Myki Dataset

We use the user trajectories of five weekdays from the Myki dataset for running experiments. Specifically we extract the public transport trips of Melbourne on the weekdays of the last week of June, 2018. The dataset contains user id, transport mode, vehicle, route, stoppage, event, timestamp, latitude, longitude and so on where we construct the trajectories grouping by user id and sorting by the events of getting on and getting off the vehicles. Moreover, we cluster the nearby trajectory points to remove the redundant stoppage ids. From this dataset, we obtain over half a million multipoint trajectories (554,650 to be exact) of Myki commuters with bus, train and tram as their modes of transport. Besides, we find a total of 22,345 stoppages along the transport routes inside the city of Melbourne.

NYC Taxi Dataset

The NYC Taxi dataset consists of taxi trips of users. Unlike the Myki dataset, the trips in this dataset are represented by pickup-dropoff pairs. Each row in the dataset contains passenger id, pickup location (longitude and latitude) and time, dropoff location and time, fare, vehicle no. and so on representing a two-point taxi trajectory. We extract the taxi trips of two weekdays, specifically, January 6 and January 7, 2016 to run our experiments. Thus we extract around 697,248 raw taxi trips subject to further preprocessing.

As our main focus is working with public transports, we map the taxi trips to the existing public transport network of New York City [50] without changing the timestamps. For this mapping, we pick the a bus stoppage within a close range (say, 100 meters) from each end point of a taxi trip. We discard the trajectories that do not map to any bus stops. Moreover, we cluster the nearby stops to the busier ones to ensure higher trajectory density at the stops. Thus we synthetically transform two point taxi trajectories to public transport trips. However, these trips still consists of two end points only, and therefore, the experiments show slightly different outcomes on this dataset compared to that of Myki public transport trip data.

Packet Keeper Generation

For the packet keepers, we pick some busy stoppages from the datasets and choose some nearby locations as the packet source and the destination randomly. To identify the busy stops, we

Parameters	Ranges
Dataset	MM , NYC
# of Trajectories (T)	100k, 250k, 400k, 550k
Keepers as % of stoppages (k)	10, 25, 50 , 100
P_{req} duration in hours (τ_P)	0.5-1, 1-2, 2-4 , 4-8, 8-24
Max. detour in km (δ)	0.5, 1 , 2, 4

Table 6.1: Parameters for Experiments on Myki and NYC Datasets

observe the frequency of visiting the stops by the daily commuters along their typical travel itineraries at different times. Besides, we randomly pick a time from the pick hours at the packet source as the earliest pickup time for the packet and vary the packet duration to identify the latest delivery time.

6.1.2 Performance Evaluation and Parameterization

We have studied the efficiency, effectiveness, and scalability of our proposed approaches and compared with the baseline. We have varied several parameters for comparison. Table 6.1 shows the parameters and their range of values where the default values are shown in bold. For each of the configurations, only one parameter is varied while the others are fixed at their the default values. Note that, we use this parameter setting for the default Myki dataset. We modify the default packet duration for the NYC dataset since it basically contains two point trajectories and thereby exhibits lower success rate in packet delivery for the previous setting. We denote the Myki dataset with *MM* and the NYC Taxi dataset with *NYC*.

We have studied the impact of each parameter on the runtime and the total number of blocks accessed for measuring efficiency and scalability respectively. As argued in [51], reporting the actual I/O cost may be misleading not only because the I/O cost largely depends on the disk type but also because it is significantly affected by various factors many of which are system dependent or not easily controllable. Therefore, we report the number of blocks accessed (i.e., # of I/Os) instead of the I/O cost. As the STCoV-index in our approach saves R-tree leaves which directly correspond to disk blocks, the task of measuring number of I/Os when SQ-STCoV-index is used is achieved by counting distinct R-tree leaves accessed. In the baseline and SQ-index method where we use a traditional quadtree index, we make a simple assumption that the trajectories whose first points are contained in the same quadtree nodes, can be grouped together or in the nearby disk blocks. We use the same value for the trajectory capacity of disk blocks as that of an R-tree leaf in the STCoV-index. Thus each trajectory id has a corresponding virtual disk block id which contributes to measuring the number of I/Os.

We generate 100 sets of packet delivery request queries with the same settings and report the average performance. For each set of query generation, we randomly choose two stoppages for

packet request with the required criteria (e.g., specific distance).

Our proposed summary index, SQ-index may provide a sub-optimal solution if a packet cannot be delivered with the pruned trajectories but joining some distant trajectories can deliver it. So, we also demonstrate the effectiveness of our solutions in terms of percentage of packets delivered. Some packets can indeed be undeliverable for the directional nature of the trajectories due to their temporal attributes. Alternatively, if the packet source and destination are parts of separate connected components of the *TrajGraph* a packet cannot be delivered given an arbitrary amount of time. So, the effectiveness is more accurately measured in terms of percentage of deliverable packets delivered. We retrieve and run path finding algorithms for all the trajectories to guarantee whether a packet is deliverable or not. While we find the effectiveness of both our approaches using SQ-index and SQ_STCoV-index identical to this exact solution for smaller cases, we exclude the comparison as the other performance metrics of the exact solution would be far worse than that of the baseline.

6.1.3 Experimental Results on Myki Dataset

We vary different parameters and present the CPU time and number of I/Os in the following.

