Scalable Spatial Predictive Query Processing for Moving Objects

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Abdeltawab M. Hendawi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY, PhD

Mohamed F. Mokbel

August, 2015

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisors Prof. Mohamed F. Mokbel for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. In addition to my advisor, I would like to thank the rest of my dissertation committee members, Prof. John Carlis, Prof. Tian He, and Dr. John Krumm, for their encouragement and support, and invaluable comments and feedback.

As a member of Data Management Lab, I would like to thank my fellow lab mates, Chi-Yin Chow, Mohamed Khalefa, Justin Levandoski, Biplob Debnath, Joe Naps, Mohamed Sarwat, Jie Bao, Ahmed Eldawy, Amr Magdy, Louai AlArabi, James Avery, Rami Alghamdi, Christopher Johnathan, and Ibrahim Sabek, for all the insightful discussions and feedback, and enjoyable friendships. I would like to thank Mohamed Ali and Ankur Teredesai, for their great mentorship and guidance during my academic training at the Center for Data Science at the University of Washington, Tacoma. I am also grateful to my research collaborator, Anas Basalamah, at the KACST GIS Technology Innovation Center at Umm Al-Qura University, for the fruitful collaboration.

Last but not least, I would like to express my deepest gratitude to my parents, brothers, sisters and friends. This dissertation would not have been possible without their warm love, continued patience, and endless support.

Dedication

This is by the grace of my Lord.

To my beloved wife Rasha, my son Omar, and my daughter Rana.

Abstract

A fundamental category of location based services relies on predictive queries which consider the anticipated future locations of users. Predictive queries attracted the researchers' attention as they are widely used in several applications including traffic management, routing, location-based advertising, and ride sharing. This thesis aims to present a generic and scalable system for predictive query processing on moving objects, e.g., vehicles. Inside the proposed system, two frameworks are provided to work on two different environments, (1) *Panda* framework for Euclidean space, and (2) *iRoad* framework for road network. Inside the *iRoad* system, a novel data structure named *Predictive Tree (P-Tree)* is proposed to index the anticipated future locations of objects on road networks. Unlike previous work in supporting predictive queries, the target of the proposed system is to: (a) support long-term query prediction as well as short term prediction, (b) scale up to large number of moving objects, and (c) efficiently support different types of predictive queries, e.g., predictive range, KNN, and aggregate queries.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The fact that there are more than one billion smart phones [1] triggered the massive explosion of location based services [2–5]. An important category of these services offers facilities based on the future location of a user rather than his/her location in the present time. Spatial queries in this categories come under the umbrella of predictive queries [6–8], where a service is supplied according to the predicted location of a user after some time in the future. Common types of predictive spatial queries include *predictive range* query, e.g., "find all hotels that will be located within two miles of a user's anticipated location after 30 minutes", *predictive KNN* query, e.g., "find the three taxis that most likely to pass by my location in the next 10 minutes", and *predictive aggregate* query, e.g., "how many cars expected to be around the stadium during the next 20 minutes".

In fact, *Predictive* queries can be employed in various types of real applications such as (1) traffic management, to predict areas with high traffic in the next half hour, so appropriate decisions can be taken before congestion appears,

(2) location-aware advertising, to distribute coupons and sales promotions to customers more likely to show up around a certain store during the sale time in the next hour, (3) routing services, that take into consideration the predicted traffic on each road to find the shortest path for a user trip starting after 15 minutes from the present time, (4) ride sharing systems, to get the drivers that mostly will pass by a rider's location within few minutes, (5) store finders, to predict the closest restaurants to a user's route after half hour, (6) emergency response, to alert the three police cars expected to be the nearest to a stolen car in a couple of minutes.

The goal of this PhD thesis is to enable the practical realization of location-based services such that they can support common types of predictive queries on spatio-temporal data, i.e., moving objects. Therefore, this thesis proposes a generic and scalable predictive spatial query processing system. Inside this system, two different frameworks are introduced, namely, *Panda* framework [9] and *iRoad* framework [10]. Each one is customized according to the underlying work environment, euclidean space and road network graph, respectively.

1.2 Challenges

Specifically, this thesis handles the following core challenges in processing of spatial predictive queries:

- **Prediction**. Unlike most of the existing related work [8, 11] that supports short-term prediction only, the proposed frameworks have the ability to evaluate long-term as well as short-term predictive queries. In addition, the employed prediction models do not rely on the historical data, as in many cases it is hard to obtain the historical data of the moving objects. For example, in new systems that there is no historical data or in confidential

systems where the data are top secrets or at least private so it can not be released to the prediction model.

- **Salability**. The proposed frameworks can scale up to support heavy query workloads on a space with a large number of moving objects. The scalability of *Panda* is resulted from adjusting the underlying prediction function to be employed to filter out the objects having no possibility to show up in the query region at the specified time. This filtering saves a lot of the processing time for each single query. While the scalability of *iRoad* comes from introducing a novel data structure named *reachability tree* to prune the space around each object. Yet, it holds only those nodes, road intersections, reachable within a specified time period from the object current location.
- **Efficiency**. The goal here is to introduce an efficient query processing engine that utilizes the prediction of each object in the underlying space to answer the predictive queries in very fast response time. Thus, users do not have to wait to get the answer to their queries.
- **Generality**. The introduced solution can support the processing of many kinds of predictive queries including *predictive range* query, *predictive KNN* query, *predictive aggregate* query, and *predictive point* query. This is done inside the running framework and using the same data structures and algorithms.

1.3 Thesis Road Map

- Chapter 1 gives an overview about spatial predictive queries and introduces the analytic goals pursued in this thesis .
- Chapter 2 briefly presents the related work, problems and solutions, in the area of handling spatial predictive queries.
- In Chapter 3 presents the *Panda* system for processing spatial predictive queries on Euclidean space.
- Chapter 4 presents both the P-Tree index structure and describes how it is integrated into the *iRoad* system to support predictive query processing on road network graphs.
- Chapter 5 presents two applications developed based on the concepts of spatial predictive queries.
- Chapter 6 discusses the open problems and future directions in research.
- Chapter 7 concludes the thesis and gives a final discussion of the work presented in the thesis.

Chapter 2

Spatial Predictive Queries, State Of The Art

In this chapter, we briefly explore the existing work related to different branches of predictive spatio-temporal queries. We start by reviewing the existing work about query processing and optimization, then summarizing the commonly used prediction functions, indexing data structures, and finally discussing the used techniques to handle uncertainty while evaluating predictive queries.

2.1 Query Processing and Optimization

Existing algorithms for predictive query processing and optimization can be classified according to the supported query type into the following categories:

(1) *range queries*, e.g., [8, 11, 12]. A predictive range query has a query region R and a future time t, and asks about the objects expected to be inside R after time t. A mobility model [8] is used to predict the coming path of each of the underlying objects and employ the prediction results to evaluate predictive range

queries. Most of existing work considers query region as a rectangle, however the Transformed Minkowski Sum supports range queries with circular regions [11]. This is done by determining whether a time parameterized bounding rectangular, as a moving object, intersects a moving circle that represents a range query. The initial rectangle of the object and the velocity of each edge in this rectangle are considered to compute the position and the rectangle after a certain duration of time in the future.

(2) *K-nearest-neighbor queries*, e.g., [11,13,14]. A predictive K-nearest-neighbor query has a location point P, a future time t, and asks about the K objects expected to be closest to P after time t. Two algorithms, RangeSearch and KNNSearchBF, [11] are introduced to traverse spatio-temporal index tree (TPR/TPR*-tree) to find the nodes that intersect with the query circular region for Range and KNN queries, respectively. Sometimes an expiry time interval is attached to a KNN query result [15,16]. Thus, the KNN query answer is presented in the form of ¡*result*, *interval*¿, where the interval indicates the future interval during which the reported answer is valid.

(3) *reverse-nearest-neighbor queries*, e.g., [13, 17]. Unlike the predictive KNN query which finds the objects expected to be the nearest to a given query region, predictive reverse nearest neighbor (RNN) query finds out the objects expected to have the query region as their nearest neighbor. This query is useful in service distribution applications such as ad-hoc networking to assign mobile devices to the nearest communication service point. For example, an algorithm [13] is proposed to evaluate RNN queries during some specific future time duration starting at the query time. This work assumes the objects movements to be in a linear behavior.

(4) *aggregate queries*, i.e., [18]. A predictive aggregate query has a query region R and a future time t, and asks about the number of objects $\mathcal{N}$ predicted to be inside R after time t. A comprehensive technique [18] that employs adaptive

multi-dimensional histogram (AMH), historical synopsis, and stochastic method is used to provide an approximate answer for aggregate spatio-temporal queries for the future, in addition to the past, and the present.

(5) *continuous queries*, e.g., [5, 19–21]. The difference between a *snapshot* predictive query and a *continuous* one is that the later needs to be continuously reevaluated many times through out its life in the system. The rate of reevaluation depends on the time gap t_{gap} between each two consecutive reported answers specified in the received query Q. Accordingly, continuous query Q needs to be stored at the server side until the end of its life. For example, a quadratic-based kNN [20] algorithm is introduced for processing predictive continuous KNN queries, and a differential update technique is used to maintain the query answers. Next, a probabilistic evaluation [2, 5, 22] is considered for processing continuous range queries.

Additional work considers *query selectivity* which plays a complementary role in the area of predictive spatio-temporal query processing and optimization, i.e., [23–26]. Selectivity prediction is defined as the number of objects expected to be retrieved divided by the size of the underlying objects data set. Accurate estimation for the predicted query selectivity is essential in query optimization and evaluation. For example, spatio-temporal histograms [23, 25] are used to predict spatio-temporal query selectivity.

To sum up, although the extensive investigation for the majority of different types of predictive queries, but each of the existing work supports one or two query types and ignore the rest. Consequently, there is still a lack for one general framework that can support all or at least most of the mentioned query types.

2.2 Prediction Functions

In terms of the underlying prediction function, existing algorithms for predictive spatio-temporal query processing can be classified into three categories:

(1) *Linearity-based prediction*, where the underlying prediction function is based on a simple assumption that objects move in a linear function in time along the input velocity and direction. So, query processing techniques in this category, e.g., [13, 14, 16, 26, 27], take into consideration the position of a moving point at a certain time reference, its direction, and the velocity to compute and store the future positions of that object in a TPR-tree-based index [28]. When a predictive query is received, the query processor retrieves the anticipated position in the given time [27]. Part of the related work in this category concerns with the applications of linearity-based prediction models to answer nearest neighbor queries [14] and reverse nearest neighbor queries [13], or to estimate the query selectivity [26].

(2) *Historical-based prediction*, where the predication function uses object historical trajectories to predict the object next trajectory. Then, query processing techniques in this category, e.g., [8, 18, 29–33] are applied to trajectory of location points. Existing work in this category is based on either mobility model [8] or ordered historical routes [29, 30, 33]. The mobility model [8] is used to capture different possible turning patterns at different roads junctions, and the travel speed for each segment in the road network for each single object in the system. Then, the model is used to predict the future trajectory of each object, and based on that they can answer predictive range queries. The main concern of that model is to put more focus on the prediction of the object behavior in junctions based on historical data of objects trajectories. In the ordered historical routes, the stored past trajectories are ordered according to the similarity with the current

time and location of the object and the top route is considered the most possible one [29,30,32–34]. Some of the existing work in this category is employed for predicting the current object trajectory in non-euclidian space [31] such as road-level granularity. For example, a Predictive Location Model (PLM) [31] is proposed to predict locations in location-based services. This model considers the start point as the object current location while the end point could be any of the possible exit points. PLM computes the shortest path trajectory between the current location and each of the exit points, then the trajectory with the highest probability is considered the predicted path. Moreover, a probabilistic prediction function based on Markov models [32] is introduced for short-term route prediction, while Bayes rule is adapted to predict the final destination of a moving object [30, 34, 35].

(3) *Other prediction functions*, where more complicated prediction functions are employed to realize better prediction accuracy. Query processing techniques in this category, e.g., [7,11,12,36] are adjusted based on the outcome of the prediction function. Existing work in this category either exploits a single function [11, 12], or mixes between two or more functions to form a hybrid prediction model [7,36]. As an example for a single function, a Transformed Minkowski Sum [11] is used to answer predictive queries with circular regions, while Recursive Motion Function (RMF) [12] is used to predict a curve that best fits the recent locations of a moving object and accordingly answer range queries. In the hybrid functions category, two methods [7, 36] are combined to evaluate predictive range and nearest neighbor queries in highly dynamic and uncertain environments.

Unfortunately, all the employed prediction functions either: (a) support only short-term prediction in terms of seconds, minutes, or next edge prediction, (b) support long-term prediction but assume a linearity movement of the underlying moving objects which is not a realistic assumption, or (c) based on complex techniques with a significant computation cost that can not scale up for large number

of objects. As a result, there is still a need for prediction models that can support long-term prediction as well as short-term prediction with the ability to scale up to huge query workloads and large number of moving objects.

2.3 Indexing Techniques

A wide variety of data structures are proposed to index spatio-temporal data to support predictive query evaluation and processing. Some of the existing work assumes simple movement pattern for the underlying moving objects while others handles more complex objects behavior. The most popular spatio-temporal indices can be categorized with respect to the base data structure as follows.

(1) *R-tree based*, e.g., [28, 37–40]. Time Parameterized R-tree (TPR-tree) [28] is an extension of R-tree by adding the time parameter which can be used to support querying current and projected future positions of moving objects. The TPR-tree was enhanced to the TPR*-tree [40] by introducing some improved construction algorithms. R^{exp}-tree [38] is proposed as an access method that indexes the current and predicted locations of moving objects assuming that their positions expire after specified time periods. A different technique based on convex hull property [41] is introduced for indexing objects with nonlinear trajectories using a traditional index structure.

(2) *B-tree based*, e.g., [42–44]. An indexing schema called B^x-tree [43] based on B^+-tree, uses a linear technique to index changes in the underlying data values such as moving-objects locations. Based on this B^x-tree index, some algorithms were provided to answer predictive range and KNN queries on near-future positions of the indexed objects. B^{dual}-tree is an enhancement of B-tree by taking into consideration the velocity in addition to the location while indexing the moving objects [44]. Next, a Self-Tunable Spatio-Temporal B^+-tree, termed ST^2B-tree, is

introduced [42] to handle frequent updates for objects locations. This is done by allowing automatic online rebuilding of its subtrees using a different set of reference points and different grid size without significant overhead. The key for an entry in the ST^{2}B-tree index consists of a time part in addition to a space part. The time part is based on dividing the dimension into equivalent partitions, while the space part is based on the Voronoi diagram for dividing the space.

(3) *kd-tree based*, e.g., [45–48]. MOVIES is a main memory indexing technique [45, 46] proposed to handle the high frequent updates while guaranteeing fast response to predictive queries on moving objects data. It is based on the kd-tree data structure [47, 48] and its main idea is to create new indexes for the most updated pieces of the main index and throw them away from the main memory after some short time period.

(4) *Quad-tree based*, e.g., [49]. A dual transformation is used in an indexing method called STRIPES [49] to index the predicted trajectories in a dual transformed space. Trajectories for objects in a d-dimensional space become points in a higher-dimensional, (a 2d-dimensional), space. This dual transformed space is then indexed using a regular hierarchical grid decomposition indexing structure. More detailed review and comparison between the existing techniques for indexing the current and predictive locations of moving objects is covered in [47, 50, 51].

2.4 Uncertainty

Unlike most of existing work in predictive queries that assume the deterministic behavior of objects movements, few research trials assume uncertainty about these movements. Basically, uncertainty deals with stochastic, (probabilistic movement patterns), of the underlying objects with respect to object locations and velocities at different time stamps. The existing few trials in the area of predictive queries

over uncertain moving objects data can be classified according to their aim as follows.

(1) *Indexing* e.g., [36]. To index uncertain motions of a set of moving objects, the B^x-tree is enhanced and two movement inferencing techniques are introduced to obtain anticipated objects locations in non-deterministic format. This work assumes uncertainty for the given past locations and velocities. The adapted B^x-tree and the inference techniques are employed to evaluate predictive range and KNN queries.

(2) *Modeling* e.g., [12, 52]. Unlike the work that is based on a deterministic linear movement, a Recursive Motion Function (RMF) [12] is presented to model uncertain motion patterns in different shapes e.g. polynomial, sinusoid, circle, and ellipse. This work assumes uncertainty about the representation of objects movements patterns. Moreover, a Spatio-Temporal Prediction tree, STP-tree [12], is introduced to index these uncertain movement patterns and to answer predictive queries. Next, a model called PutMode [52] is introduced to predict next trajectory using uncertain data about objects locations. However, none of the predictive spatio-temporal queries are explicitly supported.

Chapter 3

Index Structure For Spatial Predictive Queries on Euclidean Space

This chapter describes the leverage of a grid-based index structure within *Panda* system which is proposed to evaluate spatial predictive queries for objects moving on Euclidean space [9, 53, 54]. We start by briefing the system architecture which includes the main modules and events, explaining the long-term prediction function [30, 34] and our adaptation on it to be employed in the *Panda* framework. Then, we give detailed description for the system components and provide the experimental evaluation.

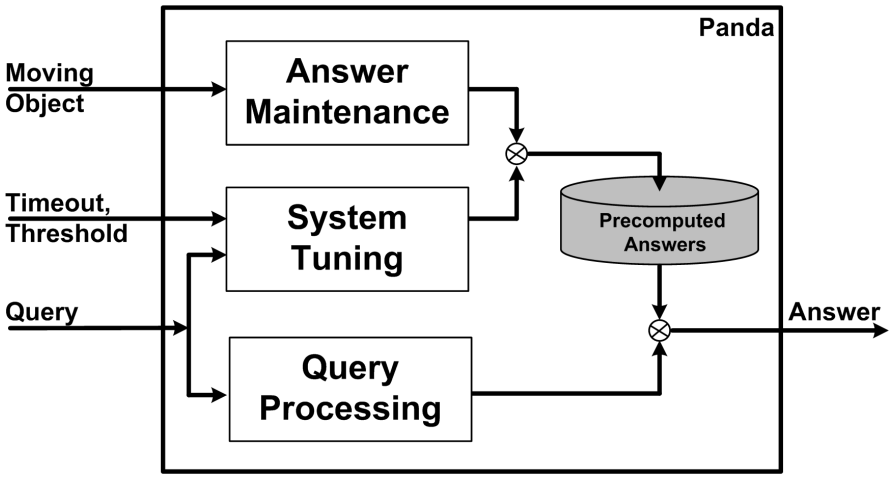


Figure 3.1: The *Panda* System Architecture

3.1 System Architecture

Figure 3.1 gives the system architecture of the *Panda* system, which includes three main modules, namely, *query processing*, *periodic statistics maintenance*, and *answer maintenance*. Each module is dispatched by an event, namely, *query arrival*, *periodic statistics maintenance trigger*, and *object movement*, respectively. As a shared storage, a list of precomputed answers is maintained, which is frequently updated offline and used to construct the final query answer for received predictive queries. Below is a brief overview of the actions taken by *Panda* for each event.

Query arrival. Once a query is received by *Panda*, the query processor divides the query area into two parts. The first part is already precomputed where this part of the answer is just retrieved from the precomputed storage. The second part is not precomputed and needs to be evaluated from scratch through the computation of the prediction function against a candidate set of moving objects.

Object movement. Whenever *Panda* receives an object movement, it dispatches

the answer maintenance module to check if this movement affects any of the precomputed answers. If this is the case, the affected precomputed answers are updated accordingly.