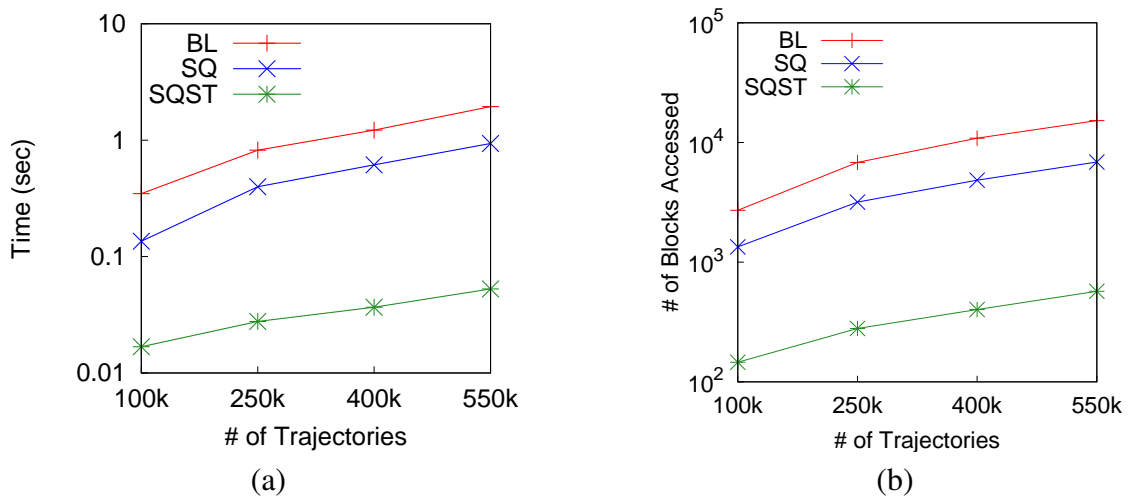


Figure 6.1: Evaluating BDPT for varying number of trajectories in MM dataset

(i) No. of trajectories: We vary the number of public transport trips in the MM dataset from 100k to 550k choosing randomly from the available trips of a week. Figures 6.1(a) & (b) show the average processing time, and # of I/Os, respectively, for the Baseline, SQ-index, and SQ_STCoV-index.

We observe that the SQ-index is twice as good as Baseline in terms of both processing time and number of block access. As SQ-index summarizes the directional orientation of the trajectories before indexing points in a quadtree (as done in Baseline), it can prune nearly half of the trajectory dataset. The spatio-temporal grouping of the trajectories in the STCoV-index of

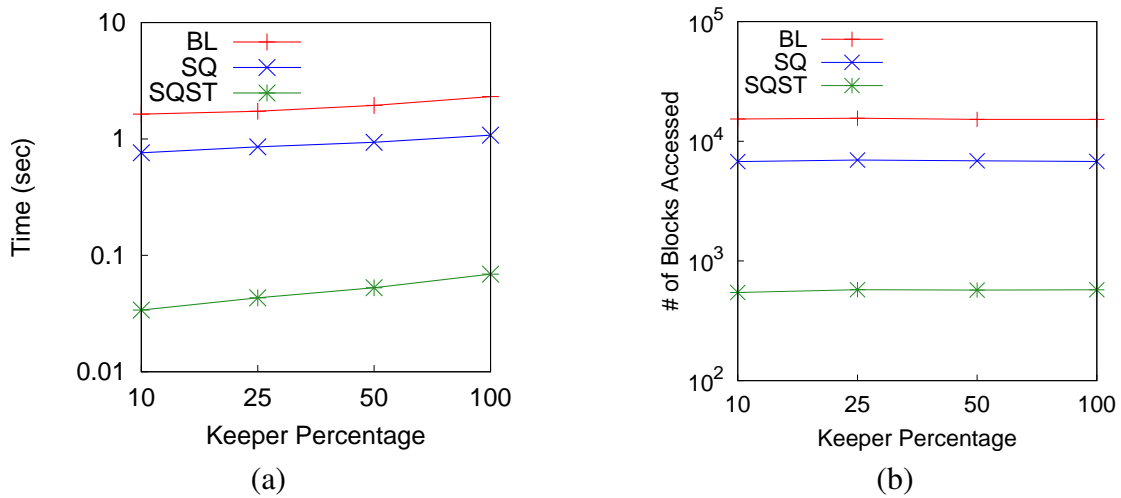


Figure 6.2: Evaluating BDPT for varying % of packet keepers in MM dataset

SQ_STCoV-index results in 1 order of magnitude faster processing time and I/O overhead than SQ-index. This is because the pruned regions from STCoV-index still contains many trajectories with disjoint timestamps with the packet delivery time window. Since working with a larger trajectory dataset requires more processing, we observe an increase in runtime and # of I/Os with the increase in the number of trajectories for all the approaches.

(ii) Percentage of packet keepers: We choose a subset of available stops as packet keepers. We vary the ratio of the cardinality of this set of keepers to the number of stops from 10% to 100%, and report the average processing time & # of I/Os to deliver a packet. The results (Figure 6.2) show that SQ-index is about two times better compared to Baseline whereas SQ_STCoV-index outperforms SQ-index by more than 1 order of magnitude in terms of both the CPU time and # of I/Os. Since keepers have little contribution in trajectory retrieval, # of I/Os remains unchanged for all three methods. The gradual increase of runtime of all of the approaches gradually increase with the number of stops, as more users can contribute in the delivery process. However, the I/Os in each approach remain almost constant because trajectory retrieval depends mainly on the number of trajectories and packet duration while it depends little on the keepers.

(iii) Detour threshold (δ): We intuitively expect more trajectories to be joined via keepers with the increase of δ . Our experimental findings supports the intuition resulting in higher runtime for processing the BDPT query. The results (Figure 6.3) demonstrate SQ-index is almost twice as good as Baseline in terms of efficiency (i.e. query processing time and number of block access). Moreover, SQ_STCoV-index is around 1 order of magnitude more efficient than Baseline.

(iv) Packet Duration: Since the number of candidate trajectories to deliver a packet increases with the increase in packet delivery request duration, the runtime and block access of SQ_STCoV-index increases because it retrieves larger number of trajectories from the SQ-index. However, based on the packet duration, we cannot prune more trajectories from SQ-index or Baseline as they do not have a temporal index and thereby no temporal filtering scheme before trajectory

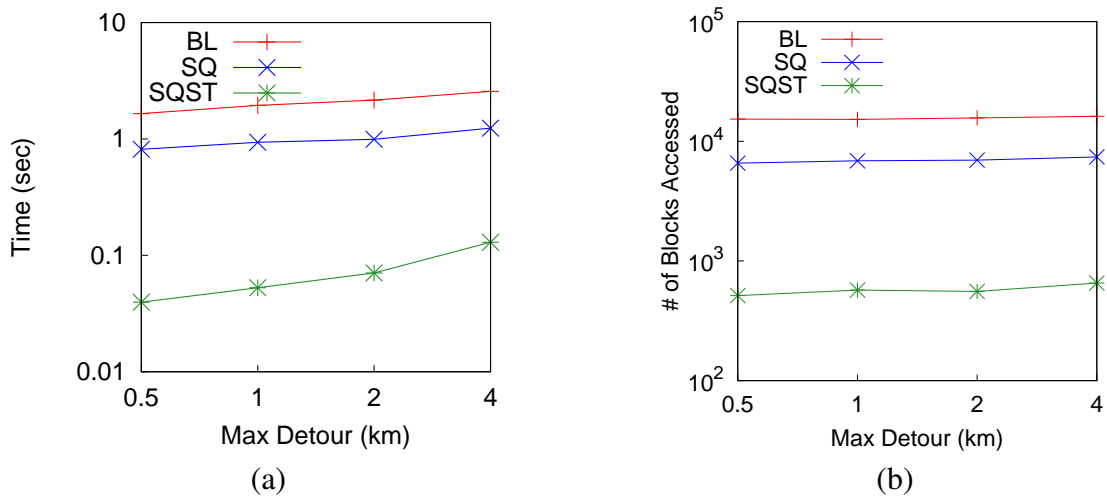
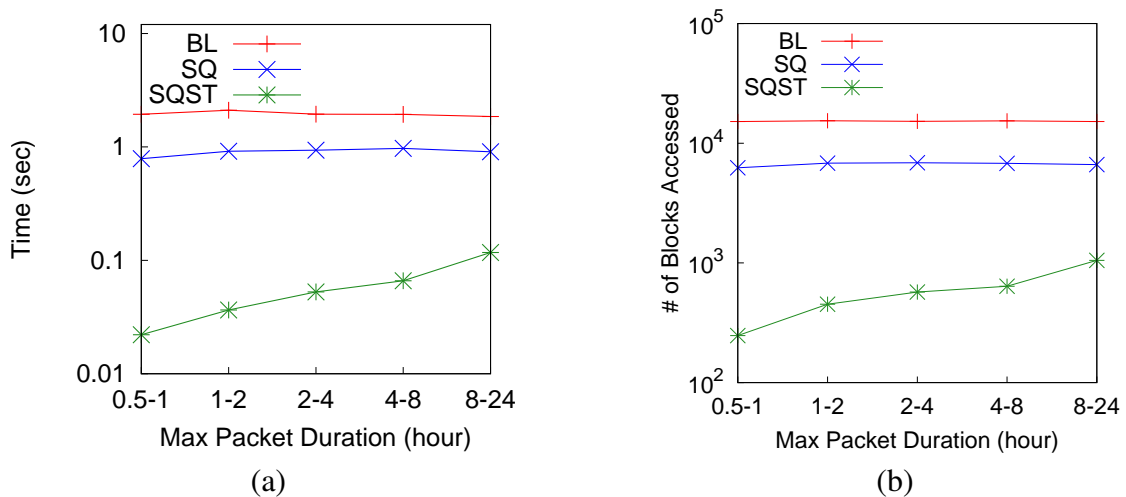
Figure 6.3: Evaluating BDPT for varying detour distance (δ) in MM dataset

Figure 6.4: Evaluating BDPT for varying packet duration in MM dataset

retrieval. As a result, we find their curves almost horizontal. Figure 6.4 shows that SQ-STCoV-index still outperforms the Baseline in terms of runtime and # of I/Os by 1 order of magnitude while only SQ-index can reduce them to half of those of the Baseline.

Packet Delivery Rate & Cost-Ratio: To demonstrate the effectiveness of the BDPT query, we measure the percentage of packets successfully delivered while varying the keeper percentage, detour threshold (δ) and packet duration. We also measure the cost of the packet deliveries which are successful in all three methods. Otherwise, the cost comparison is not meaningful since we consider it to be infinite in case of an unsuccessful delivery. We use the term *cost-ratio* to denote the relative delivery cost with respect to the baseline and report it for the successful deliveries. We find that our approaches can achieve the same delivery rate and cost as the baseline (Figure 6.5, 6.6). We have also achieved the same delivery rate and cost for different methods varying the number of trajectories. But unlike the other parameters, it does not show an increasing trend possibly because their coverage area does not change much. So, we omit the result for brevity.

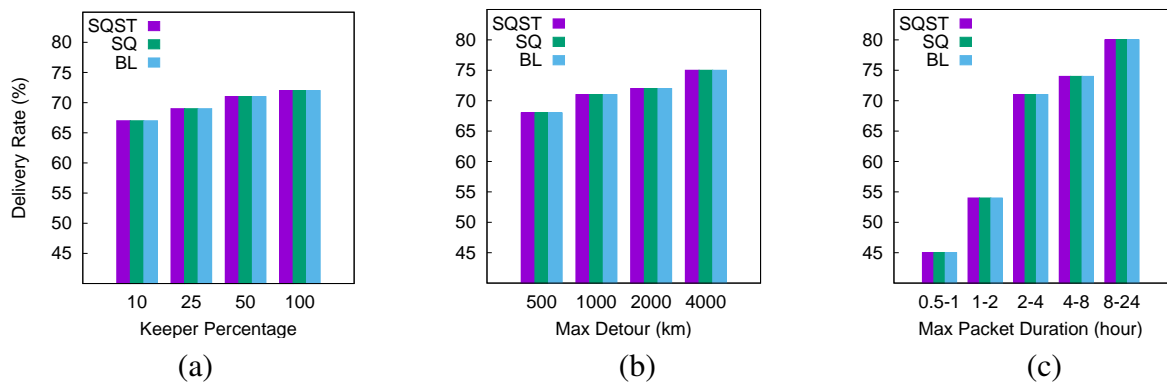


Figure 6.5: Evaluating packet delivery rate of BDPT for varying the percentage of keepers, the detour threshold δ , and the packet duration in MM dataset