Periodic statistics maintenance trigger. System statistics that decide on which parts of the space to precompute for potential incoming frequent queries need to be updated periodically using the statistics maintenance module. The module basically reset the statistics to ensure the accuracy and recency of collected statistics.

3.2 Prediction Function

The long-term prediction function deployed in *Panda* is mainly an adaptation of the one introduced by Microsoft Researchers [30, 34] to predict the final destination of a single object.

F is applied to any space that is partitioned into a set of grid cells $\mathcal{C}$. It takes two inputs, namely, a cell $C_i \in \mathcal{C}$ and a sequence of cells $O_s = \{C_1, C_2, \cdots, C_k\}$ that represents the current trip of an object O. Then, F returns the probability that C_i will be the final destination of O, Equation 3.1.

$$F \leftarrow P(C_i|O_s) = \frac{P(O_s|C_i)P(C_i)}{\sum_{j=1}^{N} P(O_s|C_j)P(C_j)} \tag{3.1}$$

The term $P(O_s—C_i)$ in the numerator is the possibility of the sequence O_s of the current traversed cells by the object O given the destination cell C_i. This term can be computed using the traveling efficiency parameter P, Equation 3.2 which measures to what extend an object O follows the shortest path in its movements from source to destination.

$$P(O_s|C_i) = \prod_{k=2}^{n} \begin{cases} P & \text{if cell } C_k \text{ in } O_s \text{ is closer to } C_i \text{ than the cell } C_k - 1 \\ 1 - P & \text{otherwise} \end{cases} \tag{3.2}$$

This parameter varies from object to another and it can be obtained by examining the most recent bunch of trajectories of each object. *P(C$_i$)*, the second term in the numerator, is the previous probability of C_i to be a destination for O_s. Initially this term is set to $1/n$, where n is the number of cells in the grid. The denominator is a normalization factor to sum up all probabilities for all cells in the grid to one, given the recent rout of an object.

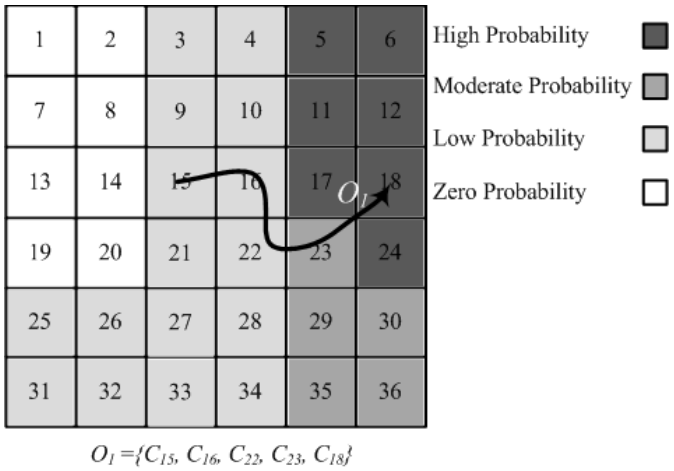


Figure 3.2: Destinations Probabilities Based on Object Sequence

The way the prediction function works is demonstrated in Figure 3.2, where the given space in which the objects move is partitioned into 6 × 6 squared cells numbered from 1 to 36. The current trajectory of the moving object O_1 is drawn as a line started at cell C_{15} and headed to cell C_{18}. The sequence of cells

representing O_1 in its current trip is $S_{O_1} = \{C_{15},\ C_{16},\ C_{22},\ C_{23},\ C_{18}\}$. The color of a cell indicates its probability of being a destination to the object O_1 given its sequence S_{O_1}, the darker the cell color, the higher the probability. As the object moves toward its final trip destination, the prediction function updates its computation. So, some of the grid cells become more probable destination (e.g. C_{24}), and others become less probability, (e.g. C_{31}).

As F only predicts the destination of an object, it does not have the sense of time. In other words, F cannot predict where an object will be after time period t. Since this is a core requirement in *Panda*, we adapt F to be able to compute the probability that object O will be passing by the given cell C_i after time t, where t is specified in the predictive query. The adaptation results in the function $\hat{F}$, Equation 3.3, which is a normalization of the results from the original prediction function F using the set of cells D_t that could be a possible destination of an object O after time t.

$$\hat{F} \leftarrow P(C_i|O_s, t) = \frac{P(C_i|O_s)}{\sum_{d \in D_t} P(C_d|O_s)} \tag{3.3}$$

Here, the numerator is the output of the original prediction function F, and the denominator is the summations of the probabilities of all grid cells in D_t, also computed from F. D_t is the set of possible destinations of object O after time t.

3.3 Panda: A Predictive Spatio-Temporal Query Processing

A salient feature of *Panda* is that it is a generic framework that supports a wide variety of predicative spatio-temporal queries. *Panda*'s query processor can support range queries, aggregate queries, and k-nearest-neighbor queries within the

same framework. In addition, *Panda* can support stationary as well as moving objects. Finally, *Panda* is easily extensible to support continuous queries. This generic feature of *Panda* makes it more appealing to industry and easier to realize in real commercial systems. This is in contrast to all previous work in predictive spatio-temporal queries that focus on only one kind of spatio-temporal queries. As described in Figure 3.1, *Panda* reacts to three main events, namely, *query arrival*, *object movement*, and a *periodic statistics maintenance trigger*. Each event prompts *Panda* to call one of its three main modules to take the appropriate response. The section first starts by describing the underlying data structure of *Panda* (Section 3.3.1). Then, the reaction of *Panda* to the events query arrival, object movement, and periodic statistics maintenance trigger are described in Section 3.3.2, 3.3.3, and 3.3.4, respectively.

3.3.1 Data Structure

Figure 3.3 depicts the underlying data structure used by *Panda*. A brief overview of each data structure is outlined below:

Object List OL. This is a list of all moving objects in the system. For each object $O \in OL$, we keep track of an object identifier and the sequence of cells traversed by O in its current trip. For example, as illustrated in Figure 3.3, O_2 in its current trip, has passed through the sequence of cells $\{C_{13}, C_7, C_2, C_3\}$. This means that O_2 has started at C_{13} and it is currently moving inside C_3.

Space Gird SG. *Panda* partitions the whole space into $N \times N$ equal-size grid cells. For each cell $C_i \in SG$, we maintain four pieces of information as: (1) *CellID* as the cell identifier, (2) *Current Objects* as the list of moving objects currently located inside C_i, presented as pointers to the Object List OL, (3) *Query List* as the list of predictive queries recently issued on C_i. Each query Q in this list is presented by the triple (*Time*, *Counter*, *Answer*), where *Time* is the future time

included in Q, *Counter* is the number of times that Q is recently issued, *Answer* is the precomputed answer for Q which may have different format based on the type of Q, and (4) *Frequent Cells* as the list of cells that one of their precomputed answers should be updated with the movement of an object in C_i.

Travel Time Grid TTG. This is a two-dimensional array of $N^2 \times N^2$ cells where each cell $TTG[i, j]$ has the average travel time between space cells C_i and C_j, where C_i and $C_j \in SG$. TTG is fully pre-loaded into *Panda* and is a read-only data structure.

3.3.2 Query Processing in Panda

The generic query processing of *Panda* does not only predict the query answer, but it also prepares partial results of the incoming queries before hand. In general, *Panda* does not aim to predict the whole query answer, instead, it predicts the answer for certain areas of the space. Then, the overlap between the incoming query and the precomputed areas controls how efficient the query would be. If all the query is precomputed, the query will have best performance in terms of lower latency, however, the *Panda* system will encounter high overhead of maintaining the precomputed answer. This isolation between the precomputed area and the query area presents the main reason behind the generic nature of *Panda* as any type of predictive queries (e.g., range and k-nearest-neighbor) can use the same precomputed areas to serve its own purpose. Another main reason for the isolation between the precomputed areas and queries is to provide a form of *shared execution* environment among various queries. If *Panda* would go for precomputing the answer of incoming queries, there would be significant redundant computations among overlapped query areas.

The *Panda* query processor utilizes its grid structure $\mathcal{G}$ to decide on precomuting the answer for some specific cells of the $\mathcal{G}$. Upon the arrival of a new predictive

spatio-temporal query Q, with an area of interest R, requesting a prediction about future time t, *Panda* first divides Q into a sets of grid cells C_f that overlap with the query region of interest R. For each cell $c \in C_f$, *Panda* goes through two main phases, namely, *result computation* and *statistic maintenance*.

The result computation phase (Section 3.3.2) is responsible on getting the query result from cell c either as a precomputed result or by computing the result from scratch. The *statistic maintenance* phase (Section 3.3.2) is responsible on maintaining a set of statistics that help in deciding whether the answer of cell c, for a future time t, should be precomputed or not.

The precomputation at cell c will significantly help for the next query that asks for prediction on c with the same future time t, yet, precomputation will cause a system overhead in continuously maintaining the answer at c. Throughout this section, Algorithm 1 gives the pseudo code of the *Panda* query processor where the first three lines in the algorithm finds out the set of cells C_f that overlaps with the query region R, and start the iterations over these cells.

Phase I: Result Computation.

Phase I receives: (a) a predictive query Q, either as range, aggregate, or k-nearest-neighbor, asking about future time t, and (b) a cell c_i that overlaps with the query area of interest R. The output of Phase I is the partial answer of Q computed from c_i.

Main idea. The main idea of Phase I is to start by checking if the query answer at the input cell c_i is already computed. If this is the case, then Phase I is immediately concluded by updating the query result Q by the precomputed answer of c_i. If the answer at c_i is not precomputed, then, Phase I will proceed by computing the answer of c_i from scratch. Phase I avoids the trivial way of computing the prediction function of all objects in the system to find which objects

can make it to the query answer at future time t. Instead, Phase I applies a smart *time filter* to limit its search to only those objects that can possibly reach to cell c_i within the future time t. Basically, Phase I utilizes the *Travel Time Grid (TTG)* data structure to find the set of cells C_R that may include objects reachable to c_i within time t. Then, we calculate the prediction function for only those objects that lie within any of the cells in C_R. The result of these prediction functions pile up to build the answer result produced from c_i.

Algorithm. The pseudo code of Phase I is depicted in Lines 4 to 16 in Algorithm 1. Phase I starts by checking if the answer of c_i at time t is already precomputed in its own *Query List* entry in the grid data structure SG. If this is the case, we just retrieve the precomputed answer as the complete cell answer (Line 6 in Algorithm 1), and conclude the phase by using the cell result to update the final query result (Line 16 in Algorithm 1). Updating the result is done through the generic function *UpdateResults* that takes two parameters, the first is the result to be updated, and the second is the value to be used to update the result. The operations inside this functions depend on the underlying query type, e.g., aggregate, range, or k-nearest-neighbor queries. In case that the answer of cell c_i is not precomputed, we start by computing this answer from scratch (Lines 8 to 14 in algorithm 1). To do so, we apply a *time filter* by retrieving only the set of cells C_R that are reachable to c_i within the future time t by checking the *Travel Time Grid (TTG)*. Only those objects that lie within any of the cells in C_R may contribute to the final cell answer, and hence the query answer. For each object O in any of the cells of C_R, we utilize our underlying prediction function (Section 3.2) to calculate the predicted value of having O in c_i within time t (Line 11 in Algorithm 1). We then use this predicted value to update the result of cell c_i using the generic *UpdateResults* function. Once we are done with computing all the predicted values of all objects in any of the cells of C_R, we again utilize

the generic function *UpdateResults* to update the final query result by the result coming from cell c_i (Line 16 in Algorithm 1).

Example. Figure 3.4 gives a running example of Phase I where 19 objects, O_1 to O_{19} are laid on a 6×6 grid structure. Figure 3.4(a) indicates the arrival of a new predictive range query Q_{30}, a shaded rectangle in cell C_{19}, that asks about the set of objects that will be in the area of Q_{30} after 30 minutes. Though we are using a range query as a running example, all idea here are applied to aggregate and k-nearest-neighbor queries. First, we find out all the cells that overlap the area of query Q_{30}. For ease of illustration, we intentionally have Q_{30} covering only one cell, C_{19}, in which we are going to carry on for the next steps. If Q_{30} covers more than one cell, then, the next steps will be repeated for each single cell covered by Q_{30}. Figure 3.4(b) gives the *Query List* structure of C_{19}, where two previous predictive queries came at this cell before; a query that asks about 30 minutes in future, and it came only one time before ($counter = 1$) and another query that asks about 20 minutes in the future and were issued 10 times before. By looking at this data structure, we find that the answer of the future time t is set to *null*, i.e., it is not precomputed. In this case, we need to compute the answer for this cell from scratch. Note that if this query was asking about the set of objects after 20 minutes, we would just report the answer as $\{O_1, O_8\}$ as it is already precomputed. Unfortunately, for the case of $t = 30$, we need to proceed for more computations.

Figure 3.4(c) starts the process of computing the answer of cell C_{19}. As a first step, we utilize the *Travel Time Grid (TTG)* data structure to find out the set of cells that are reachable to C_{19} within 30 minutes. We find that there are only three cells that can contribute to the answer of C_{19}, namely, C_9, C_{16}, C_{33}. This means that objects that are not located in any of these cells are not going to make any contribution to C_{19} within 30 minutes, which filters out large number of

moving objects that. Then, we can only focus on the objects located in C_9, C_{16}, C_{33}, where there are only four objects O_5, O_9, O_{18}, and O_{19}. For each of these four objects, we calculate the prediction function $\hat{F}$ to find out the probability that these objects can be in C_{19} in 30 minutes. With probability calculation, we find out that O_{19} has a zero probability of being in C_{19} in 30 minutes, while the other three objects have a non-zero probability. We finally report the answer in Figure 3.4(d) as $\{O_5, O_9\, O_{18}\}$ along with their probabilities of being in C_{19} in 30 minutes (Probabilities are not shown in the figure as it is an illustrative example).

Phase II: Statistics Maintenance

Phase II does not add anything to the query answer. Instead, it updates a set of statistics that help in deciding what parts of the space and queries need to be precomputed. The input to this phase is a cell c_i and its answer list, computed in Phase I. Then, Phase II uses this information to update the statistics maintained by *Panda*.

Main idea. The main idea of Phase II is to employ a tunable threshold, $0 \leq \mathcal{T} \leq \infty$, that provides a trade-off between the predictive query response time and the overhead for precomputing the answer of selected areas. At one extreme, $\mathcal{T}$ is set to 0, which means that all queries will be precomputed beforehand. Though this will provide a minimal response time for any incoming query, a significant system overhead will be consumed for the precomputation and materialization of the answer. On the other extreme, $\mathcal{T}$ is set to ∞, which means that nothing will be precomputed, and all incoming queries need to be computed from scratch. This will provide a minimum system overhead, yet, an incoming predictive query will suffer from high latency. To efficiently utilize the tunable threshold $\mathcal{T}$, Phase II utilizes the *counter* information in the *Query List* data structure of cell c_i (described in Section 3.3.1). If this *counter* exceeds the threshold value $\mathcal{T}$,

then, this query is considered frequent, and the answer of this query in cell c_i is precomputed, i.e., stored in the *Query List* data structure. In addition, we add cell c_i to the list of frequent cells in all cells that are reachable to c_i within time t. This is mainly to say that any object movement in any of these reachable cells will affect the result computed (and maintained) at cell c_i. Such list of reachable cells can be directly obtained from the *Travel Time Grid (TTG)* data structure.

Algorithm. The pseudo code of Phase II is depicted in Lines 18 to 27 in Algorithm 1. Phase II starts by retrieving the entry e from the *Query List* of c_i that corresponds to the querying time t. If there is no such prior entry, i.e., e is *NULL*, we just add a new blank entry in the *Query List* of c_i for time t, with *counter* set to zero and *answer* set to null (Lines 18 to 21 in algorithm 1). Then, we just increase the *counter* of e by one to update the number of times that this query is issued at cell c_i with time t. Then, we check the *counter* against the system threshold T and the value of the current cell *Answer*. This check may result in three different cases as follows: (1) $e.counter < \mathcal{T}$, i.e., the *counter* is less than the system threshold $\mathcal{T}$. In this case, Phase II decides that it is not important to precompute the result of this query, as it is not considered as a frequent query yet. So, Phase II is just concluded. (2) $e.Answer \neq$ NULL. In this case, the query time t is already considered frequent and the answer is already precomputed. In this case, Phase II will also just conclude as there is no change in status here. (3) $e.counter \geq \mathcal{T}$ AND $e.Answer$ is NULL. This case means that the query time t has just become a frequent one, and we need to start precomputing the result for t at cell c_i. In this case, we first add the computed cell result from Phase I to the answer of e. Then, we find out the set of cells C_R that are reachable to cell c_i within time t. For these cells, we add cell c_i to their list of frequent cells. This is mainly to say that any object movement of any cell $c_j \in C_R$ will affect the result computed at cell c_i (Lines 18 to 23 in algorithm 1).

Example. Figure 3.5 gives a running example of Phase II continuing the computations of Phase I in Figure 3.4. Figure 3.5(a) shows that the *counter* of the time entry 30 is updated to be 2. Assuming the time threshold $\mathcal{T}$ is set to 2. Then, the time t is now considered frequent. Figure 3.5(b) depicts the actions taken by Phase II upon the consideration that Q_{30} becomes frequent. First, the *Query List* of C_{19} is updated to hold the computed answer from Phase I. Second, the cell C_{19} is added to the list of frequent cells for C_9, C_{16}, and C_{33} to indicate that any movement in these three cells may trigger a change of answer for cell C_{19}.

3.3.3 Object Movement in Panda

As has been discussed in the previous section, the efficiency of the *Panda* generic query processor relies mainly on how much of the query answer is precomputed. Though we have discussed how *Panda* takes advantage of the precomputed answers, we did not discuss how *Panda* maintains those precomputed answers, given the underlying dynamic environment of moving objects. This section discusses the *answer maintenance* module in *Panda*, depicted in Figure 3.1, which is basically triggered with every single object movement in any cell c_i in the space grid SG.

Main idea. The main idea behind the *answer maintenance* module is to check if this object movement has any effect on any of the precomputed answers. If this is the case, then *Panda* computes this effect and propagates it to all affected precomputed answers. If *Panda* figures out that this object movement has no effect on any of the precomputed answers, then, it does nothing for this object movement. As the underlying prediction function $\hat{F}$ mainly relies on the sequence of prior visited cells for a moving object, an object movement within the cell does not change the object predication function, and hence will not have any effect on any of the precomputed answers. It is important to note that the *answer*

maintenance module does not decide upon which parts of the queries/space to be precomputed, as this decision is already taken by the statistics collected in the generic query processor module. Instead, the *answer maintenance* module just ensures efficient and accurate maintenance of existing precomputed answers.

Algorithm. Algorithm 2 gives the pseudo code of the *Panda answer maintenance* module. The algorithm takes three input parameters, the moved object O, its old cell C_{old} before movement, and its new cell after movement C_{new}. The first thing we do is to check if the new cell is the same as the old cell. If this is the case, the algorithm immediately terminates as this object movement will not have any effect on any of the precomputed cells. On the other side, if the new cell is different form the old one, the algorithm proceeds in two parts. In the first part (Lines 4 to 10 in Algorithm 2), we first add O to the set of current objects of C_{new}. Then, we retrieve the set of *frequent cells* of C_{new}, i.e., those cells that have precomputed answers and may be affected by any change of objects in C_{new}. For each cell C_i in the set of *frequent cells*, we do: (a) retrieve the travel time t from the new cell to C_i from the *Travel Time Grid* data structure, (b) compute the predicted value of O being in C_i after t time units, and (c) update the precomputed result at cell C_i by the predicated value, using the generic function *Update Results*. The second part of the algorithm (Lines 11 to 15 in Algorithm 2) is very similar to the first part, except that we are working with C_{old} instead of C_{new}, where we remove O form the set of objects of C_{old}, and update all the precomputed frequent cells of C_{old} accordingly. It is important to notice here that we do not need to compute the object prediction in the second part as it is already calculated before and stored in the precomputed answer at C_i.