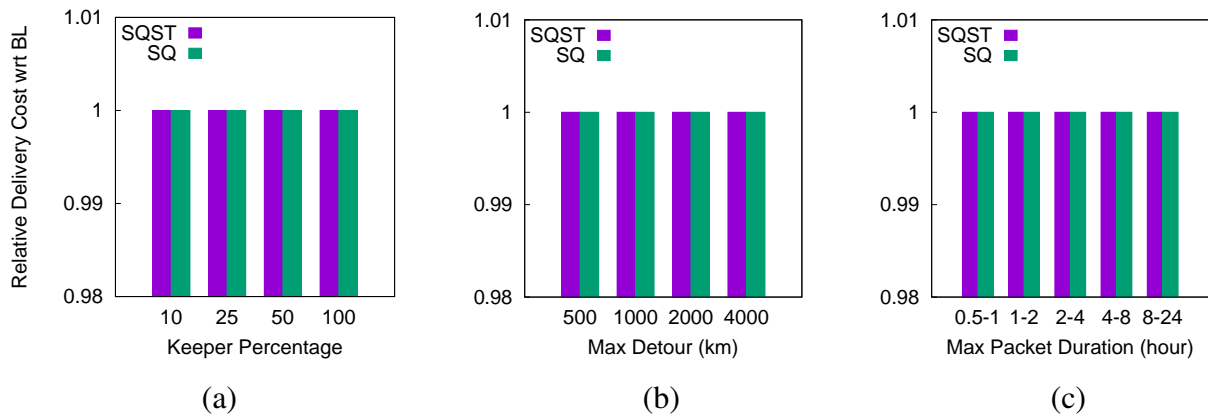


Figure 6.6: Evaluating cost of successful deliveries of BDPT for varying the percentage of keepers, the detour threshold δ , and the packet duration in MM dataset

6.1.4 Experimental Results on NYC Dataset

Similar to our approach for the Myki dataset, we present the results of our experiments on the NYC dataset in the following.

(i) *No. of trajectories*: We choose between 100k to 550k two-point NYC taxi trips mapped to the public transport stoppages randomly from the available trajectories of two weekdays. Figures 6.7(a) & (b) show the average processing time, and # of I/Os, respectively, for the Baseline, SQ-index, and SQ_STCoV-index.

In terms of I/O, the performance is as good as that of Myki dataset as shown in the Figure 6.7(b) since SQ-index requires around half the number of block accesses compared to Baseline and SQ_STCoV-index outperforms Baseline by an order of magnitude. But in terms of the runtime, SQ-index achieves around 25% decrease while SQ_STCoV-index performs twice as fast as the Baseline as Figure 6.7(a) demonstrates (note that, the vertical axis of is in linear scale unlike that in log scale in the Figure 6.1(a)). The relatively lower performance gain in runtime is possibly because of working with two-point trajectories and mapping them a limited number of busy stoppages. However, we have calculated the runtime independently from the impact of the

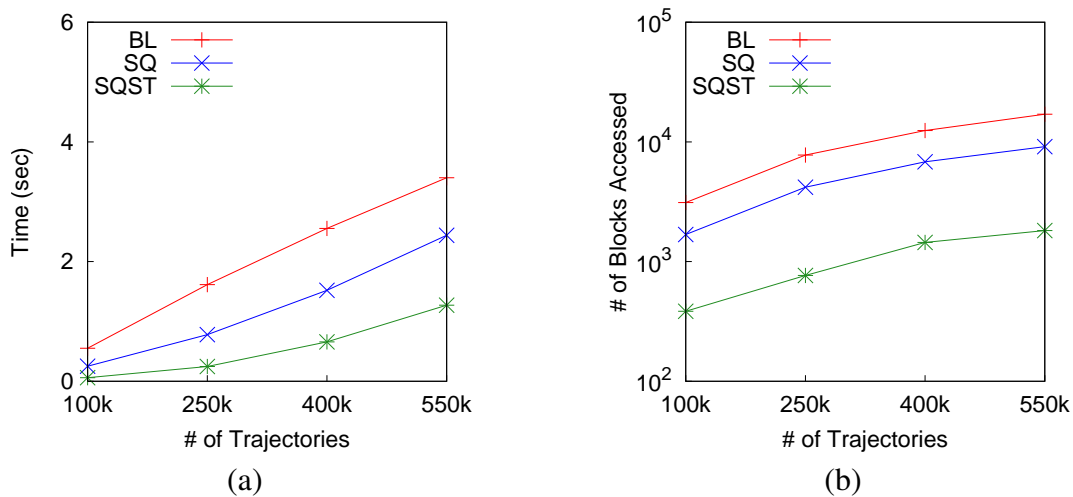


Figure 6.7: Evaluating BDPT for varying number of trajectories in NYC dataset

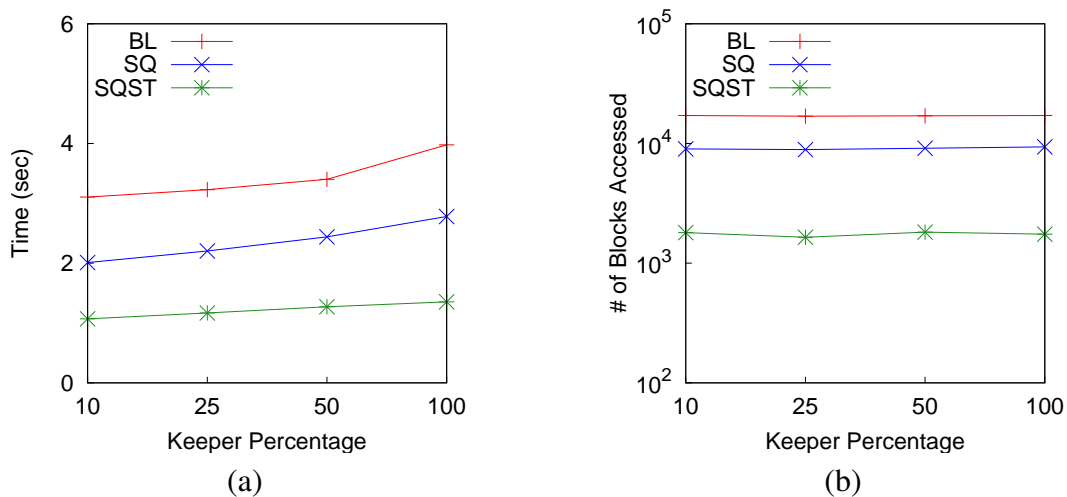


Figure 6.8: Evaluating BDPT for varying % of packet keepers in NYC dataset

I/O cost. Since the I/O cost is a major bottleneck in disk based systems, the performance of SQ_STCoV-index would be even better if the actual runtime considering trajectory retrieval from physical disks into account.

Besides, we see an increasing trend in the runtime and the # of blocks accessed with the increase in the number of trajectories similar to that of the Myki dataset.

(ii) Percentage of packet keepers: We vary the ratio of packet keeper count to the total stoppage count from 10% to 100%, and show the average processing time & # of I/Os to deliver a packet. The results (Figure 6.8) show that SQ-index performs approximately 25% faster than Baseline and SQ_STCoV-index outperforms Baseline by around half the amount of CPU time. The I/O performance is however similar to the results for Myki dataset (Figure 6.2(b)).

(iii) Detour threshold (δ): We vary the detour threshold of the commuters from 500m to 4km and find that although the runtime of SQ-index shows slightly irregular pattern, it does not get worse than Baseline while SQ_STCoV-index outperforms Baseline by 50% of runtime

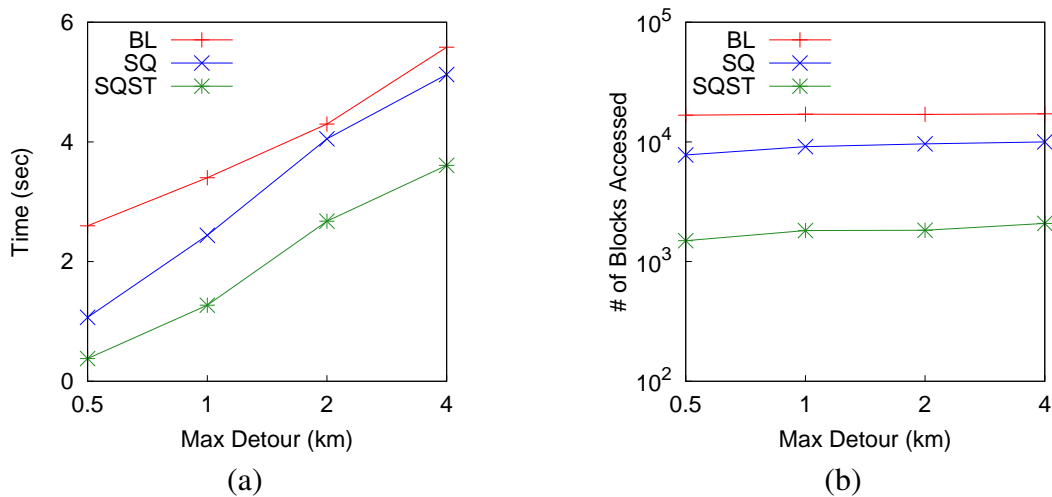
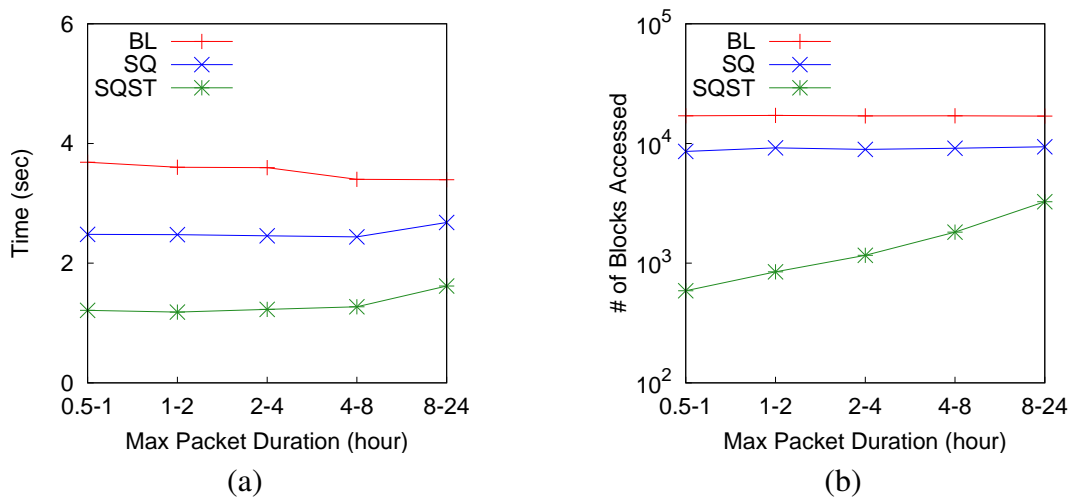
Figure 6.9: Evaluating BDPT for varying detour distance (δ) in NYC dataset

Figure 6.10: Evaluating BDPT for varying packet duration in NYC dataset

(Figure 6.9). The results of measuring I/O is similar to that of varying the other parameters (e.g., an order of magnitude performance gain by SQ_STCoV-index over the Baseline and around 50% gain over the Baseline by SQ-index).