Example. Back to our running example in Figure 3.5(b) that illustrates the precomputed answer for the query Q_{30} in cell C_{19}. Assume that object O_9 moves out from its cell C_{16} to C_{17}. In this case, we add O_9 to the list of *current objects*

in C_{17}, and get its list of *frequent cells*, which only includes C_1. Then, we obtain the time t between C_1 and C_{17} as 40. We then compute $\hat{F} = P(C_1\text{—}O_9, 40)$, which gives the probability that O_9 will be in C_1 after 40 time units. We then incrementally update the answer at C_1 by the value of $\hat{F}$. We do the same for C_{16}, the cell that O_9 has just departed. We first delete O_9 from the list of *current objects* in C_{16}. Then, we read the list of *frequent cells* of C_{16} which returns only C_{19}. At this point, we do not need to compute the prediction function $\hat{F}$ as it should be already stored in the *query list* of C_{19}. So, we just update the answer at C_19 by removing O_9 and its probability.

3.3.4 Periodic Statistics Maintenance

As has been seen earlier, the *query processing* module (Section 3.3.2) mainly relies on simple maintained statistics, namely, the *counter* in the *Query List* data structure, to decide on which parts of the space to precompute for which future query time. However, the *counter* information just keeps increasing while frequent queries may no longer be frequent any more, yet, as they still keep their *counter* value intact, their answers may still be unnecessarily precomputed, causing extra system overhead. It is the job of the *periodic statistics maintenance* module, discussed in this section, to ensure that current statistics information is accurate and updated.

Main Idea. The main idea behind this module is to run periodically each t units to sweep over current statistics and update it. For a query Q to be considered frequent, it has to appear at least $\mathcal{T}$ times in the last time period t, where $\mathcal{T}$ is the system threshold, described in Section 3.3.2. In the mean time, a frequent query Q, who failed to appear at least $\mathcal{T}$ times in the last time period t is demoted to be infrequent.

Algorithm. Algorithm 3 gives the pseudo code for the *periodic statistics*

maintenance module. The algorithm sweeps over all grid cells in the grid data structure SG. For each cell C_i, the algorithm goes through every single entry e in $C_i.QueryList$. For each entry e, we compare its *counter* against the system threshold $\mathcal{T}$, which will result in one of these two cases: (1) $e.Counter \geq \mathcal{T}$. In this case, e represents a frequent query, and thus we just reset its counter to 0 to restart its statistics with the next time period t. (2) $e.Counter < \mathcal{T}$, in which e represents a query that failed to appear more than $\mathcal{T}$ times in the last time period t. In this case, we remove e from the list $C_i.QueryList$ while doing a clean up by removing the entry for C_i from the list of frequent cells in each of its reachable cells within $e.Time$.

Example. After running *Panda* for one hour, we had the precomputed answer in the cell C_{19} in our example in Figure 3.5(b). Assuming the used threshold $\mathcal{T}$ value is three, then we check the entries in the query list in the cell C_{19}. The *counter* value in the first record (where $time = 30$) is less than 3 and its *answer* is $\neq$ NULL, then we set this precomputed answer to NULL and remove C_{19} from the list of *frequent cells* of of its reachable cells, ($\{C_9$, C_{16}, and $C_{33}\}$), then the *counter* is set to zero. For the second record (where $time = 20$), the *counter* is greater than $\mathcal{T}$ and the *answer* has a non NULL value, therefore, we keep everything as it is except the *counter* which will reset to zero too. For the next timeout, the query list of C_{19} will not contain the record for $t = 30$ since it is not frequent.

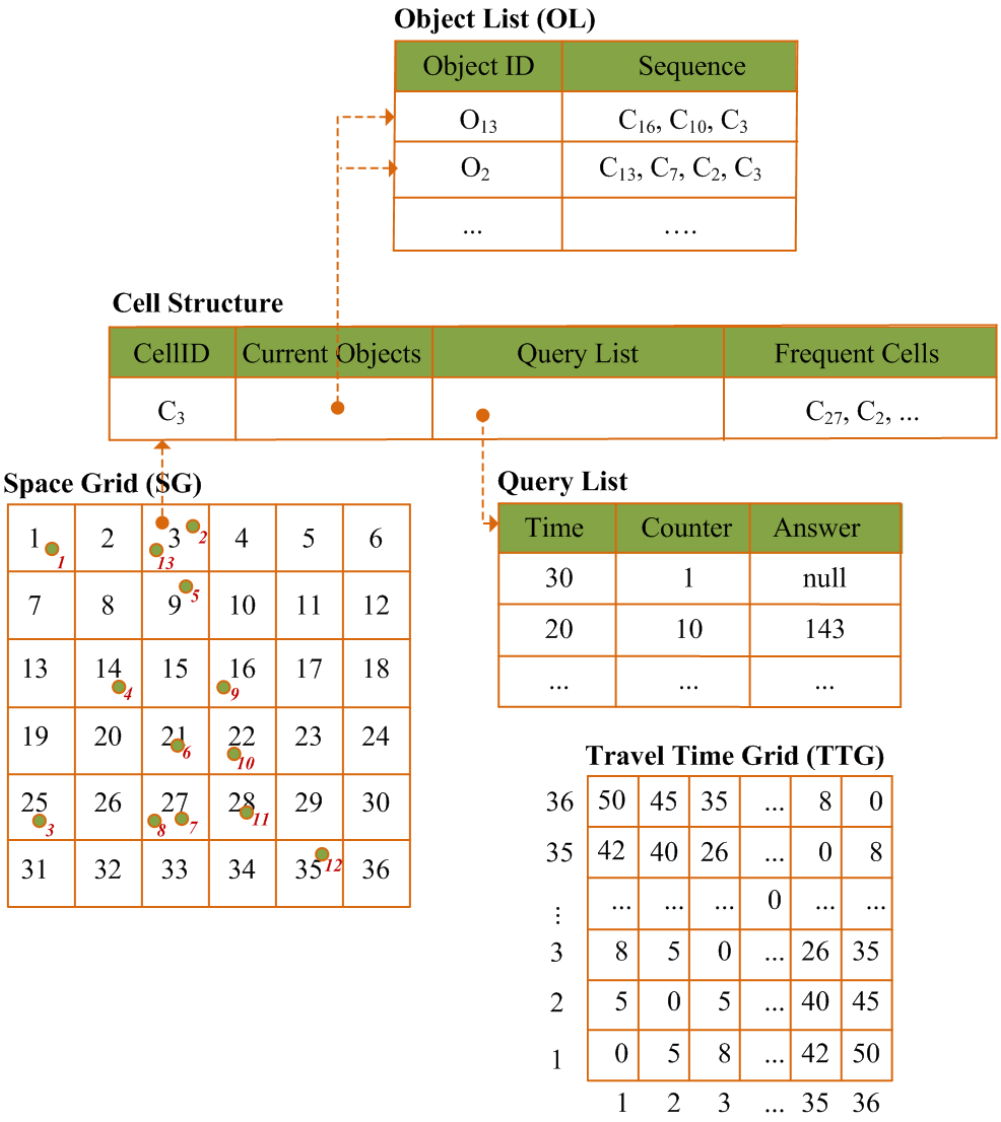
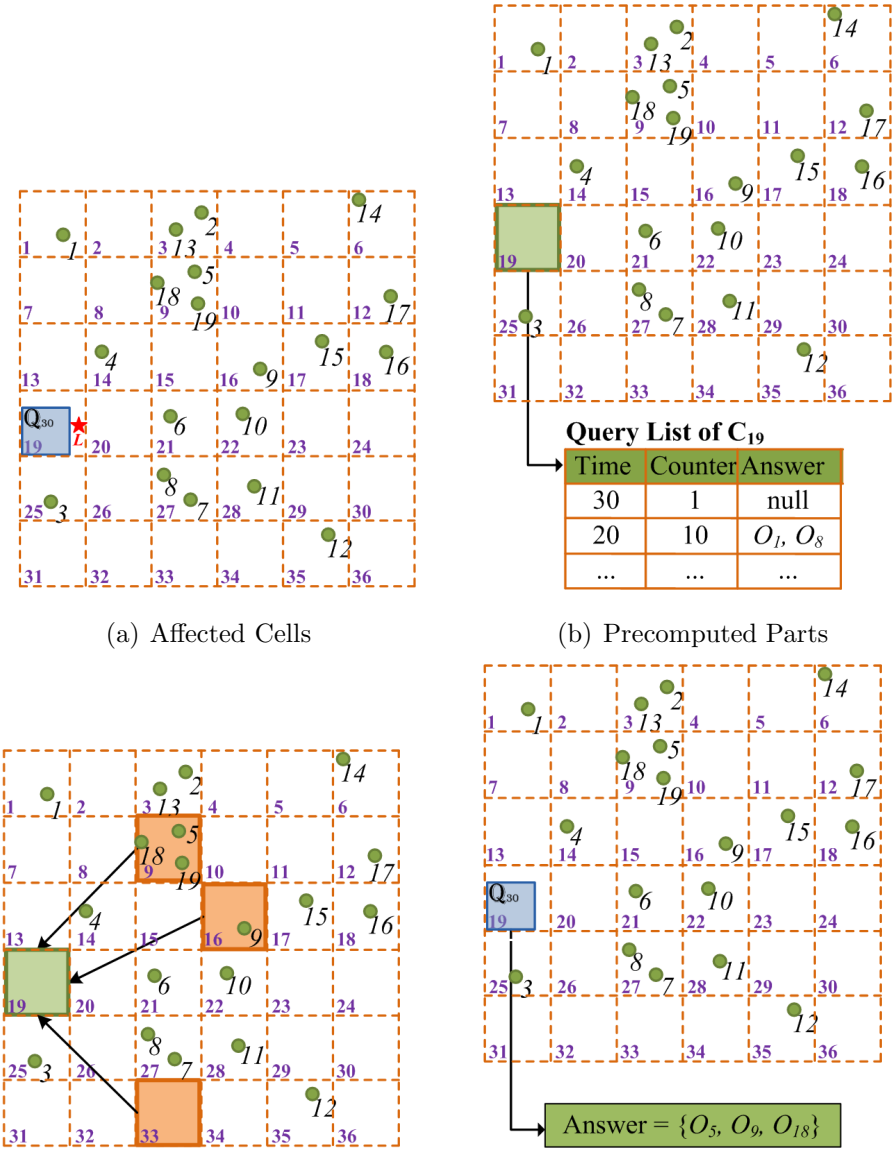


Figure 3.3: Data Structures in *Panda*

Algorithm 1 *Panda* Predictive Query Processor

Input: Region R, time t, Threshold $\mathcal{T}$

```
1: QueryResult ← null, CellResult ← null
2: C_f ← the set of grid cells intersecting with (R)
3: for each cell c_i ∈ C_f do
4:    /* Phase I: Result Computation */
5:    if there is an answer in c_i at time t then
6:       CellResult ← read answer from c_i
7:    else
8:       C_R ← the set of grid cells reachable to c_i in time t
9:       for each cell c_j ∈ C_R do
10:         for each object O ∈ current objects in c_j do
11:            ObjectPrediction ← Compute F̂ = P(c_i—O, t)
12:            UpdateResults (CellResult, ObjectPrediction)
13:         end for
14:      end for
15:   end if
16:   UpdateResults (QueryResult, CellResult)
17:   /* Phase II: Statistics Maintenance */
18:   e ← the entry in the query list of c_i at time t
19:   if e is NULL then
20:      e ← Insert a new blank entry e to the query list of c_i with e.Counter=0 and e.Answer
         is Null
21:   end if
22:   e.Counter ← e.Counter + 1
23:   if e.Counter ≥ T AND e.Answer is NULL then
24:      e.Answer ← CellResult
25:      C_R ← the set of grid cells reachable to c_i in time t
26:      Add c_i to the list of frequent cells in all cells in C_R
27:   end if
28: end for
29: Return QueryResult
```



(a) Affected Cells

(b) Precomputed Parts

(c) Travel Time Filter

(d) Result Formulation

Figure 3.4: Phase I Example.

Algorithm 2 Answer Maintenance

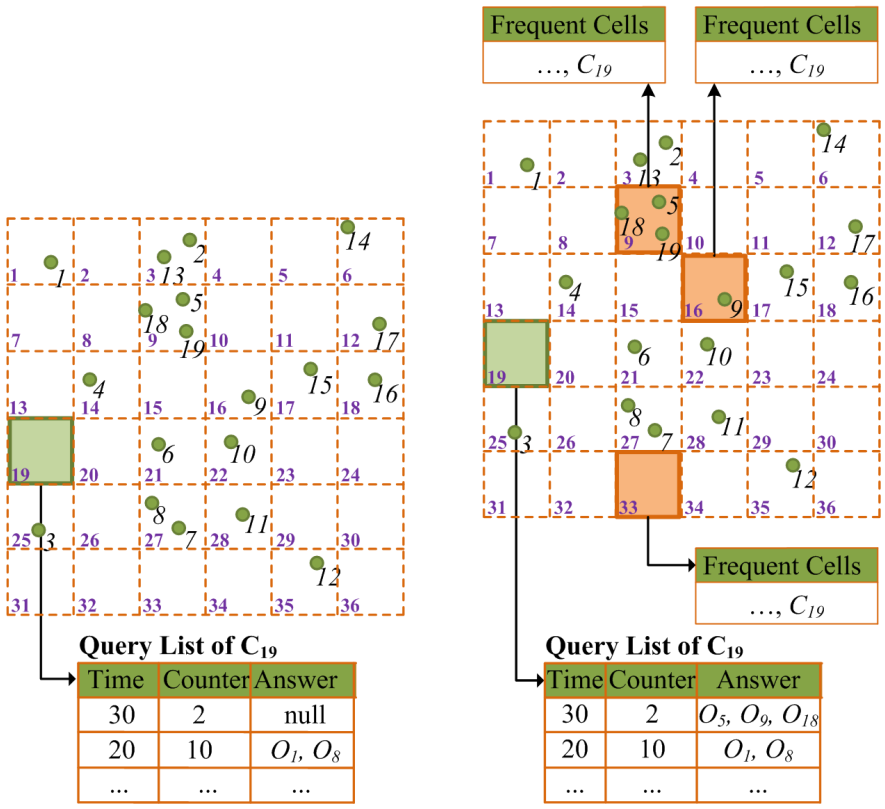
Input: *Object* O, *Cell* C_{old}, , *Cell* C_{new}

if $C_{old} = C_{new}$ **then**
  Return
end if
Add O to the set of *current objects* of C_{new}
$\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{new}
for each cell $C_i \in \mathcal{C}$ **do**
  $t \leftarrow$ travel time from C_{new} to C_i from $TTG[new, i]$
  ObjectPrediction $\leftarrow$ Compute $\hat{F} = P(C_{new}—O, t)$
  UpdateResults (CellResult, ObjectPrediction)
end for
Remove O from the set of *current objects* of C_{old}
$\mathcal{C} \leftarrow$ The set of *frequent cells* of C_{old}
for each cell $C_i \in \mathcal{C}$ **do**
  UpdateResults (CellResult, O**)**
end for
Return

Algorithm 3 Periodic Statistics Maintenance

Input: System Threshold $\mathcal{T}$

for each cell $C_i \in$ the Space Grid SG **do**
  for each entry $e \in C_i.QueryList$ **do**
    if $e.Counter \geq \mathcal{T}$ **then**
      $e.Counter \leftarrow 0$
    else
      Remove C_i from the list of frequent cells in each of its reachable cells within $e.Time$
      Delete e from $C_i.QueryList$
    end if
  end for
end for
Return;

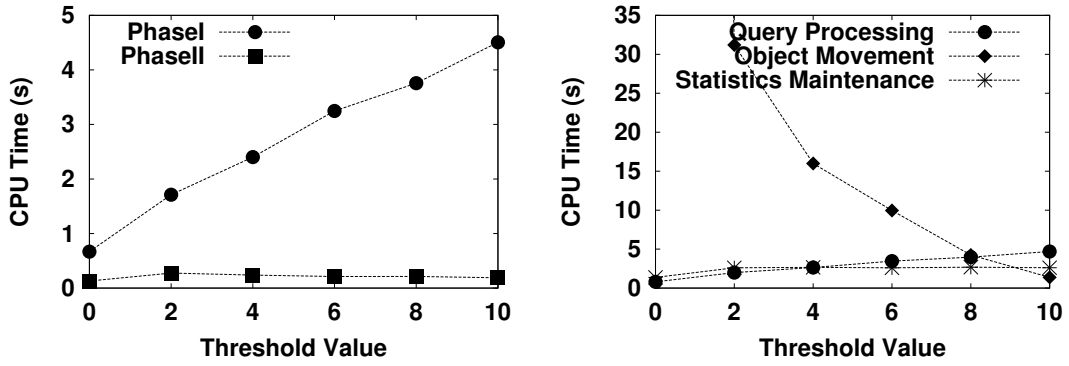


(a) Statistics Update (b) Reachable Cells

Figure 3.5: Phase II Example.

3.4 EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency and the scalability of our proposed system *Panda* for processing predictive queries. We start by explaining the environment of the conducted experiments in Section 3.4.1. Then, we study the impact of threshold tuning on the performance of the main modules of *Panda* in Section 3.4.2. Section 3.4.3 provides the effect of different timeouts on *Panda*



(a) Query Processing Phases (b) Main modules

Figure 3.6: Effect of Threshold Tuning

efficiency. The behavior of *Panda* with different query workloads is given in Section 3.4.4. Finally, we examine the scalability of *Panda* with respect to large number of objects in Sections 3.4.5.

3.4.1 Experiment Setup

In our performance evaluation experiments, we use the Network-based Generator of Moving Objects [55] to generate large sets of synthetic data of moving objects. The input to the generator includes a real road network map for Hennepin County, Minnesota, USA. The output of the generator includes different sets of moving objects that move on the given road network map. The used data sets require some data preprocessing to partition the space in which objects move is into $N \times N$ squared grid cells of width relative to the minimum and the maximum step taken by any of the underlying moving objects. Our *space grid* data structure mirrors the space partitions by storing an identifier for each cell C_i and updatable list of objects moving within that cell C_i. To have the travel time grid TTG filled before starting the experiment, the travel time between any pair of cells, C_i and

C_j, is obtained by taking the average time it takes from the underlying set of objects to move from C_i to C_j.

To have the algorithms tested against different workload rather than single queries, a query workload generator is built to obtain workloads of predictive queries that vary in the number of queries, the query region size, and the query future time. The number of queries in the generated workloads starts at 10K queries per batch, and increases by 10K until reaches 100K queries in a query workload. The generated query regions are squares and their locations are uniformly distributed over the space. The size of the generated queries vary from 0.01 to 0.16 of the total space size. The future times for the generated queries vary from 10 to 80 time unites. A 3k query workload is used for warming up the system before we start to measure the experimental results.

All experiments are based on an actual implementation of *Panda*. All the behaviors of the generated objects, query workload generator, and query processing algorithms are implemented on a Core(TM) i3 4GB RAM PC running Windows 7 with C++. In all experiments, the evaluation and comparison are in terms of CPU time cost.