(iv) Packet Duration: Figure 6.10 shows that SQ_STCoV-index still outperforms the Baseline in terms of # of I/Os by an order of magnitude and in terms of runtime by approximately 50%. The SQ-index too, follows results of experiments on Myki dataset in terms I/O while it achieves nearly 25% faster runtime than the Baseline. Note that, the packet duration has little impact on the performance of Baseline and SQ-index since both use a traditional spatial index as explained in the results on Myki dataset. Thus their runtime and I/O curves are almost horizontal. In fact, the decrease in the runtime of Baseline for higher duration is because of higher successful deliveries. The runtime and I/O of SQ_STCoV-index increases in case of upto 24 hours packet duration. This is because we have total 48 hours of temporal span from the trajectories (since trajectories of 2 weekdays are considered) and when we consider around half of the total window

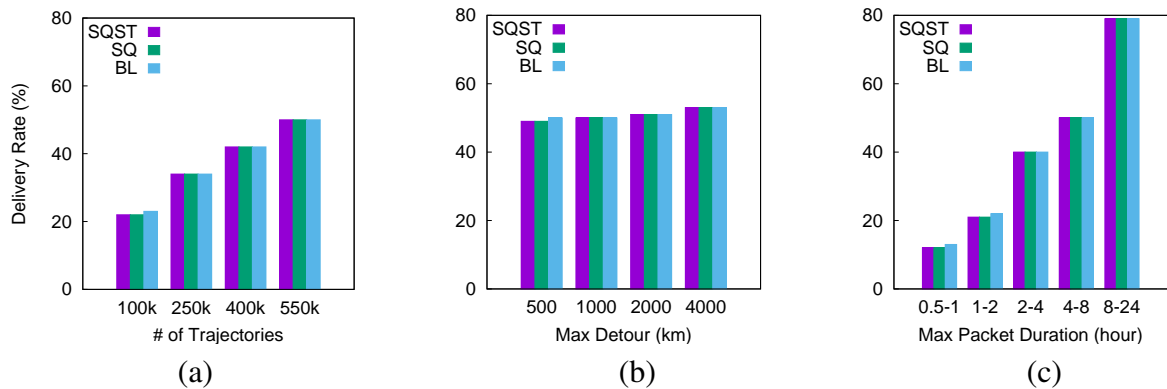


Figure 6.11: Evaluating packet delivery rate of BDPT for varying the number of trajectories, the detour threshold δ , and the packet duration in NYC dataset

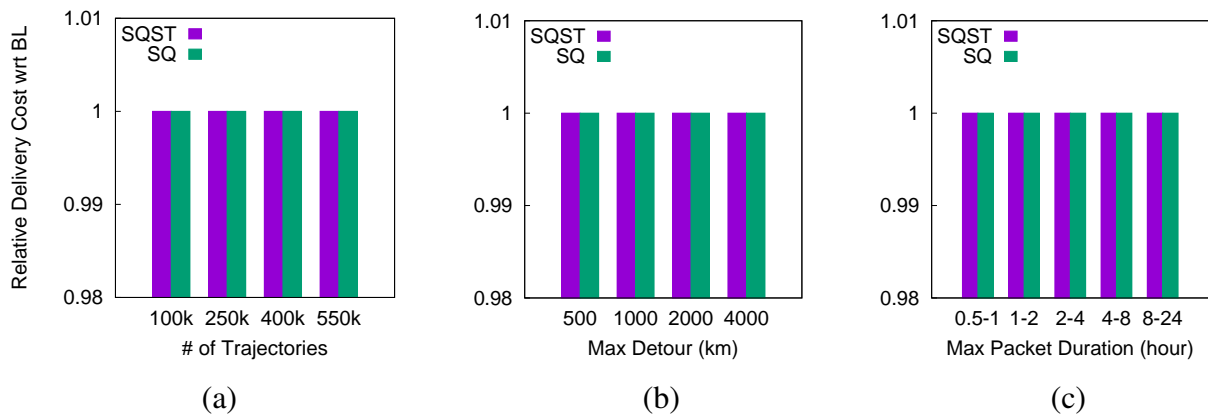


Figure 6.12: Evaluating cost of successful deliveries of BDPT by varying the number of trajectories, the detour threshold δ , and the packet duration in NYC dataset

in the packet duration, the temporal pruning capability gets curtailed.

Packet Delivery Rate & Cost-Ratio: We measure the packet delivery rate, i.e., percentage of packet delivered from the packet source to destination within the delivery time window and the cost-ratio, i.e., the relative cost in successful deliveries with respect to the baseline to assess the effectiveness of our solution to the BDPT query on the NYC dataset. We vary the number of trajectories, detour threshold (δ) and packet duration and find that our technique achieves nearly the same success rate as the baseline (Figure 6.11) and exactly the same cost-ratio (Figure 6.12). Specifically, with the increase in the number of trajectories the packet delivery rate reaches from 22% to 50% demonstrating the merits of more commuters involved in the delivery process (Figure 6.11(a)). Besides, with the increase in packet duration from 0.5-1 hours to 8-24 hours, the success rate increases to 12% to 79%. The increase in the maximum detour, however, impacts the delivery rate in the NYC dataset less than that of the other parameters, possibly because of our synthetic mapping of the taxi trajectories to the public transport stops. We get a delivery rate between 49% to 53% when we vary the detour from 500m to 4km. The cost incurred in SQ-index and SQ_STCoV-index are the same as that of Baseline in case of successful deliveries

as found experimentally.

We have also achieved nearly the same delivery rate and cost for Baseline, SQ-index and SQ_STCoV-index (around 50% with at most 2% deviation across the methods) while varying the percentage of keepers. We have omitted the result to avoid any confusion as it does not show an increasing trend, possibly because not requiring more keepers for joining commuter trajectories which are already mapped to the busy stoppages.