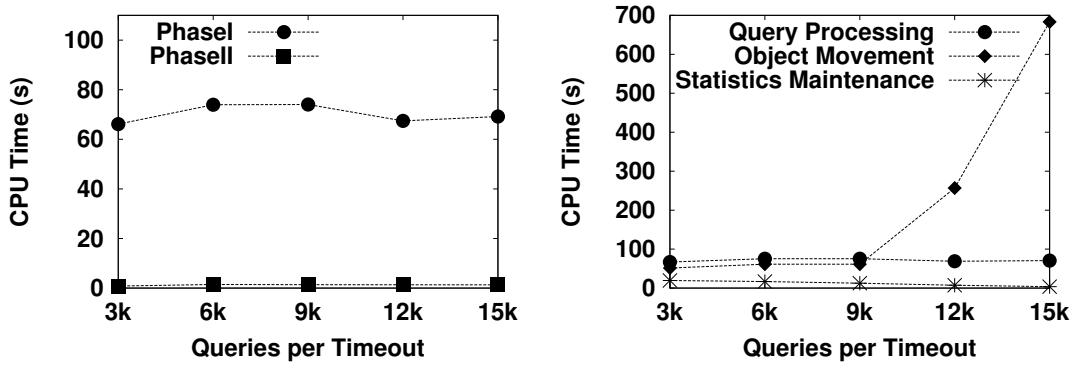
3.4.2 Impact of Threshold Tuning

In the first set of experiments, we study the impact of different threshold $\mathcal{T}$ values on the efficiency of *Panda*. The minimum value that the $\mathcal{T}$ can take in this experiment is zero, which means that the results for any possible query will be precomputed in advance. Accordingly, once a predictive query is received, the answer is read and returned to the user without any further computation. It is obvious here that at this threshold value, we will have the fastest response time, since no computation happens after receiving a query. Just the precomputed result is accessed and returned directly as a final query answer. However, it is

expected to have the most significant overhead for updating those precomputed answers. The maximum threshold value is ten which decided based on the number of queries in a timeout divided by the number of cells in our space grid.

Figure 3.6 illustrates the effect of choosing different threshold values on the performance of *Panda*. In this experiment we run 5k queries on 10K moving objects at $\mathcal{T}$ = 0, 2, 4, 5, 6, 8, 10. In Figure 3.6(a), we study the influence of threshold tuning on the two phases of the *query processing* module. As the value of $\mathcal{T}$ increases, the cost of *Phase I* increases. The justification is that when $\mathcal{T}$ has large value, this means that most of the answers in the cells intersecting with the query region need to be computed from scratch. With large threshold value, only queries with frequency rate above that value are precomputed. *Phase II* is not sensitive to threshold tuning. The reason is that *Phase II* concerns mainly with updating the statistics inside the cells affected by received queries, and those cells are only sensitive to the size of the query region rather than the threshold value.

Figure 3.6(b) depicts the sensitivity of the main modules in *Panda* to different threshold values. The cost of *query processing* module increases when $\mathcal{T}$ increases. We justified that in the previous sub-figure. The *periodic statistics maintenance* module, *statistics maintenance* for short, is not sensitive to $\mathcal{T}$ at all. The third module, *object movement* is significantly affected by threshold value, as with small values, more answers are precomputed, hence more updates are triggered with each single object movement, and vise versa. In a nutshell, between the minimum $\mathcal{T}$ and the maximum $\mathcal{T}$, the threshold value can be tuned to provide the required balance between the time a user has to wait to receive a query result and the overhead cost used to prepare this answer in advance.

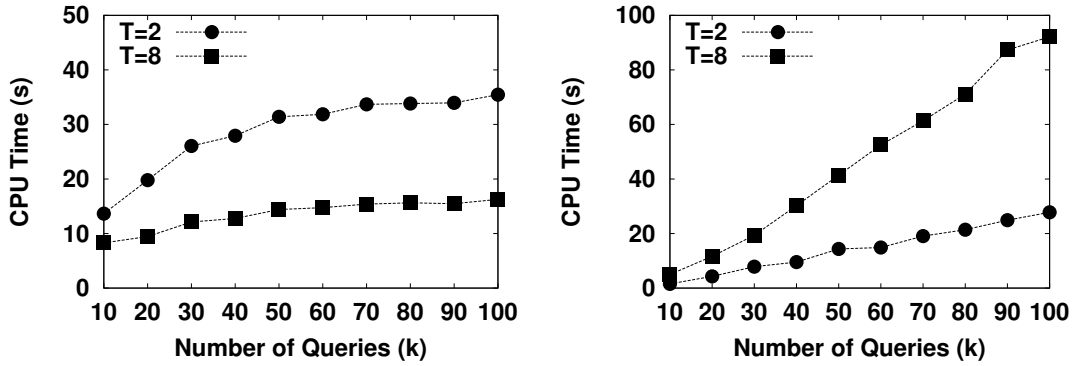


(a) Query Processing Phases (b) Main modules

Figure 3.7: Effect of Timeout Value

3.4.3 Impact of Timeout

In this set of experiments, we study the effect of timeout values on the efficiency of *Panda*. We run a workload of 30K queries on 20K objects with threshold $\mathcal{T} = 8$, while varying the timeout values from 3k queries every timeout to 15K queries per timeout. In Figure 3.7, the x-axis represents the number of queries per timeout and the y-axis represents the total CPU cost in seconds for processing the given workload at that timeout value. Figure 3.7(a) shows that there is a slim effect of selecting different timeouts on the two phases of the *query processing* module. At larger timeouts, 12K and 15K, there is a higher opportunity for queries counters to exceed the given threshold. However, it takes most of the experiment time before the counter passing that threshold. So the effect of timeout is not significant on the two phases of this module. In all cases, more portions of the queries answers are expected to be precomputed, which in turn starts to decrease the cost of computing answers from scratch at timeout 12K in *phaseI*. Figure 3.7(b) assures that timeout values have different impact on the three main modules of the *Panda* system. With smaller timeouts, the *periodic statistics maintenance*



(a) Update Cost (b) Response Time

Figure 3.8: Efficiency Evaluation with Different Workloads

module is dispatched more times which gives it higher cost than with bigger timeouts. The curve of the statistics maintenance cost has a decreasing trend from 19 seconds at timeout = 3k to three seconds at timeout = 15k. As depicted in the previous figure, there is a slim effect of timeout tuning on the total cost of the *query processing* module. With an opposite impact, the cost of the *object movement* module sharply increases with higher timeouts as a result of more queries parts are being precomputed which means each single object movement triggers a possible update.

3.4.4 Efficiency Evaluation

To evaluate the efficiency of *Panda*, we processed workloads of predictive queries while varying the number of received queries from 10K to 100K. We compare the response time and update cost of *Panda* at two different threshold values, two and eight respectively. The response time is the time a user has to wait to get the query result, and it is equal to *Phase I* of the *query processing* module. The update cost is the time consumed to update the precomputed answer when a

change happens, and it is equivalent to the cost of the *object movement* module. In this experiment, we use *timeout* equals to the time required to process 3K queries which means after every 3k queries, we call the *statistics maintenance* module to adjust *Panda* decision about which parts to precompute. As we mentioned in the setup, we use a warmup workload with 3k queries before we write down the experimental results. The used data file contains 20k of moving objects.

Figures 3.8 provides a comparison between *Panda* response time and update cost with different queries workloads at the two selected threshold values. The horizontal axis represents the number of queries in a workload file, and the vertical axis measures the total CPU cost in seconds. Figure 3.8(a) studies the update cost at different workloads. As expected, the update costs at $\mathcal{T} = 2$ are much bigger than the ones at $\mathcal{T} = 8$ in all workloads, while the behavior of the response time, Figure 3.8(b), is the opposite. It is also remarkable that unlike the trend of the update cost which has a slightly increasing curve, the response time with $\mathcal{T} = 8$ sharply increases while the number of processed queries increases. To sum up, based on the expected workload, the required response time, and the affordable update cost, *Panda* can be easily adjusted to fulfill these requirements.

3.4.5 Scalability with Number of Objects

In our second set of scalability experiments, Figure 3.9 depicts the behavior of the main components of *Panda* when the number of moving objects increases from 5K to 80K. We run these experiments with $\mathcal{T} = 5$, timeout = 3K, and 20K queries. As seen in this figure, the *object movement* is the most affected module as there is a positive relationship between the number of objects and update overhead cost results from their movements. It is also observed that there is almost no impact of increasing the underlying number of objects on both the *statistics maintenance* and the *query processing* modules. As noticed from the total cost when the number

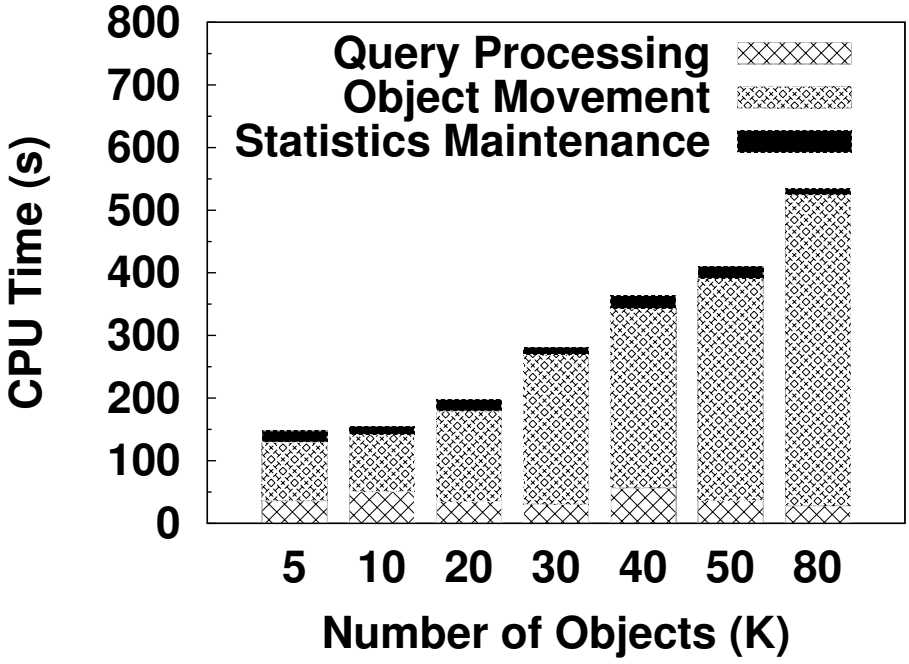


Figure 3.9: Scalability With Objects Number

of objects increases by 16 times, from 5K to 80K, the total cost increases only by less than four times, i.e., from 147.77 to 534.43 seconds. This shows that *Panda* can scale up with large number of moving objects without scarifying neither the response time nor the overall performance.

Chapter 4

P-Tree: Index For Spatial Predictive Queries on Road Networks

This chapter describes a novel index structure, named *Predictive tree (P-tree)*, proposed for processing predictive queries against moving objects on road networks [10, 53, 56]. The *predictive tree*: (1) provides a generic infrastructure for answering the common types of predictive queries including predictive *point, range, KNN, and aggregate* queries, (2) updates the probabilistic prediction of the object's future locations dynamically and incrementally as the object moves around on the road network, and (3) provides an extensible mechanism to customize the probability assignments of the object's expected future locations, with the help of user defined functions. The proposed index enables the evaluation of predictive queries in the absence of the objects' historical trajectories. Based solely on the connectivity of the road network graph and assuming that the object follows the shortest route to destination, the *predictive tree* determines the reachable nodes

of a moving object within a specified time window $\mathcal{T}$ in the future. The *predictive tree* prunes the space around each moving object in order to reduce computation, and increase system efficiency. Tunable threshold parameters control the behavior of the *predictive trees* by trading the maximum prediction time and the details of the reported results on one side for the computation and memory overheads on the other side. The predictive tree is integrated in the context of the *iRoad* system in two different query processing modes, namely, the *precomputed query result* mode, and the *on-demand query result* mode. Extensive experimental results based on large scale real and synthetic datasets confirm that the *predictive tree* achieves better accuracy compared to the existing related work, and scales up to support a large number of moving objects and heavy predictive query workloads.

4.1 Preliminaries

In this section, we formalize the basic predictive query we address in this chapter. Then, we define different types of predictive queries that the *predictive tree* can support within the *iRoad* framework. After that, we explain the intuition of the leveraged prediction model.

4.1.1 Basic Query

In this problem, we focus on addressing the *predictive point* query as our basic query on the road network. In this query, we want to find out the moving objects with their corresponding probabilities that are expected to be around a specified query node in the road network within a future time period. The example of such query could be like, *"Find out all the cars that may pass by my location in the next 10 mins"*. The *predictive point* query we address in this chapter can be formalized as: "Given (1) a set of moving objects $\mathcal{O}$, (2) a road network graph $G(N, E, W)$,

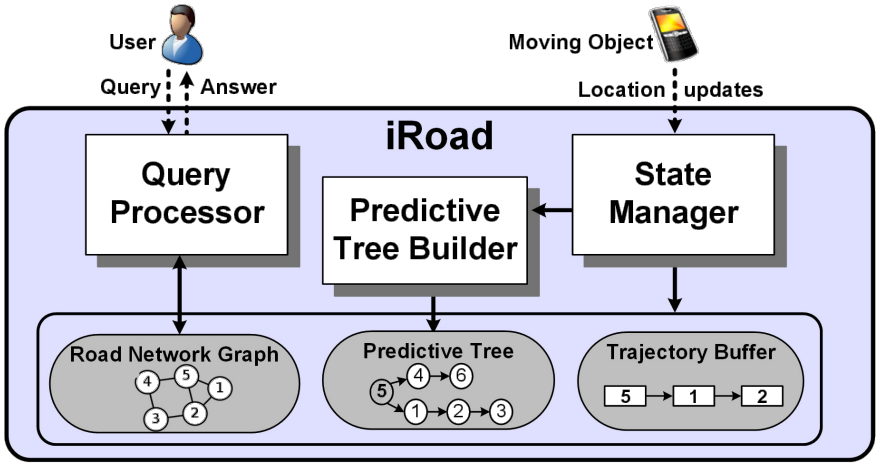


Figure 4.1: iRoad System Architecture

where N is the set of nodes, E is the set of edges, and W is the edge weights, i.e., travel times, and (3) a predictive point query $Q(n,\ t)$, where $n \in N$, and t is a future time period, we aim to find the set of objects $R \in \mathcal{O}$ expected to show up around the node n within the future time t. The returned result should identify the objects along with their probabilities to show up at the node of interest. For example, within the next 30 mins, object o_1 is expected to be at node n_3 with probability 0.8, $R(Q(n_3,30)) = \{< o_1,0.8>\}$.

4.1.2 Extensions

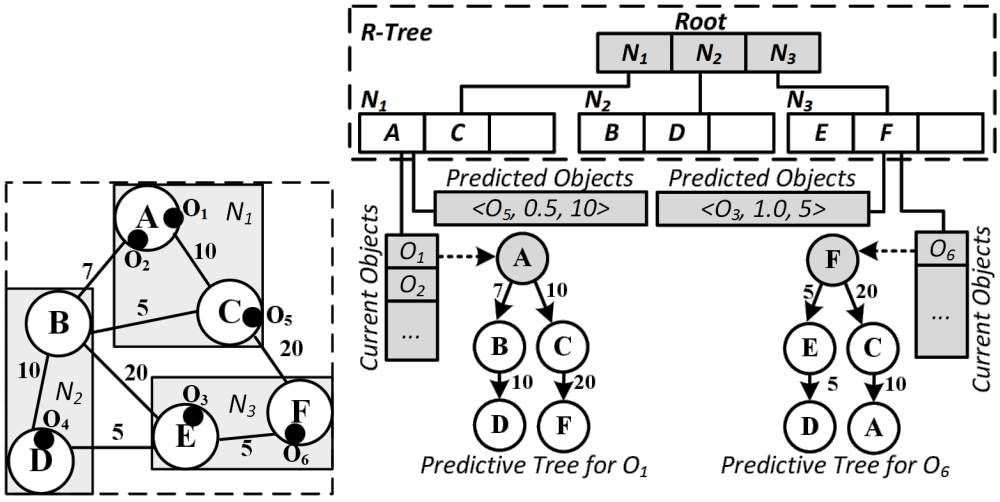
We consider the aforementioned *predictive point* query as a building block upon which our framework can be extended to support other types of predictive queries including: (i) *Predictive range* query, where a user defines a query region that might contain more than one node and asks for the list of objects expected to be inside the boundaries of that region within a specified future time, (ii) *Predictive*

KNN query to find out the most likely K objects expected to be around the node of interest within a certain time period, and (iii) *Predictive aggregate* query to return the number of objects predicted to be within a given location in the next specified time duration.

4.1.3 Prediction Model

Our prediction model employed by the introduced *predictive tree* index structure is based on two corner stones. (1) The assumption that objects follow the shortest paths in their routing trips. The intuition behind this assumption is based on the fact that in most cases, the moving objects on road networks, e.g., vehicles, travel through shortest routes to their destinations [34, 57]. In fact, this assumption is aligned with the observation in [55] that moving objects do not use random paths when traveling through the road network, rather they follow optimized ones, e.g., fastest route. As a result, this model prevents the looping case that appears in the traditional turn-by-turn probability model and assigns a probability value for the moving object to turn when facing an intersection [8].

(2)The probability assignment model that assigns a probability value to each node inside the object's *predictive tree*. In fact, the probability assignment is affected by the node's position with respect to the root of the tree, the travel time cost between the object in its current location to this node, and the number of the sibling nodes. In general, our *predictive tree* is designed to work with different probability assignment models. For example, a possible probability model can give higher values to nodes in business areas, e.g., down town, rather than those in the suburbs. In our default probability model, each node in the *predictive tree* has a value equal to one divided by the number of nodes accessible from the root within a certain time range.



(a) Network & Objects (b) Predictive Trees Integrated With R-Tree

Figure 4.2: Example Of The Proposed Index Structure

4.2 The iRoad System

The proposed *predictive tree* is implemented in the context of the *iRoad* System. More precisely, the *predictive tree* and its construction, maintenance and querying algorithms form the core of the *iRoad* System. The *iRoad* System is a scalable framework for predictive query processing and analysis on road networks. The architecture of the *iRoad* system consists of three main modules, namely, the *state manager*, the *predictive tree builder* and the *query processor*, Figure 4.1. In this section, we present an overview of the *iRoad* System and give a brief description of its key components. Moreover, we focus on the interaction and workflow between these components under both the precomputed query result mode and the on-demand query result mode.

4.2.1 State Manager

The *state manager* is a user facing module that receives a stream of location updates from the moving objects being monitored by the system. The state manager maintains the following data structures. (1) An *R-tree* [58] that is generated on the underlying road network graph. It differs from the conventional *R-tree* in that at each leaf node, i.e., a node in the road network, in addition to storing the corresponding MBR, it also keeps track of two lists: (a) *current objects* that records the pointers to the objects around this node, and (b) *predicted objects* that maintains the predicted results of the objects that most likely to show up around that node. (2) A *trajectory buffer* that stores the most recent one or more nodes in the road network that are visited by the moving object in its ongoing trip. (3) A *predictive tree* such that root of a predictive tree is the current location of the moving object. Figure 4.2(a) gives an example of a set of objects moving on a road network, while Figure 4.2(b) depicts how the *predictive trees* are integrated within the basic data structures layout to facilitate the processing of predictive queries.

As we mentioned, the system can be running under either (1) a precomputed query result mode or (2) an on-demand query result mode. The first is the default mode inside the *iRoad* framework. In either modes, upon the receipt of a location update of a moving object, the *R-tree* is consulted and the new location is mapped to its closest node N_{new} in the road network. If the new node N_{new} is the same as the object's old node N_{old}, the object movement is not significant enough to change the system's state and no further action is taken. Otherwise, the object has moved to a different node and an evaluation of the impact of the object's movement is triggered in the system. We differentiate between the precomputed and the on-demand query result modes as follows.

Precomputed query result mode: In this mode, the predictive tree builder

is invoked immediately once the moving object changes its current node and, consequently, the predictive tree is either constructed from scratch or updated in response to the object's movement. Remember that the predictive tree is constructed from scratch if the incoming location update belongs to a new object that is being examined by the system for the first time. Also, the predictive tree is constructed from scratch if N_{new} is not a child of the root of the object's in-hand predictive tree. As will be described in Section 4.3, this case happens if the object decided not to follow the shortest path, e.g., made a u-turn or started a new trip. Otherwise, the tree is incrementally maintained. Note that, in this mode, the trajectory buffer data structure boils down to one single node (i.e., the current node) of the moving object because of the eagerness to update the predictive tree with the receipt of every location update. Hence, the past trajectory is entirely factored in the predictive tree.

On-demand query result mode: In this mode, the *trajectory buffer* stores all nodes the moving object passed by since the start of its current trip. Initially, We do not perform any computation until a query is received. Then, we identify the vicinity nodes within the time range determined by the query. Those nodes might contribute in the predicted results. For each object in these nodes, we construct its predictive tree and run a series of updates according to the list of passed nodes in its *trajectory buffer*, Figure 4.3. For example, in this figure, nodes *A, G, and E* are within the time range specified in the query at node B. Then, we construct the predictive tree for each object, O_1, O_2, O_3, in those nodes and update them according the passed nodes by each one. Obviously, O_1, O_3 will contribute in the predicted objects at node B, while O_2 will not contribute as node B is no longer a possible destination for O_2 based on its trajectory buffer. Then, we get rid of any data structure, i.e., the predictive trees and predicted results, directly once the query processing is completed and the results are carried

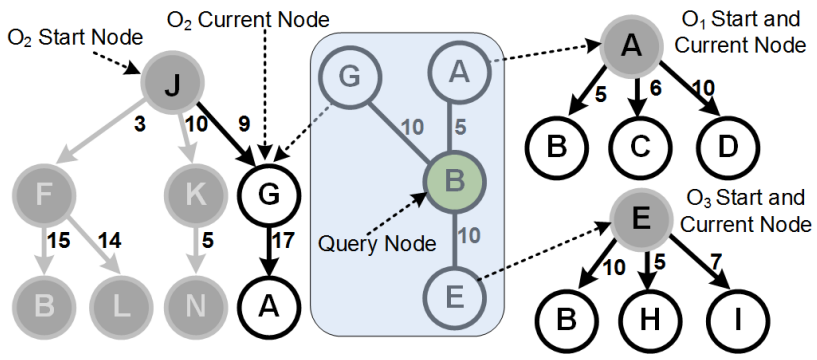


Figure 4.3: On-Demand Approach

back to the query issuer. We ending by adding the object's current node N_{new}, i.e,. Node B in this example, to the object's *trajectory buffer*.

4.2.2 Predictive Tree Builder

The *predictive tree builder* is the component that encompasses the predictive tree construction and maintenance algorithms. It takes as input, (1) the moving object's trajectory buffer, (2) the moving object's current predictive tree (if exists), (3) the tunable parameters ($\mathcal{T}$ and $\mathcal{P}$) that trade the prediction length and accuracy for system's resources, and (4) a user defined probability assigned function. The *predictive tree builder* reflects the most recent movements of the object (as recorded in the object's trajectory buffer) to the object's predictive tree. Upon the completion of a successful invocation of the tree builder, an up-to-date predictive tree rooted at the object's current location is obtained and the object's trajectory buffer is modified to accommodate the object's current node.

The predictive tree builder is invoked in two different ways. In a *precomputed query result* mode, the builder is invoked by the *state manager* upon the receipt

Algorithm 4 Predictive Tree Construction

Input: *Node* n, *Time Range* $\mathcal{T}$, Road Network Graph $G(N, E, W)$

Step 1. Initialize the data structures
Set n as the root of the Predictive Tree PT
Visited nodes list $NL \leftarrow \emptyset$
Min-Heap $MH \leftarrow \emptyset$
for all Edge e_i connected with n **do**
    Insert the node $n_i \in e_i \rightarrow MH$
end for
Step 2. Expand the road network and create the predictive tree
while the minimum time range T_{min} in $MH < \mathcal{T}$ **do**
    Get the node n_{min} with T_{min} from MH
    if The $n_{min} \notin NL$ **then**
        Insert $n_{min} \rightarrow PT$
        Insert $n_{min} \rightarrow NL$
        for all Edge e_j connected with n_{min} **do**
            Insert the node $n_j \rightarrow MH$
        end for
    end if
end while
Return PT

of every location update. The *state manager* pushes the incoming location update of a moving object O_i to the *predictive tree builder* that eagerly reflects the location update in the predictive tree of O_i. Afterwards, the tree builder updates the precomputed query results at every node in the road network that is on the shortest path route from the object O_i's current location.

In an *on-demand query result* mode, the *predictive tree builder* is invoked by the *query processor* once a query Q is received. The *predictive tree builder* consults the road network graph and retrieves a list of nodes $N_{vicinity}$ that are within the time distance determined by the query Q. Then, it pulls, from the *state manager*, the predictive trees and the trajectory buffers of moving objects whose current nodes are in $N_{vicinity}$. In other words, lazy or selective processing of moving objects that are believed to affect the query result is carried over without taking the burden of updating the predictive tree of every single moving object in the system.

4.2.3 Query processor

The main goal behind predictive query processing in the *iRoad* system is to be generic and to provide an infrastructure for various query types. This goal is achieved by mapping a query type to a set of nodes ($N_{vicinity}$) in the road network graph such that the query result is satisfied by predictions associated with these nodes. For predictive point queries as an example, the point query is answered using the information associated with the road network node that is closest to the query point. For predictive range queries, all nodes in the range are considered to compute the query result. For predictive KNN queries, we sort those predicted objects associated with $N_{vicinity}$ based on their probabilities. $N_{vicinity}$ is rationally expanded till K objects are retrieved, if visible.

In a *precomputed query result* mode, generic results are prepared in advance and are held in memory. The process is triggered by an update in object's location and the precomputed results are constructed/updated for *all* nodes along the shortest path route of that object. Therefore, most of the work is done during the location update time. Upon the receipt of a query, the *query processor* fetches the precomputed results only from nodes in $N_{vicinity}$, adapts them according to the type of the received query and gives a low latency response back to the user.

In an *on-demand query result* mode, nothing is precomputed in advance and all computation will be performed after the receipt of the user's query. $N_{vicinity}$ is identified and the predictive tree of objects whose current node belong to $N_{vicinity}$ are constructed/upadted as described earlier in this section. Then, the results are collected and adapted to the query type in a similar way to the precomputed result approach.

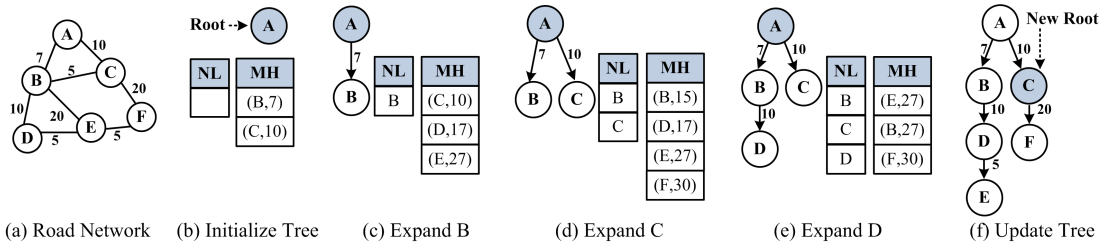


Figure 4.4: Constructing And Expanding The P-Tree Started At Node A.

4.3 Predictive Tree

In this section, we describe the proposed *predictive tree* index structure that is leveraged inside the *iRoad* framework to process predictive queries based on the predicted destinations of the moving objects within a time period $\mathcal{T}$. We first introduce the main idea and the motivation to build the *predictive tree*. After that, we provide a detailed description for the two main operations in the *predictive tree*: 1) predictive tree construction, and 2) predictive tree maintenance.

The idea of the *predictive tree* is to identify all the possible destinations that a moving object could visit in a time period $\mathcal{T}$ by traveling through the shortest paths. As there may only exist one shortest path from a start node to a destination node, we can guarantee it will be a tree structure (i.e., without any loop). The intuition for constructing the *predictive tree* with a time boundary $\mathcal{T}$ is based on two real facts: 1) most of the moving objects travel through shortest path to their destinations [34,57], and 2) majority of the real life trips are within a time period, e.g., 19 minutes [57, 59]. As a result, we only need to care about the possible destinations reachable through a shortest route from the object's start location within a bounded time period. Based on that, we build the *predictive tree* to hold only the accessible nodes around a moving object and assign a probability for each

one of them.

The *predictive trees* leveraged in the *iRoad* system significantly improves the predictive query processing efficiency for two main reasons. 1) The possible destinations of the prediction shrinks as a result of using the time boundary $\mathcal{T}$. Yet, prediction computation is performed on few number of nodes instead of millions of nodes in the underlying road network, e.g., road network of California state in USA has about 1,965,206 nodes and 5,533,214 edges [60]. 2) Inside the *predictive tree*, we maintain only those nodes with probability higher than a certain probability threshold parameter $\mathcal{P}$, e.g., 10%. By doing this, we cut down the computation overhead consumed for continuously maintaining the predicted results at each node in the predictive trees. Moreover, we control *iRoad* to focus on those nodes that more likely to be reached by a moving object. Yet, the query reported results can be more reasonable to users.

4.3.1 Predictive Tree Construction

Main idea. When a moving object starts its trip on the road network, we build a *predictive tree* based on its starting location to predict its possible destinations within a certain time frame $\mathcal{T}$. We propose a best-first network expansion algorithm for constructing predictive tree for time period $\mathcal{T}$, e.g., 30 minutes. We set the object's initial node as the start node, then, we visit the nodes and edges on the road network that are reachable using a shortest path from this start node [61]. The algorithm proceeds to traverse and process the edges in the road network based on the travel time cost from the start node until all the costs to the remaining edges are over $\mathcal{T}$.

Algorithm. The pseudo code for the predictive tree construction algorithm is given in Figure 4. The algorithm takes the road network $G = \{N, E, W\}$, a starting node n and a time range $\mathcal{T}$ as input. The algorithm consists of two main

steps:

- **Initialization.** We first initialize the predictive tree under construction by setting the start node n as the root of the tree. We also create the *visited nodes list* NL to store the nodes that have been processed by the algorithm so far. An empty min-heap, MH, is employed to order the nodes based on its distance to the root node n. After that, we insert the nodes that are directly connected with the root node n into the min-heap MH, (Lines from 2 to 7 in Algorithm 4).

- **Expansion.** We continuously pop the node n_{min} that is the closest to the root node from the min-heap. Then, we check if that node has been visited by our algorithm before, which means there was a shorter path from the root to this node n_{min}. If *visited nodes list* NL does not contain n_{min}, we insert the node n_{min} to it as well as a child to the current expanding branch of the predictive tree PT. After that, we insert to the min-heap MH the node n_j that is connected with the yet processed node n_{min} for further expansion. The algorithm stops when the distance between the next closest node in the min-heap is over the boundary $\mathcal{T}$, (Lines from 8 to 18 in Algorithm 4).

Example. Figure 4.4 gives an example for constructing a predictive tree for node A from the given road network. For this example, we set the time period $\mathcal{T}$ to 20 minutes. Figure 4.4(a) gives the original road network structure, where circles represent nodes and lines between nodes represent edges and the number on each edge represents the time cost to traverse that edge. In the first iteration, we start by setting the root of the tree to node A. Then, we insert nodes B and C into the min-heap, as they are the connected ones to the root node A, Figure 4.4(b). After that, we expand the closest child to the root, B, where we insert D and E into the min-heap MH and put B in the predictive tree and the

visited nodes list NL as well, Figure 4.4(c). In the second iteration, node C is the closest node to the root with travel time cost equals 10 minutes. Yet, we insert C into the predictive tree in addition to the visited nodes list. At the meanwhile, we put B and F into the min-heap. In this turn, the cost from root A to node B is 15 minutes, Figure 4.4(c). The next iteration picks up the node B again from the min-heap MH. The algorithm skips the processing for B, as it is already in the visited node list NL. Accordingly, we continue the expansion of node D, and insert E to the min-heap. After that, all the remaining nodes in the min-heap have the distance more than the time interval $\mathcal{T}$. The algorithm terminates and returns the constructed predictive tree, Figure 4.4(e).

Predictive Tree Expansion. As we mentioned above, the *iRoad* framework can answer predictive queries up to a maximum time boundary $\mathcal{T}$ in the future. This means, the in-hand predictive tree can not support prediction for object's trips that last longer than the specified future time $\mathcal{T}$. Therefore, to support those longer trips, we have to expand the tree to make sure that its depth under the object new location is not less than $\mathcal{T}$. In this section, we explain how to expand the predictive tree according to an object's movements.

Main idea. The main idea to expand a predictive tree is to locate the original predictive tree based on the start location of the object. After that, we take the new location of the object and make it as the new root for the just updated predictive tree. Then, the new destinations of the moving objects are the nodes located within the sub-tree underneath the new root. At the meanwhile, all the other nodes will be discarded for the further predication and processing. One important thing we have to take into consideration while we are inserting new nodes to the expanded predictive tree is that, all inserted nodes have to be reached in a shortest path from the object's start node. This means, we can postpone the deletion of the data structures used to build the original predictive tree until its

corresponding object finishes its trip. This decision is to avoid reconstructing the predictive tree from scratch, instead, the expansion process acts as one more iteration in the construction algorithm.

Example. Figure 4.4(f) gives an example for updating a predictive tree according to the object movement. Figure 4.4(e) gives the original predictive tree for the object o when it starts the trip at node A. In this example, the object moves from the starting location A to C. We get the current location of the object and locate it to the new node C in the original predictive tree. After that, the system returns C directly to the moving object as the root for its expanded predictive tree. Then, only the children of the new root, i.e., node F, will be considered in the further processing.

4.3.2 Predictive Tree Maintenance

In this section, we explain how to maintain the constructed predictive tree such that objects movements are reflected on the structure of the tree as well as the probabilities of its nodes.

Main idea. When a moving object starts a new trip, we get its start location and map it to the closest node in the road network graph. Then, we dispatch the *tree construction* module to construct a predictive tree with the root pointing to the object's start node. At this point, the *tree construction* computes the probability of each node in the in-hand predictive tree to be a destination to that object. If a node has a probability higher than the predefined threshold parameter $\mathcal{P}$, we insert the object to the list of predicted objects stored at this node. When the object travels to its next node in the tree, the *predictive tree maintenance* maintains the predicted results at each node in the tree as follows. (1) The predictive tree is updated by changing its root to point out to the object current node, and/or by extending the tree if its depth underneath the objects'

Algorithm 5 Predictive Tree Maintenance

Input: Probability Threshold $\mathcal{P}$, Time Range $\mathcal{T}$, Current Node n, *trajectory buffer*

```
1: if n = last added node in trajectory buffer then
2:     return;
3: else if (n is the first movement of o or movement to n violates shortest path then
4:     trajectory buffer ← ∅
5:     Construct the predictive tree PT for o rooted at n
6: else if Tree depth underneath n < T then
7:     Expand the predictive tree PT
8:     Prune the predictive tree PT
9:     Delete o from the list of predicted objects at n_i
10: else
11:    Prune the predictive tree PT
12:    Delete o from the list of predicted objects at n_i
13: end if
14: for each node n_i in PT do
15:    Compute the probability P(n_i)
16:    if P(n_i)≥ P then
17:        Insert o → list of predicted objects at n_i
18:    end if
19: end for
20: Insert n → trajectory buffer
```

current node is less than the time period $\mathcal{T}$. (2) All nodes other than those in the current subtree underneath the new root will be pruned. Hence, we remove all occurrences of that object along with its probability from the predicted results at those pruned nodes. (3) The list of predicted objects at the current subtree will be updated by either adding the object to it, or modifying its probability in case the object is already there. In fact, the *predictive tree maintenance* is a core operation inside the *predictive tree* index structure, as it is responsible for maintaining the tree and hence updating the predicted objects at each node in the tree and reflecting that to the nodes in the original road network graph.

Algorithm. Algorithm 5 gives the pseudo code for the *predictive tree maintenance* algorithm. The algorithm takes the *trajectory buffer* for the moving object o, and its current node n, in addition to two system parameters, the probability

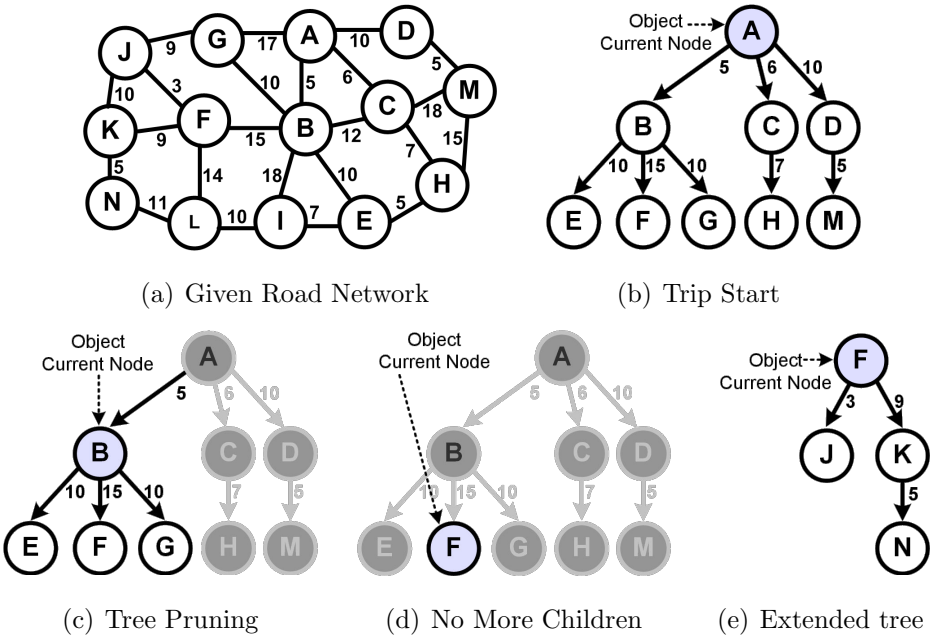
threshold $\mathcal{P}$ and time range for maximum predictable period in the future $\mathcal{T}$. Initially, we check if the object is still around its original node, then, the algorithm does nothing, (Line 2 in Algorithm 5). Otherwise, the algorithm preforms one or more of the following possible steps.

(1) *Construct New Predictive Tree.* The algorithm examines if the object starts a new trip by checking for two cases; (a) it has not passed by any previous node, or (b) its recent movement violates shortest path routing which internally means the object moves to a node that is not a child of its previous node within its predictive tree. For any of these cases, the algorithm constructs a new predictive tree PT for the object o with a root points to the object's current node n, (Lines 4, and 5, in Algorithm 5).