Note that, in all the aforementioned cases, the SQ-index and SQ_STCoV-index delivery rates are same while they lag the delivery rate of Baseline by a small margin (1% for example). This is because pruning the trajectories from the spatial connectivity network through SQ-index can sometimes prune a candidate solution that may contribute to packet delivery through distant locations. Besides, when the packets are deliverable in all three methods, we find the cost of SQ-index and SQ_STCoV-index exactly the same as that of Baseline. So the approximation loss, as found from the experiments is negligible.

Chapter 7

Conclusion

We have introduced the crowdshipping problem as a location oriented online query (BDPT). In this query, we find a delivery path comprising of daily commuter trajectories and packet keepers with an objective of minimizing cost given a packet delivery request. While the state-of-the-art techniques solve the problem offline or for taxi OD pairs only, we have addressed it as a trajectory database problem with commuter trajectories for engaging more users. We have then proposed a baseline solution to the BDPT query as a trajectory matching problem based on an informed search for path finding in trajectory graph containing information of the trajectory points and segments. For scalability of the solution, both in terms of CPU time and number of I/Os, we have introduced two novel quadtree-based indexes for pruning the candidates for the graph construction. We have exploited the spatial connectivity of regions through the trajectories and the trajectory co-visiting patterns in these indexes. By combining these two indexes, we have found the performance in terms of CPU time and number of I/Os to improve by an order of magnitude compared to the baseline. Besides, we have discussed the generalization of our solution to finding the m best paths in the m -BDPT query.

Although we have formulated the problem as an online query, we have not considered the capacity of the commuters, size of the packets etc. while delivering a packet. There are scopes for novel works incorporating these additional constraints in the online trajectory matching. Besides, while considering the historic itinerary profile of the commuters, the likelihood of predicted future trajectories can be taken into account to maximize the throughput. Finding and deploying a suitable probability aggregation model for this probabilistic trajectory matching may have a good potential in future researches. Moreover, the trajectory indexing schemes we have proposed may not be limited to our proposed query only. Their applicability, in general, to other query problems calls for further independent studies.

References

- [1] S. Ma, Y. Zheng, and O. Wolfson, “T-share: A large-scale dynamic taxi ridesharing service,” in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 410–421, IEEE, 2013.
- [2] C. Chen, D. Zhang, Z.-H. Zhou, N. Li, T. Atmaca, and S. Li, “B-planner: Night bus route planning using large-scale taxi gps traces,” in *2013 IEEE international conference on pervasive computing and communications (PerCom)*, pp. 225–233, IEEE, 2013.
- [3] Y. Liu, C. Liu, N. J. Yuan, L. Duan, Y. Fu, H. Xiong, S. Xu, and J. Wu, “Exploiting heterogeneous human mobility patterns for intelligent bus routing,” in *2014 IEEE International Conference on Data Mining*, pp. 360–369, IEEE, 2014.
- [4] S. R. B. Gummidi, T. B. Pedersen, and X. Xie, “Transit-based task assignment in spatial crowdsourcing,” in *32nd International Conference on Scientific and Statistical Database Management*, pp. 1–12, 2020.
- [5] C. Bassem, “Mobility coordination of participants in mobile crowdsensing platforms with spatio-temporal tasks,” in *Proceedings of the 17th ACM International Symposium on Mobility Management and Wireless Access*, pp. 33–40, 2019.
- [6] M. Placek, “Amazon logistics: Package volume in the u.s.” url=<https://www.statista.com/statistics/1178979/amazon-logistics-package-volume-united-states/>, Apr 2022. Online, Accessed: 2022-09-24.
- [7] M. Placek, “Fedex express: Total average daily packages 2022.” url=<https://www.statista.com/statistics/878354/fedex-express-total-average-daily-packages/>, Sep 2022. Online, Accessed: 2022-09-24.
- [8] M. Placek, “Fedex’s revenue 2022.” url=<https://www.statista.com/statistics/267501/revenue-of-fedex/>, Jul 2022. Online, Accessed: 2022-09-24.
- [9] “Uber announces results for first quarter 2022.” url=<https://investor.uber.com/news-events/news/press-release-details/2022/Uber-Announces-Results-for-First-Quarter-2022/default.aspx>, 2022. Online, Accessed: 2022-09-25.

- [10] “Uber announces results for second quarter 2022.” url=<https://investor.uber.com/news-events/news/press-release-details/2022/Uber-Announces-Results-for-Second-Quarter-2022/default.aspx>, 2022. Online, Accessed: 2022-09-25.
- [11] “Become a delivery driver using uber eats.” url=<https://www.uber.com/us/en/deliver/>, 2022. Online, Accessed: 2022-09-25.
- [12] D. C. Everest and A. Wheelwright, “Amazon job opportunities.” url=<https://hiring.amazon.com/job-opportunities/flex-driver-jobs/>. Online, Accessed: 2022-09-24.
- [13] “Become a rider.” url=<https://www.foodpanda.com/careers/riders/>. Online, Accessed: 2022-09-25.
- [14] C. Archetti, M. Savelsbergh, and M. G. Speranza, “The vehicle routing problem with occasional drivers,” *European Journal of Operational Research*, vol. 254, no. 2, pp. 472–480, 2016.
- [15] C. Chen, S.-F. Cheng, A. Gunawan, A. Misra, K. Dasgupta, and D. Chander, “Traccs: a framework for trajectory-aware coordinated urban crowd-sourcing,” in *Second AAAI Conference on Human Computation and Crowdsourcing*, 2014.
- [16] A. M. Arslan, N. Agatz, L. Kroon, and R. Zuidwijk, “Crowdsourced delivery—a dynamic pickup and delivery problem with ad hoc drivers,” *Transportation Science*, vol. 53, no. 1, pp. 222–235, 2019.
- [17] G. Macrina, L. Di Puglia Pugliese, F. Guerriero, and D. Laganà, “The vehicle routing problem with occasional drivers and time windows,” in *Optimization and Decision Science: Methodologies and Applications*, pp. 577–587, 2017.
- [18] C. Chen, S. Yang, Y. Wang, B. Guo, and D. Zhang, “Crowdexpress: a probabilistic framework for on-time crowdsourced package deliveries,” *IEEE transactions on big data*, 2020.
- [19] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [20] R. Nandal, “Spatio-temporal database and its models: a review,” *IOSR J. Comput. Eng.*, vol. 11, no. 2, pp. 91–100, 2013.
- [21] “Spatial data.” url=<https://www.sciencedirect.com/topics/engineering/spatial-data>, 2022. Online, Accessed: 2022-10-20.