(2) *Maintain Existing Predictive Tree.* If the object o takes one more step toward its destination through a node in its original predictive tree PT, we need to make sure that the depth of the subtree underneath its current node n is greater than or equal to the maximum prediction time $\mathcal{T}$. If this is the case, we prune the in-hand tree by cutting off all nodes not in the subtree underneath the new root n, Line 11 in Algorithm 5. Pruned nodes are no longer possible destinations to the object o as they can not be reached in a shortest path from the object's current location. If it is not the case, the algorithm dispatches the *predictive tree expansion* submodule, explained earlier in this section, to extend the original tree PT such that the time distance from the current node n to the end of PT is at least equal to $\mathcal{T}$. Then we prune the updated tree, (Lines 7, and 8 in Algorithm 5).

(3) *Assign The Probability.* After having the modified predictive tree PT for the object o, we compute the probability for each node n_i in PT being a possible destination to o, Line 15 in Algorithm 5. This is done according to the probability assignment model, Section 4.1.3.

(4) *Maintain The Predicted Results.* The aim of this step is to make sure that



(a) Given Road Network (b) Trip Start

(c) Tree Pruning (d) No More Children (e) Extended tree

Figure 4.5: Example For An Object Trip And Predictive Tree Maintenance

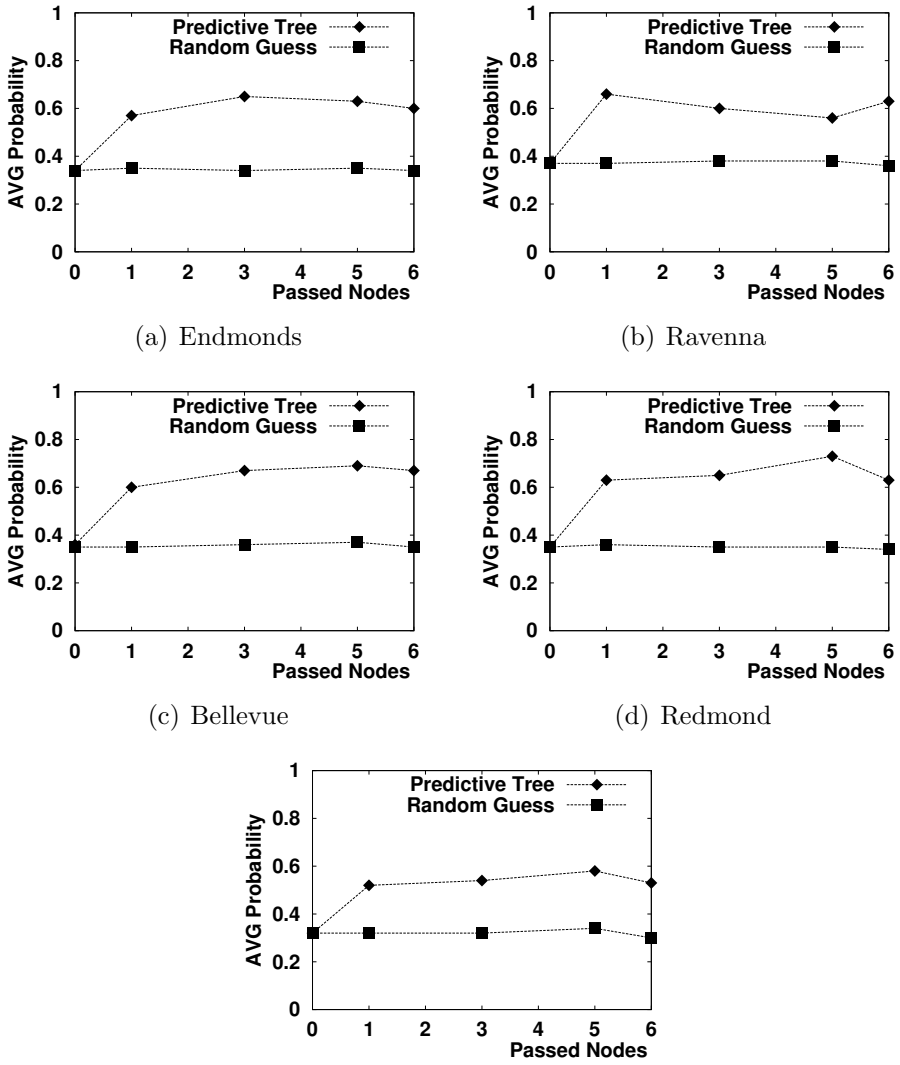
the list of predicted objects stored with each node in the road network get updated according to the effect of the object's movements. This is done by performing the following two actions. (First) Within the pruning process we have to delete any occurrence for the object o from the list of predicted objects at the excluded nodes, (Lines 9, and 12 in Algorithm 5). (Second) After computing the probability of each node in the current PT, we check if this probability is above the given probability threshold $\mathcal{P}$, we insert a new record to the list of predicted results at the node n_i, in case that o is not in this list. Otherwise, we just modify its corresponding probability value. This record consists of identifier for the object o, its probability $\mathrm{P}(n_i)$ to show up at n_i, the travel time from the tree root to the node n_i,(Lines from 15 to 18 in Algorithm 5). This list of predicted results will be used later by the query processor module to evaluate users' queries. Finally, the algorithm updates the *trajectory buffer* to accommodate the current node n,

Line 20 in Algorithm 5.

Example. Figure 4.5 illustrates how the *tree maintenance* algorithm works. Given the road network graph shown in Figure 4.5(a), and assuming that the object o has started its trip at node A, we construct its predictive tree rooted at node A and its depth is 20 minutes, Figure 4.5(b). When the object o moves to the next node B in its trip, then the algorithm preforms some consequent actions, Figure 4.5(c). First, it assures that the depth of the tree under the object current node B is at least equal to the maximum prediction time $\mathcal{T}$, which is valid in this step. Second, it updates the list of predicted objects in all nodes in the original tree rooted at node A. This is done by modifying the probability at nodes B, E, F, and G, and deleting the occurrence of o at nodes A, C, D, H, and M as they are no longer possible destinations to o. Third, we prune the tree by cutting off all nodes that are not possible destination to o via a shortest path from start node A. So, we exclude the nodes A, C, D, H, and M. This step reduces the CPU cost required later to compute and update the probability at each reachable node. As the object o travels to the next node F, we find that no further children of F in the current predictive tree, Figure 4.5(d). Yet, we need to check if it is the end of a trip, in case that the tree can not be extended. Here, it is not the case, so we extend the subtree under F, compute the probability of the nodes in this subtree, modify the list predicted objects at all nodes under node B, and finally prune the nodes under B except the subtree of F, Figure 4.5(e).

4.3.3 Querying The Predictive Tree

The main idea of processing predictive queries in *iRoad* system is to have generic results prepared in advance and held in memory. So for coming queries, the *query processor* module fetches those results, adapts them according to the type of received query and responds to users in a very fast response time.



(a) Endmonds

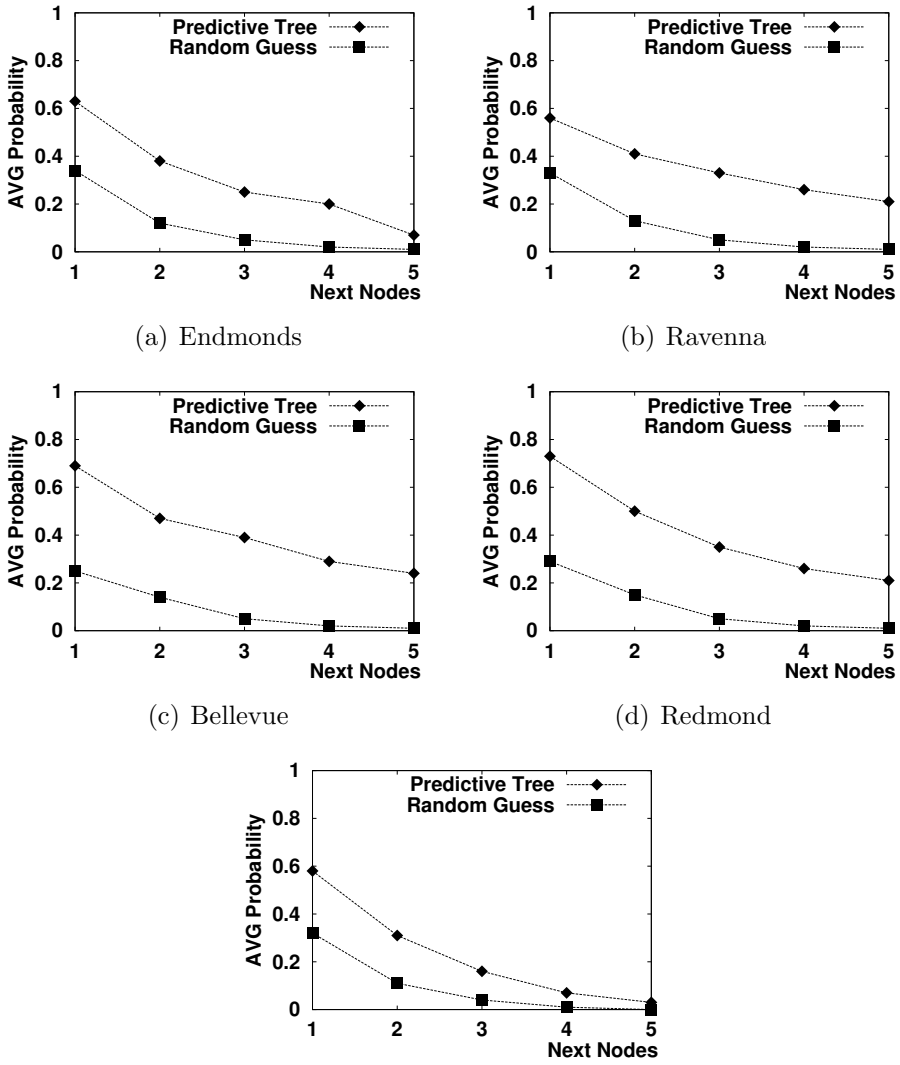
(b) Ravenna

(c) Bellevue

(d) Redmond

(e) Kenmore

Figure 4.6: Accuracy Evaluation For Short Term Prediction



(a) Endmonds

(b) Ravenna

(c) Bellevue

(d) Redmond

(e) Kenmore

Figure 4.7: Accuracy Evaluation For Long Term Prediction

4.4 Experimental Evaluation

This section experimentally evaluates the performance of the *predictive tree* index within the *iRoad* framework.

4.4.1 Experimental Setup

In all experiments of this evaluation, we use real road network data of the Washington state, USA. For the accuracy evaluation, we use real data sets for cars trajectories around the area of Seattle [62, 63]. For the scalability and efficiency experiments, we use the Minnesota traffic generator [64] to generate large sets of synthetic moving objects on the Washington real map. The number of objects varies from 10K to 50K objects per set. To test the introduced approaches against different workloads, we build a query workload generator. The number of generated predictive queries varies from 10K to 50K queries per workload. In case of predictive range queries, the query sizes range from 0.01 to 0.08 of the total size of the underlying road network graph. For predictive KNN queries, we increase K from 5 to 25. The future times $\mathcal{T}$ for the generated queries vary from 5 to 25 minutes. The probability threshold $\mathcal{P}$ varies from 0.02 to 0.10. Predictive point query is the default query type unless we mention a different one. In each run we use 3k queries for warming up before we start measuring the experimental metrics, i.e, CPU cost, memory overhead, and accuracy percentage. Also, we let the system to have a sufficient number of moving objects before we start firing the queries. All experiments are based on an actual implementation of the *predictive tree* and the whole *iRoad* framework. All the components are implemented in C# inside visual studio 2013 with .net framework. Our implementation source code for the *predictive tree*, and sample datasets are accessible through [65]. All evaluations are conducted on a PC with Intel Xeon E5-1607 v2 processor and 32GB RAM,

and running Windows 7.

4.4.2 Accuracy Evaluation

In this section, we compare the accuracy achieved by employing the *predictive tree* versus the Mobility Model [8]. We feed both of them with current objects trips and without historical trajectories. The prediction accuracy is measured by the probability each approach assigns to the actual future node of an object. We use real objects trajectories at five different cities around the Settle area in Washington, USA. In Figure 4.6, we examine the effect of the number of passed nodes on the accuracy of the next node prediction as an indicator for short term prediction. Clearly, the *predictive tree* gains benefits from the object's current trip by pruning the possible destinations to lower number, while the *mobility model* does not prune at all. Hence, the *mobility model* can be best described as a random walk model, in case of the absence of historical knowledge. So, in the accuracy test figures, we will label the history-free version of *MM* as *Random Guess*.

As given by Figure 4.6(d), the *predictive tree* can achieve up to 73% accuracy in predicting the next node. Overall, the average accuracy for the *predictive tree* is about 67% while it is less than 40% for the *mobility model* approach. The fall down in Figure 4.6(b) is caused by the high density of the road network in this area which increases the number of possible destinations, hence, decreases the prediction accuracy. In general, there are two main factors that affect the prediction accuracy inside the *predictive tree* . (1) The extent to which a moving object follows the shortest paths in its travel, and (2) the average degree of a node in the underlying road network. In attempt to test the accuracy pattern of each approach in long term prediction, we let the objects pass by a sufficient number of nodes, i.e., five nodes, then we measure the probability given to each of the five nodes next to the objects' current node, Figure 4.7. As expected, the accuracy

decreases while we look far in the future. However, the *predictive tree* still acts better.

4.4.3 Scalability Evaluation

For the scalability evaluation, we compare the two different approaches, precomputed and on-demand, of employing the predictive tree inside the *iRaod* framework. As discussed in Section 4.2, the earlier is the default approach inside the *iRoad* framework where the predicted objects are computed and stored at each node in advance. While in the latest, we postpone all computations until a query is received. In these sets of experiments, we study the scalability of the *predictive tree* within the two aforementioned approaches with numbers of queries, query sizes, i.e., for predictive range queries, number of moving objects, future prediction time $\mathcal{T}$, and different levels of the probability threshold $\mathcal{P}$. For each test, we measure the average CPU time and the maximum memory footprint.

Scalability with number of queries. As depicted in Figure 4.8(a), the CPU cost for the on-demand approach is almost steady around 75 millisecond per query. While for the precomputed approach, it decreases from 8 millisecond per query to less than half millisecond when the number of queries increases from 20K to 100K. This concludes that the precomputed approach significantly saves the CPU time. The reason is that once we compute and save the predicted answer at a node, it serves as many queries as needed without extra computation. However, the on-demand approach acts better w.r.t. memory overhead, Figure 4.9(a), as it does empty the data structures once the query is processed.

Scalability with query size. In this set of experiments, we study the scalability of each approach with large query sizes in predictive range queries. We vary the query size ratio from 0.01 to 0.08 of the the given road network graph. We found that the average CPU cost per query for the precomputed is almost steady

at 3MS, while it varies from 194MS to 755MS for the on-demand. This means the CPU cost for the on-demand approach linearly increases when the query size increases, Figure 4.8(b). Intuitively, this is a rational behavior, as the on-demand performs computations for more objects with bigger sizes. Obviously, from the CPU time perspective, the precomputed approach is faster by orders of magnitude. From the memory overhead perspective, Figure 4.9(b), the on-demand approach is more memory friendly. However, it has some slightly increasing trend as a result of keeping the predictive trees longer time till the query is processed as a whole.

Scalability with prediction time $\mathcal{T}$. When we examine the scalability with the maximum prediction time $\mathcal{T}$ the *predictive tree* can support, we can notice a remarkable increase in both approaches w.r.t. the average CPU cost, Figure 4.8(c). However, there is a significant difference between the minimum/maximum CPU cost for each approach. The lowest CPU cost for the precomputed approach is 0.5MS at $\mathcal{T} = 5$ minutes and 2MS for the on-demand, while the maximum is 45MS and 5395MS respectively at $\mathcal{T} = 25$ minutes. The CPU time overheads come from the extra computations required to construct bigger predictive trees and maintain more reachable nodes. On the other hand, the memory overhead for the on-demand is almost stationary, i.e., from 19 to 22MB. This is extremely less than the min/max overheads for the precomputed, i.e., 219MB and 3.4GB, Figure 4.9(c).

Scalability with probability threshold $\mathcal{P}$. Figure 4.8(d) and Figure 4.9(d) illustrate the effect of increasing the probability threshold $\mathcal{P}$ from 0.02 to 0.10 on the scalability of the *iRoad* in general. For the on-demand, the average CPU cost reduces from 86 to 77MS, and the memory overhead from 50 to 20MB. Also, for the precomputed, the average CPU cost reduces from 5 to 3MS and the memory overhead from 724 to 702MB. The justification for this reduction comes from the

fact that we exclude the nodes with probability less than $\mathcal{P}$.

Scalability with number of objects. The more objects to consider, the more predictive trees to build, and the extra computation and memory overhead to pay. That fact is obviously given by Figure 4.8(d) and Figure 4.9(d). The average CPU cost for the precomputed is between 4MS at 10K objects and 18MS at 50K objects. This is dramatically less than the one for the on-demand approach, 112 to 667MS. From the memory overhead view, the on-demand consumes 20MB more to answer the same number of queries with 10K objects and 88MB with 50K objects, while the precomputed adds up to 670MB with 10K objects and 3GB with 50K objects.

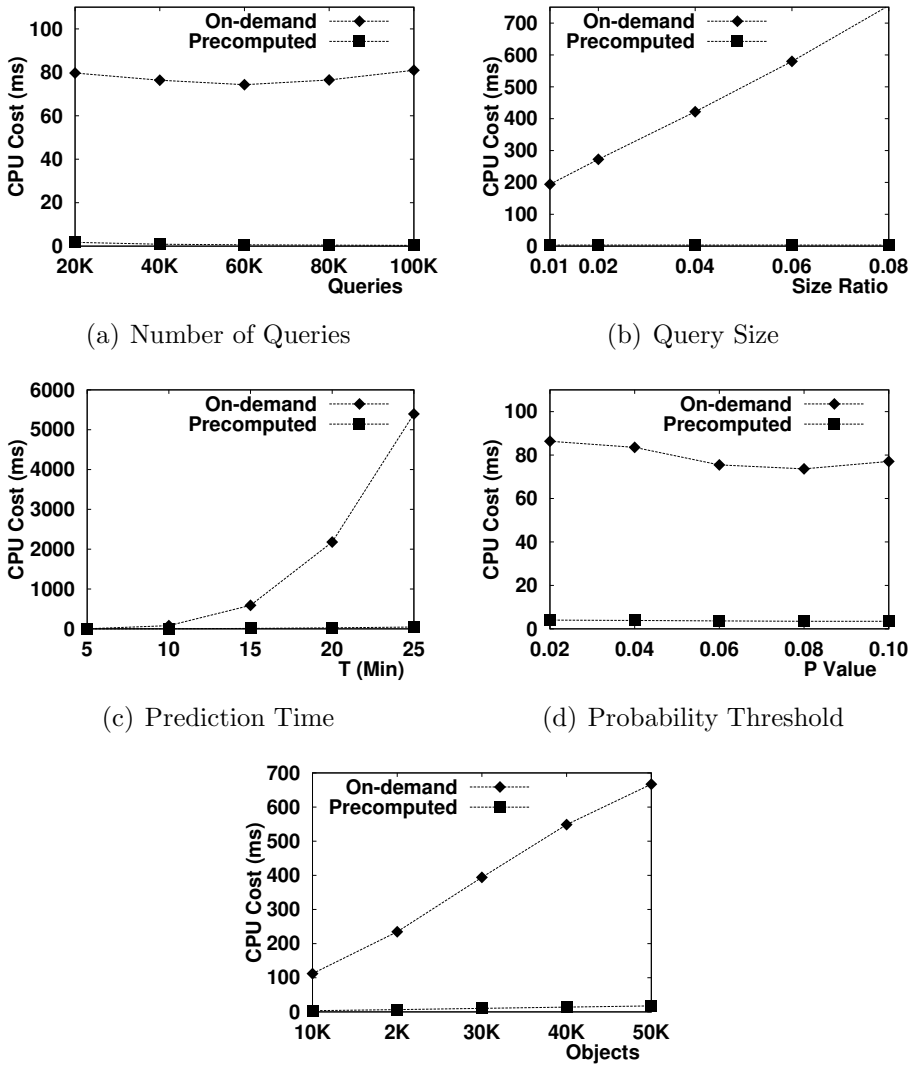
4.4.4 Efficiency Evaluation

In this set of experiments, we evaluate the effect of the different parameters on the two main operations of the *predictive tree* index structure, namely *tree construction* and *tree maintenance*. We ignore querying the tree operation, i.e., *query processor*, since it just retrieves the answer from the query node, hence, its cost is negligible compared to the other two operations. We study those operations within the evaluation of the precomputed approach as it is the dominant w.r.t. the CPU cost. Figure 4.10(a) gives the impact of increasing the number of queries. We can notice the average CPU cost drops down for both operations as the number of queries goes up. As mentioned earlier, the reason is that once a predictive tree is constructed, it can participate in processing as many queries as needed without extra computation, and the same for the maintenance operation. Therefore, the average CPU time decreases while the number of queries increases. From Figure 4.10(b), we can conclude that both operations are significantly impacted by the maximum prediction time $\mathcal{T}$. The average CPU cost for the *tree construction* starts with 0.43MS at $\mathcal{T} = 5$ minutes, and tends to 31.57MS at $\mathcal{T} = 25$ minutes.

A similar trends happens with the *tree maintenance* operation where it charges the CPU 0.05 and 14.03MS at $\mathcal{T} = 5$ and 25 respectively. The impact of tuning the probability threshold $\mathcal{P}$ is provided in Figure 4.10(c). The major recorded influence is on the *tree maintenance* operation. That is because larger probability means less detailed tree, i.e., include only those nodes with probability $\geq \mathcal{P}$. We can notice that the average CPU cost for the tree maintenance drops from 0.47MS at $\mathcal{P} = 0.02$ to 0.09MS at $\mathcal{P} = 0.10$. Finally, Figure 4.10(d) provides the consequence of large numbers of moving objects on the two fundamental operations. Normally, we construct and maintain a predictive tree for each extra moving object. Hence, the average CPU cost increases for both the *tree construction* and *tree maintenance* operations. It is also remarkable in all graphs in Figure 4.10 that *tree construction* represents the big portion of the CPU cost relative to *tree maintenance* operation.

4.4.5 Experiments Summary

From the accuracy evaluation set of experiments, it is obvious that the predictive tree achieves more accurate short and long term prediction compared to the most related work. From the scalability study, the predictive tree proves its ability to scale up to efficiently process heavy query workloads and handle large number of moving objects. It is also concluded that the precomputed mode is CPU friendly while the on-demand mode is memory friendly. In addition, the controllable parameters ($\mathcal{T}$ and $\mathcal{P}$) are assessed to adjust the overall system performance such that a graceful degradation takes place under periods of heavy workloads. Moreover, the study of query type assured that predictive tree construction and maintenance operations are not affected by the query type. This confirms the generic nature of the predictive tree index.



(a) Number of Queries

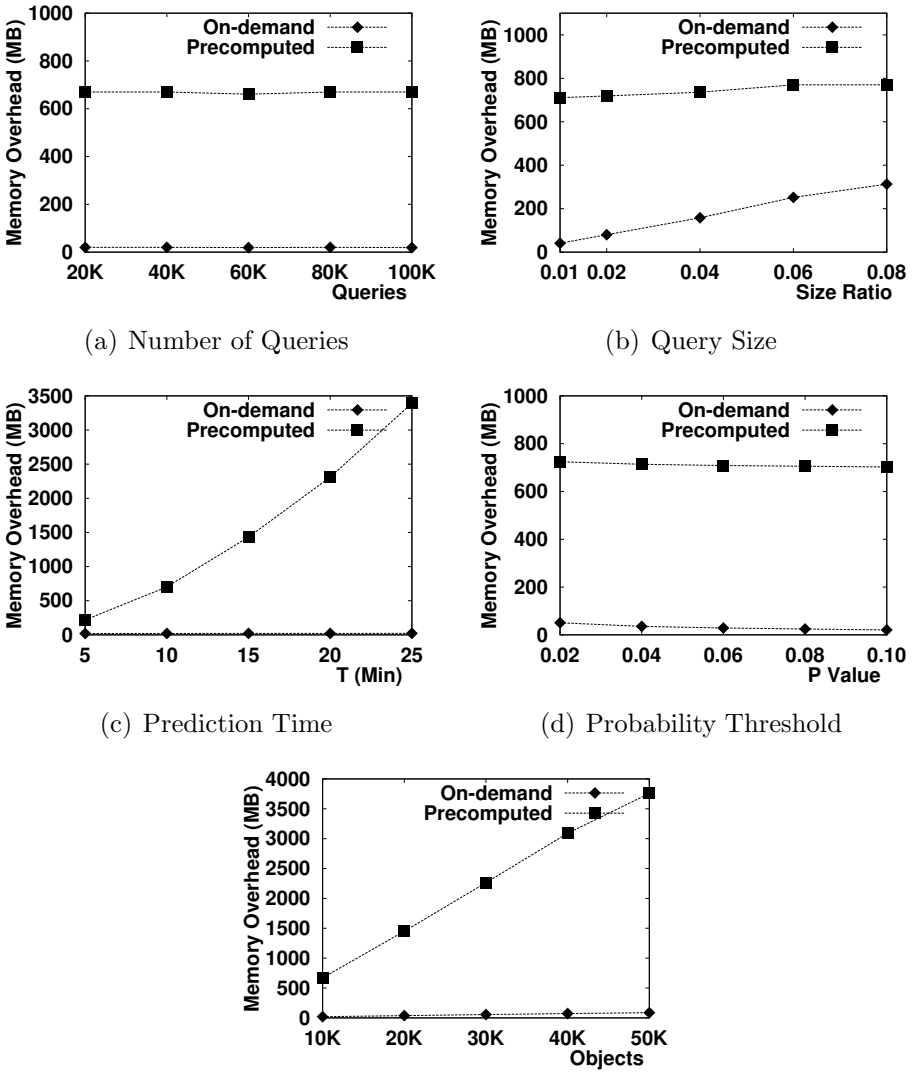
(b) Query Size

(c) Prediction Time

(d) Probability Threshold

(e) Number of Objects

Figure 4.8: Scalability Evaluation (CPU Time)



(a) Number of Queries

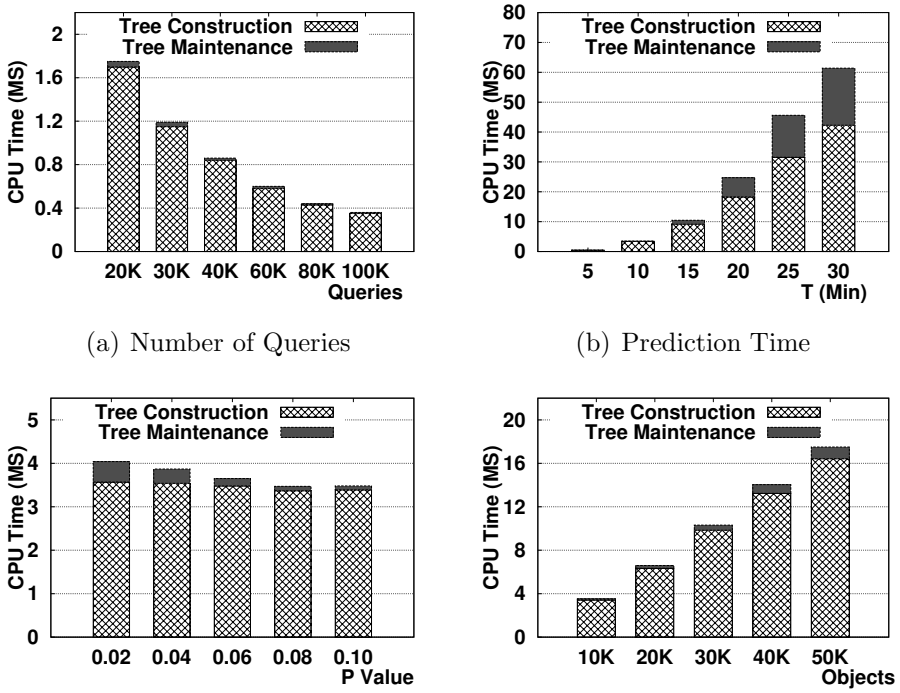
(b) Query Size

(c) Prediction Time

(d) Probability Threshold

(e) Number of Objects

Figure 4.9: Scalability Evaluation (Memory Overhead)



(a) Number of Queries

(b) Prediction Time

(c) Probability Threshold

(d) Number of Objects

Figure 4.10: Efficiency Evaluation Of Main Operations In The Predictive Tree

Chapter 5

Extensions

This chapter discusses how we extend the work introduced in Chapters 3 and 4 to Support different types of spatial predictive queries for objects on Euclidean space and road network graphs respectively.

5.1 Extensibility of Panda

In the previous section, we have discussed the generic framework of *Panda* as a predictive spatio-temporal query processor, and we have elaborated its main modules that compose its core. In this section, we illustrate how *Panda* can be extended to support a wide variety of predicative spatio-temporal queries and how it can be harmonized according to the nature of the underlying data. Basically, this is accomplished through the optimized implementation of the generic function *UpdateResults* to serve the needs of the underlying *query type*, e.g., aggregate, range, or k-nearest-neighbor queries, and the underlying *data nature*, e.g., moving data, or stationary data.

5.1.1 Query Type

In this section we study the extensibility of *Panda* to support the evaluation of three main predictive query types, namely range, K-NN, and aggregate query. However, this extensibility is not limited to these types only because the universal nature of the aforementioned data structures and algorithms that can be easily tailored to other feasible predictive spatio-temporal queries.

Range Query Processing

Idea. A predictive range query is defined by two elements, a rectangular query region R and a future time t, and asks about the objects expected to be inside the determined query region after the specified future time. *Panda* starts the range query processing by getting the grid cells that overlap the query region. Those cells are divided into two groups. The first one contains the cells that already have precomputed answers that we need to retrieve them only without further processing, while the second group contains the overlapped cells that their answers have to be computed from scratch. For each cell c_i in the second group, *Panda* visits the travel data structure *TTS* and gets the list of the reachable cells C_R to the cell in hand c_i. For each object in a reachable cell, *Panda* applies the prediction function and checks if that object is predicted to arrive at c_i after the desired future time, if this is the case, this object with its probability is appended the result of c_i. To do that, the *UpdateResult* function is implemented to precisely serve the predictive range query processing such that it is able to build up the result of the cell in hand as well as the final query result by continue appending the objects identifiers and with their probabilities. Also, the *UpdateResult* function is used to form the final result of a range query by merging the objects in the precomputed cells and those that have just been computed. Yet, the returned

answer encompasses the list of objects expected to be inside the overlapped cells after the desired time unites in the future.

Example. The illustrated example in the two phases in Section 3.3.2 is sufficient to explain the processing of predictive range query.

K-NN Query Processing

Idea. A predictive K-NN query has two parameters, a specified location point and a future time, and enquires about the K objects expected to be the nearest to that location after the given time. Initially, *Panda* locates that point into a corresponding grid cell which will be checked if having a precomputed answer for the K or more objects expected to appear at this cell after the determined future time. If this is the case, the precomputed answer is simply returned without extra computation. Otherwise, in the case that there is no precomputed answer, *Panda* computes it in similar way to range query in order to find the satisfactory K objects. In the case that the precomputed answer or the yet computed answer has objects less than the desired K, *Panda* expands the query location by adding the nearest adjacent cell and recall the original computation steps on this recently added cell. This process is repeated until the query is satisfied. Definitely, *Panda* has a different interpretation of the term nearest here, which reflected as the objects with the highest probability to be within the nearest cell(s) of the query point. This interpretation inherited from the nature of the underlying prediction function which locates the anticipated location of an object in a cell size area rather than predicting its exact future point. The *UpdateResult* function is exactly implemented as in the range query without difference. The results for both queries are lists of objects.

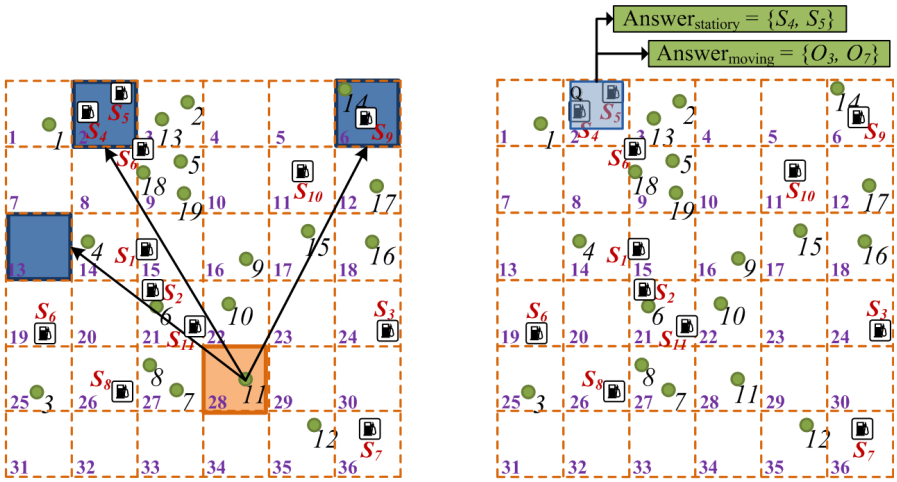
Example. If we consider Q_{30}, described in Section 3.3.2, as a K-NN query with $K = 5$, $t = 30$, and location point = L which is represented by a star in

Figure 3.4(a), *Panda* will start by locating L in C_{19}. Since C_{19} has a precomputed answer for $t = 30$, then it fetches it. Since this answer has only three objects while the query asks for five, so it is not sufficient as a final query result. Accordingly, C_{20} the nearest cell to L is added to the query region, then we compute its answer which will return $\{O_{15}\}$ as the set of the expected objects for C_{20} after 30 minutes. Now, we have four objects in for both cells C_{19} and C_{20}, which means that we still have not the desired K. Consequently, we iterate on the next nearest cell, C_{26}, which gives answer = $\{O_{12}\}$. Finally, we have fulfilled the query requirement and the final answer will be $\{O_5, O_{18}, O_9, O_{15}, O_{12}\}$.

Aggregate Query Processing

Idea. A predictive aggregate query consists of a query region R and a future time t, and it finds out the number of objects $\mathcal{N}$ predicted to be inside that region after the given time. Clearly, it looks similar to the range query, hence, it follows the same exact steps. However the format of the final result is different where the aggregate query wants the number of objects $\mathcal{N}$ rather than the objects themselves. Therefore the *UpdateResult* function is customized to sum up the number of objects instead of appending objects to the result list. So it is employed to compute the the expected number objects to be inside each single cell in R after time t, and also aggregate those numbers to obtain the final query result.

Example. Considering Q_{30} as an aggregate query, *Panda* will apply the same steps in Figures 3.4 and Figures 3.5, and the final result will give the number of objects expected to arrive at the query region after 30 minutes which will be $\{3\}$.



(a) Candidate destinations of O_{11} (b) Query Results

Figure 5.1: Query processing on (Stationary vs Moving) Data

5.1.2 Data Nature

The shared data structure, the isolation between the precomputed areas and the query region, and the generic query processor frame the infrastructure that allows *Panda* to support a wide variety of predictive queries including range queries, k-NN queries , and aggregate queries. Apparently, each query type can operate the same data structure in a different way to compute its own answer. As described in the mentioned algorithms, to handle the case of predictive range or K-NN queries, the precomputed answer in the *query list (QL)* maintains a list of objects expected to be inside a query region R after future time t, while in the case of predictive aggregate queries, the same list is used to carry a number.

In this section, we explain the ability of *Panda* to act suitably according to the nature the underlying data whether it represents stationary or moving objects. We provide some examples to illustrate this flexibility feature.

Stationary Data

In this class of data, the points of interest that a user query questions about are static objects that almost do not have mobility nature. Examples for stationary data include gas stations, restaurants, theaters, cinemas,...etc. We refer to a stationary object i by S_i. Basically, *Panda* can preserve information about the underlying stationary objects in addition to the moving objects using its grid data structure where each object is linked to its corresponding cell. The predictive query in this case is tight to a certain moving object not to any of the stationary objects. For example, as illustrated in Figure 5.1, a moving object O_{11} sends a range query to find out the gas stations within half mile of its future location after 40 minutes, of course without releasing its intension. To process this query, *Panda* initially finds the current cell, C_{28}, in which O_{11} is currently moving. Thus, C_{28} is added to the current trajectory of O_{11}. Then it obtains the list of its candidate destination cells after 40 minutes which will include $\{C_2, C_6, C_{13}\}$. This is achieved by accessing the travel time structure under the column of C_{28}. After that, it determines the highly anticipated destination in a granularity equivalent to a grid cell area. That is accomplished by calling our prediction function feeded with the current trajectory of O_{11}, Figure 5.1(a). Since C_2 is the predicted destination, a half mile query region is placed starting from its center. Therefore, the set of the stationary objects that intersect with that region is returned as the query results which will be $\{S_4, S_5\}$, Figure 5.1(b).

Moving Data

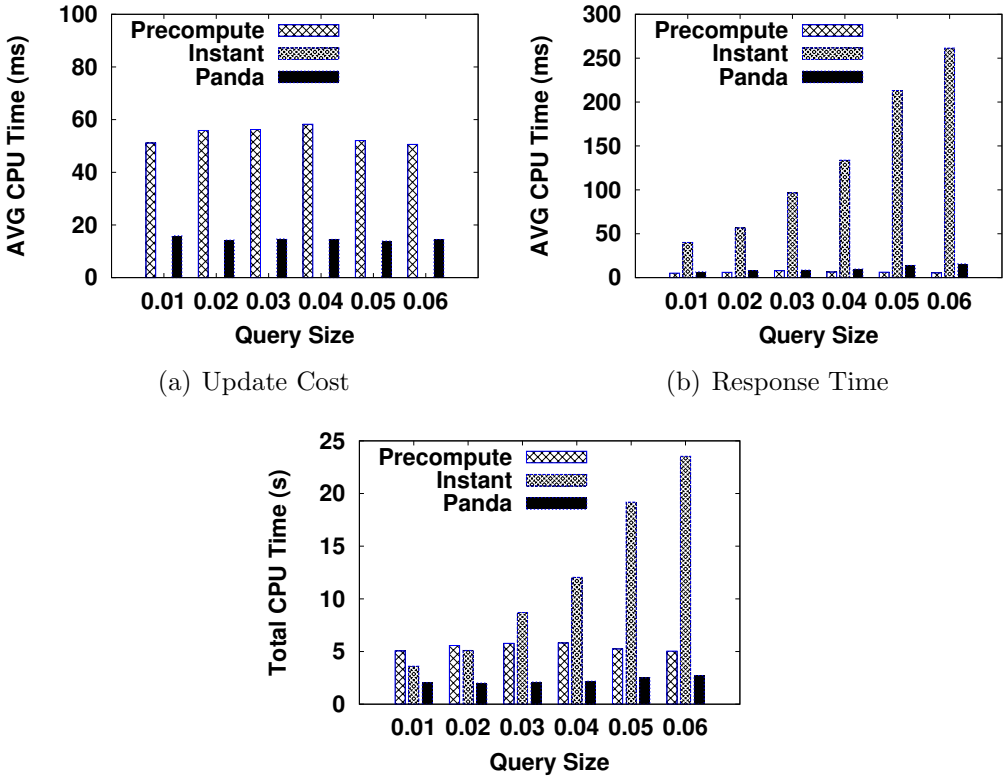
This class of data has both the spatial and temporal features, so they dynamically change their locations. Predictive query on this class of data has two options. The first option, is to be connected to a specified moving object which imposes *Panda*

to do a preprocessing step by calling our prediction function $\hat{F}$ to identify the possible destination cell c_d for that object at the given time t. Accordingly, the query is located at that destination cell c_d. Then it ordains a customized version of the predictive query processor according to the query type as explained in the previous section. For example a moving object O_{11} asks about the moving objects, (i.e., friends with mobile phones, some moving service like police cars), expected to show up within half mile of its future location after 40 minutes. To process this query, we find that C_2 is the most predicted target cell after 40 minutes based on the recent trajectory of O_{11}. After that, we dispatch *Panda* query processor to evaluate a range query positioned at the center of C_2. The expected answer will be $\{O_3, O_7\}$, Figure 5.1(b).