- [22] A. Zola and M. Fontecchio, “What is spatial data and how does it work?.” url=<https://www.techtarget.com/searchdatamanagement/definition/spatial-data>, Jun 2021. Online, Accessed: 2022-10-20.
- [23] “Database globalization support guide.” url=https://docs.oracle.com/cd/E11882_01/, Jul 2013. Online, Accessed: 2022-10-20.
- [24] “Spatiotemporal data.” url=<https://www.sciencedirect.com/topics/computer-science/spatiotemporal-data>. Online, Accessed: 2022-10-20.
- [25] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, *Spatial Indexing Techniques*, pp. 2702–2707. Boston, MA: Springer US, 2009.
- [26] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57, 1984.
- [27] H. Samet, “The quadtree and related hierarchical data structures,” *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, 1984.
- [28] P. van Oosterom and T. Vrijlbrief, “The spatial location code,” in *Proceedings of the 7th international symposium on spatial data handling, Delft, The Netherlands*, pp. 12–16, 1996.
- [29] G. Macrina, L. Di Puglia Pugliese, F. Guerriero, and G. Laporte, “Crowd-shipping with time windows and transshipment nodes,” *Computers & Operations Research*, vol. 113, p. 104806, 2020.
- [30] T. V. Le, A. Stathopoulos, T. Van Woensel, and S. V. Ukkusuri, “Supply, demand, operations, and management of crowd-shipping services: A review and empirical evidence,” *Transportation Research Part C: Emerging Technologies*, vol. 103, pp. 83–103, 2019.
- [31] V. Gatta, E. Marcucci, M. Nigro, S. M. Patella, and S. Serafini, “Public transport-based crowdshipping for sustainable city logistics: Assessing economic and environmental impacts,” *Sustainability*, vol. 11, no. 1, p. 145, 2018.
- [32] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, “Spatio-temporal indexing for large multimedia applications,” in *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pp. 441–448, 1996.
- [33] D. Pfoser, C. S. Jensen, Y. Theodoridis, *et al.*, “Novel approaches to the indexing of moving object trajectories.,” in *VLDB*, pp. 395–406, 2000.

- [34] M. E. Ali, S. S. Eusuf, K. Abdullah, F. M. Choudhury, J. S. Culpepper, and T. Sellis, "The maximum trajectory coverage query in spatial databases," *PVLDB*, vol. 12, no. 3, pp. 197–209, 2018.
- [35] C. Li, J. Chen, C. Jin, R. Zhang, and A. Zhou, "Mr-tree: an efficient index for mapreduce," *International Journal of Communication Systems*, vol. 27, no. 6, pp. 828–838, 2014.
- [36] G. Li and J. Tang, "A new hr-tree index based on hash address," in *2010 2nd International Conference on Signal Processing Systems*, vol. 3, pp. V3–35, 2010.
- [37] Y. Tao and D. Papadias, "The mv3r-tree: A spatio-temporal access method for timestamp and interval queries," in *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome, 2001*.
- [38] Z. Song and N. Roussopoulos, "Seb-tree: An approach to index continuously moving objects," in *International Conference on Mobile Data Management*, pp. 340–344, Springer, 2003.
- [39] V. P. Chakka, A. Everspaugh, J. M. Patel, *et al.*, "Indexing large trajectory data sets with seti.," in *CIDR*, vol. 75, p. 76, 2003.
- [40] S. Ranu, D. P. A. D. Telang, P. Deshpande, and S. Raghavan, "Indexing and matching trajectories under inconsistent sampling rates," in *ICDE* (J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, eds.).
- [41] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *SIGMOD*, pp. 491–502, 2005.
- [42] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis, "User oriented trajectory search for trip recommendation," in *EDBT*, pp. 156–167, 2012.
- [43] E. Frentzos, K. Gratsias, and Y. Theodoridis, "Index-based most similar trajectory search," in *ICDE*, pp. 816–825, 2007.
- [44] H. Wang, H. Su, K. Zheng, S. W. Sadiq, and X. Zhou, "An effectiveness study on trajectory similarity measures," in *ADC*, pp. 13–22, 2013.
- [45] S. Shang, R. Ding, K. Zheng, C. S. Jensen, P. Kalnis, and X. Zhou, "Personalized trajectory matching in spatial networks," *VLDB J.*, vol. 23, no. 3, pp. 449–468, 2014.
- [46] L. A. Tang, Y. Zheng, X. Xie, J. Yuan, X. Yu, and J. Han, "Retrieving k-nearest neighboring trajectories by a set of point locations," in *SSTD*, pp. 223–241, 2011.
- [47] Y. Han, L. Chang, W. Zhang, X. Lin, and L. Wang, "Efficiently retrieving top-k trajectories by locations via traveling time," in *ADC*, pp. 122–134, 2014.

-
- [48] T. Wright, “Exploring melbourne’s myki data with aws athena,” Jan 2019.
- [49] “Tlc trip record data.” url=<https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page>, 2022. Online, Accessed: 2022-10-20.
- [50] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, and G. Cong, “Reverse k nearest neighbor search over trajectories,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 4, pp. 757–771, 2018.
- [51] S. Yang, M. A. Cheema, X. Lin, and W. Wang, “Reverse k nearest neighbors query processing: experiments and analysis,” *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 605–616, 2015.

Generated using Postgraduate Thesis L^AT_EX Template, Version 1.03. Department of
Computer Science and Engineering, Bangladesh University of Engineering and
Technology, Dhaka, Bangladesh.

This thesis was generated on Monday 28th November, 2022 at 5:50am.