The second option for predictive query on moving data, is to be connected a static area or location in the given space, for example a store wants to notify the cars expected to be within two miles of its location about offers after 3o minutes. In this situation, the query processor is fired directly without any additional preparation steps. The example illustrated in Section 3.3.2 fits in this class of data.

5.1.3 Effect of Query Size

We proceed to study the scalability of *Panda* with large query sizes. This set of experiments, Figure 5.2, illustrates that *Panda* can save about 50% of the CPU time required to answer a user query while preserving its response to be almost equals to the fastest one with a vast dropping in the update cost. The other set of experiments, Figure 5.3, evaluates the influence of the threshold tweaking on the scalability of *Panda* with respect to the query size in three different query workloads. The used workloads are 20K queries, 40K queries, and 80K queries. The provided figures suggest the use of a large threshold value with small sized



(a) Update Cost

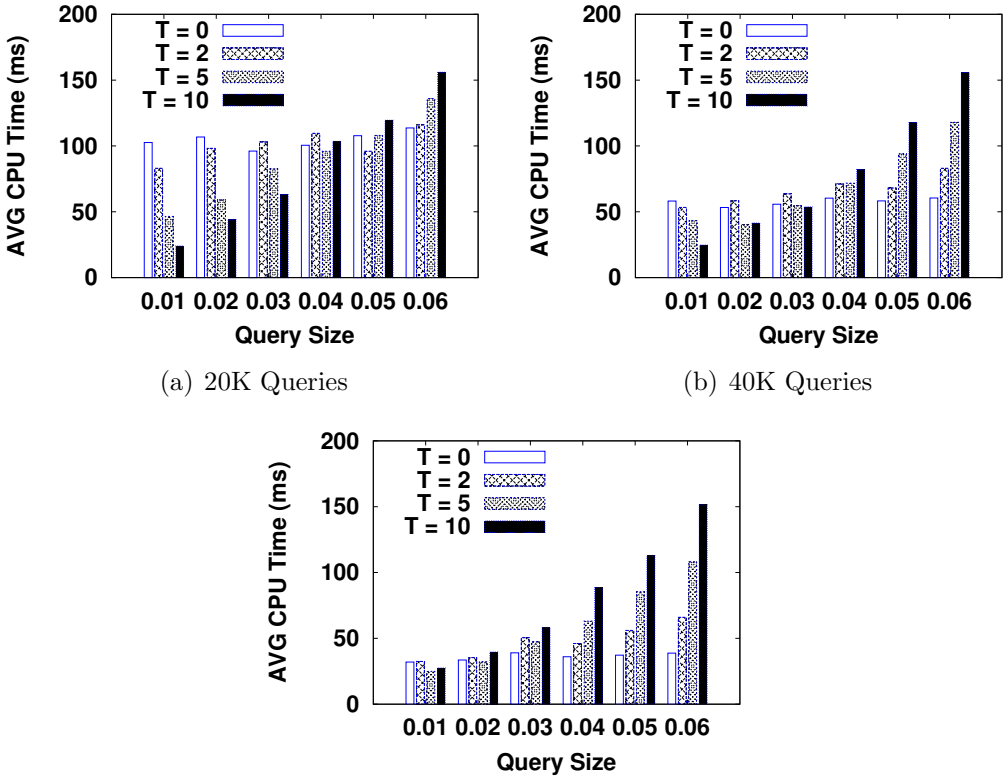
(b) Response Time

(c) Processing Time

Figure 5.2: Scalability with Query Size

queries and a small threshold value with the ones with larger query size.

In one word, we can establish that the main reason behind the capability of *Panda* to achieve smooth scalability comes from its ability to adapt according to the nature of the moving objects behavior and the heaviness of the query workload.



(a) 20K Queries

(b) 40K Queries

(c) 80K Queries

Figure 5.3: Effect of Threshold Tuning on Scalability w.r.t Query Size

5.2 Extensibility of P-Tree

Here, we discuss we to extend the query processing inside the *P-Tree* such that common types of spatial predictive are supported.

5.2.1 Extended Query Processor

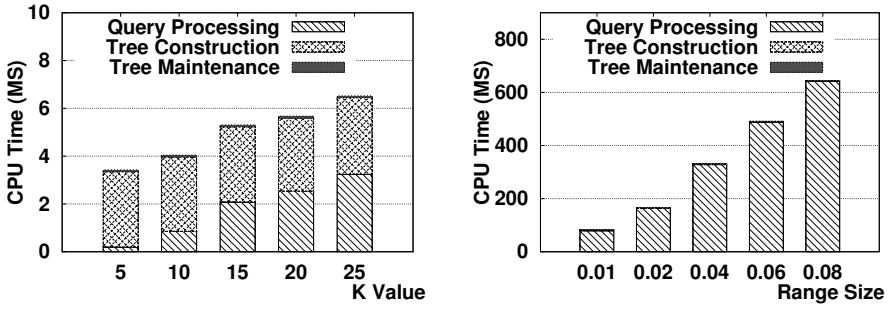
As we mentioned in Section 4.3.3, P-tree stores generic results that are either precomputed in advance while new objects start trips and updated with objects' movement or computed instantly upon query arrival.

Since the computation of predicted answer is already accomplished by the *predictive tree construction* algorithm and maintained by the *predictive tree maintenance* algorithm, thus, the of task of *query processor* module becomes straight froward. What we need to do here is,

- **First.** we find out the node of interest for which the query is asking about its predictable objects. Then, we retrieve the list of the *predicted objects* saved with this node. Then we shape the answer to match the user's query type.
- **Second.** we examine if the query invokes the predicted results at many nodes, (e.g., predictive range query), rather than a single node, then, we combine the answers at those nodes of interest into a single basket by taking the union of the *predicted objects* lists associated with them. This will get rid of redundant objects. To unify the probabilities for object predicted to appear in more than one node of interest with different probabilities, we use the maximum probability among its occurrences.
- **Third.** according the query type,(i.e., predictive range, KNN, aggregate) we adjust the in hand generic results. For predictive range, we return the list of objects along with their probabilities to appear at the query node(s). For predictive aggregate query, users concern more about the number of objects expected to show up around the query node(s). So, we return that number as accumulation of objects' probabilities. For the case of predictive KNN query, we pick up those K objects with highest probabilities.

5.2.2 Effect Of Query Type

This section studies the effect of query type on the performance of the *predictive tree*. To be more specific, we study the effect of K value in case of predictive KNN query, Figure 5.4(a), and region size in case of predictive range query,



(a) Predictive KNN Query (b) Predictive Range Query

Figure 5.4: Performance With Query Type

Figure 5.4(b), on the *predictive tree* operations. Here, we use the same setup mentioned in Section 4.4.1. We notice that the cost for the *tree construction* and *tree maintenance* does not change, while it does for the *query processing*. This concludes that the cost of the two basic operations in the *predictive tree* are isolated from the query type. The reason is that the predicted results have to be computed and saved at each node regardless the query type. In other words, the costs for constructing and maintaining the trees have to be paid anyway. The only difference happens at the *query processing* as we pay extra CPU time for processing and compiling those predicted results to match the query needs. Figure 5.4 shows a steady increasing pattern for the *query processing* with the increase of both the K values and the query sizes. Note that, in Figure 5.4(b), the *tree construction* and *tree maintenance* are not seen because their costs are minor, i.e., stationary around 3 and 0.07MS respectively, compared to the ones for the *query processing*, i.e., vary from 80 to 640MS.

Chapter 6

Conclusion and Future Directions

In this chapter, we summarize the work presented through the thesis. Then, we highlight the challenges that are still open and need more contribution from the data management research community. Finally, we present the future plan for potential research on top of the work in this thesis.

6.1 Summary

Predictive query processors answer users questions based on the anticipated future locations of a set of moving objects, (e.g., cars, bikes, pedestrians), traveling on a given space. One example for such queries includes predictive aggregate query that aims at estimating the number of objects expected to show in a specific region after a certain future time, e.g., 20 minutes. Other types includes predictive KNN, point, and range queries. The metrics to compare different query processors include: (1) Accuracy of the prediction. (2) Length of the prediction in the future time horizon. (3)Types of supported queries, e.g., predictive KNN, range, aggregate queries. (4) Efficiency and scalability with respect to objects

and queries loads. Unfortunately, existing work in predictive query processing on moving objects suffer from the following major drawbacks: (a) Most of existing work mainly target short-term prediction in terms of only few minutes and seconds in the future which is not useful for applications, e.g., traffic management systems. (b) Existing techniques can support one or at most two different types of the common predictive queries (e.g., predictive point, range, aggregate, and K-Nearest-Neighbor queries). Therefore, there is lack of general-purpose predictive queries processor. (c) Lack of extensibility, as the majority of existing systems accommodate a single prediction model, (i.e., the prediction function that predicts the future location of a given moving object). This means that if novel higher-accuracy prediction models appear, existing systems cannot be adapted smoothly to augment them. (d) The scalability issue is a significant drawback in existing predictive query processing systems, especially with the emergence of the big-data era. Due the aforementioned shortcomings, this thesis introduced innovative systems that provide support to various types of spatial predictive queries. This includes the proposed Panda system for processing spatial predictive queries on Euclidean space. In addition, it also introduced a novel index structure named Predictive Tree (P-tree) augmented inside the iRoad system to support predictive Query processing

For each framework, the thesis presented its main idea, architecture, and the embedded data structures. Three core challenges, necessary to realizing the introduced frameworks, along with the proposed approaches to solving these challenges were presented. These challenges include (1) Supporting long-term query prediction, many steps in the future, as well as short term prediction, next destination or step, (2) Scaling up to large number of moving objects, and large number of outsized predictive queries, (3) Supporting common types of predictive spatio-temporal queries including range, aggregate, and k-nearest-neighbor

queries. Experimental evidence was given to prove the scalability and efficiency of the presented frameworks. As a future work, we plan to study how to deal with uncertainty in the underlying moving objects locations and directions, and to support different prediction models within the proposed framework.

6.2 Open Challenges

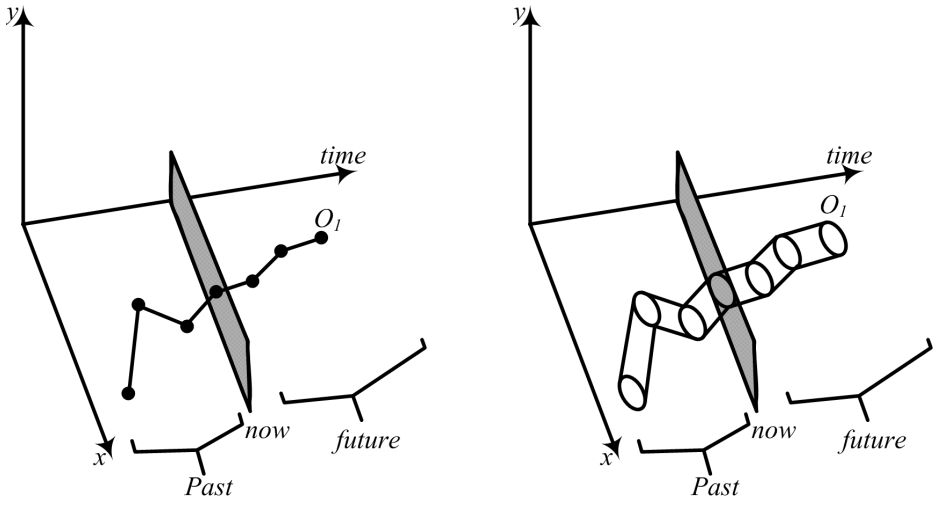
In this Section, we highlight the fundamental challenges arising from the new opportunities and the open problems. We illustrate which areas have been well investigated and which still need more digging.

Challenge 1: Uncertainty

Here, we discuss the challenging points that should be addressed while designing a model for predicting next/final destination, or complete trajectory of moving objects given imprecise spatio-temporal data.

The importance of dealing with uncertainty while processing predictive queries comes from the fact that spatio-temporal data is usually imprecise. Data uncertainty results from many sources such as the erroneous in GPS readings, accuracy limitation in measuring devices, infrequent readings due to battery shortage, communication delays and computation limitations, inexact velocity estimations, environmental obstacles such as buildings and severe weather that obstruct the communication between reading devices and satellites, and the stochastic objects motion behavior. As a result, there is a vital need to take into consideration data uncertainty while supporting predictive query processing. Basically, not considering the imprecise nature of data leads to inaccurate prediction for object future location.

Figure 6.1 illustrates the difference between defining an object trajectory using exact location points (Figure 6.1)a versus the one defined using imprecise locations (Figure 6.1b). In Figure (Figure 6.1b), object locations at different time stamps are expressed in circular regions rather than exact points. Accordingly, giving these imprecise locations to a predictive query processor as an input produces imprecise prediction for its next path.



(a) Exact Trajectory (b) Imprecise Trajectory

Figure 6.1: Impact of Uncertainty on Prediction

Little work has been done in this area. For example, in a recent work [66], a new concept based on u-bisector is used to evaluate two types of KNN queries, namely, Possible Nearest Neighbor Query (PNNQ) and Trajectory Possible Nearest Neighbor Query (TPNNQ) given imprecise object location represented as circular regions rather than exact points. However, this work does not support evaluation of predictive queries. Even for those few trials that attempt to support predictive queries, they still have accuracy, generality, and scalability issues. Consequently, it is challenging to process different kinds of predictive queries given imprecise and uncertain data about objects locations, velocities, and nondeterministic motion behaviors. Expected solutions should consider being online, which means consuming less computation time as objects dynamically change their locations and velocities.

Challenge 2: Monitoring

Tracking and monitoring moving objects is a fundamental challenge in the area of predictive spatio-temporal queries. The objective of this challenge is to capture moving objects updates that result from changing their locations, directions, and/or velocities, and efficiently handle the effect of those updates. To the best of the authors' knowledge, existing literatures do not address the moving objects monitoring and tracking problem while considering predictive queries. They only consider it for queries in the current time such as KNN and range queries [67–69]. The challenging in this point arises from the fact that many factors should be optimized. These factors not only include correctly capturing objects updates, but also reducing the number of updates to capture and send to the server, hence, reducing the communication cost between the server and the moving objects, which in turns safes the computation time required to handle the received updates. To achieve this objective, expected solution can benefit from the techniques used in the previous work which include, (a) specifying some regions in the given space in which objects movements are known to mostly affect queries results, and outside those regions updates are neglected, and (b) predicting the next trajectory of an object and report only updates if its real movement is different from the predicted one.

Challenge 3: Privacy Preserving

Although preserving users privacy while evaluating their queries is a fundamental issue for the database filed in general and for the spatio-temporal data processing in specific, but none of the existing work, to the best of authors's knowledge, discusses the privacy issue in predictive spatio-temporal query processing. Ignoring privacy means we assume that users are willing to reveal their exact trajectories data while asking for some location-based services which is not always true assumption in many applications, specially with data management

outsourcing.

Concealing moving objects' trajectories while evaluating predictive query is challenging because of the fact that there is no accurate prediction output if there is no precise history as input. So at one side, query processor needs to have an image of objects movements to be able to answer predictive queries. However, on the other side, users do not want to uncover their privacy by revealing all of their movements. In order to resolve this issue, many common techniques for dealing with privacy while answering location-based queries in the current time can be adapted for protecting privacy while responding to location-based queries for the future. Examples for these techniques include *anonymization* [70, 71] which aims to make an object's location unrecognized among K other objects, *cloaking* [72–74] which aims at expressing locations into bounding rectangles rather than exact points, *perturbation* which replaces or stuffs the real location values with synesthetic values such as noise addition [75], *cryptography* [76] which turns location readings into unreadable format to anyone except those who have the decryption key, and *transformation* [77] which protects users locations by converting the underlying space into another space.

Each of the aforementioned techniques has some merits and drawbacks. For example *cloaking* can achieve sufficient privacy while providing an accurate answer for predictive queries about tornado movements in a given city or zip code rather than an exact point. However, for predictive nearest neighbor queries, *cloaking* can not provide such accurate answer. A comprehensive study is required to assert which technique can be adjusted to support predictive spatio-temporal queries while preserving privacy of the underlying moving objects.

Challenge 4: Authentication

Authentication is a consequence issue to the age of cloud computing and data

management outsourcing. The emersion of the clouds with its offered services in affordable cost, encourages the outsourcing of data management from the data owner $\mathcal{DO}$ side to be located at the data management service provider $\mathcal{SP}$ side. With this phenomenon, user's query is received, processed and the answer is returned by the service provider $\mathcal{SP}$. With the assumption that service providers are not always trust worthy , the returned answers from their side are not trusted to be accurate. For Example, The $\mathcal{SP}$ might be hacked such that a certain instance is added to all returned results, or the $\mathcal{SP}$ itself might be not trust worthy, so it might change the results by adding, removing or modifying some parts of the returned answer.

This schema triggers the need for techniques to check the accuracy of the returned answer at the user side. Consequently, some techniques for authenticated query processing [78, 79] in which the query answer returned by the untrusted $\mathcal{SP}$ can be tested against the completeness and soundness are introduced. Completeness of answer means that no correct piece of data that should be included in the final results is disappeared from the returned answer to users. Soundness means that no modification, neither by adding non existence record nor modifying an existing one, takes place on the result [80].

Although authentication is an essential issue in the paradigm of data management outsourcing, but we did not come across any related work that addresses this issue on moving objects data neither for current time queries nor predictive queries. Even for recent work [81], it handles the authentication problem with preserving users' privacy for queries on static objects at current time instance. To the best of our knowledge, until now, all existing work in this area deals only with authentication for current time queries over stationary objects.

The big portion of those existing techniques for authenticated query processing

is based on, (a) an authenticated data structure (ADS) such as MB-tree and MH-tree [82], and MR-tree [80], which stores the outsourced data along with hash values computed for each tuple t and signed by the $\mathcal{DO}$, Figure 6.2 , and (b) the concept of verification object $\mathcal{VO}$ for carrying out the answer records with additional digest (hash) values. At the client side, this verification object is used to check the soundness and the completeness of the answer where the summation of the received hash values should be equal to the hash value of the root.

The second main technique used in the authentication existing work is signature aggregation [83, 84], in which each record in the database has a signature from the data owner $\mathcal{DO}$. The difference between the authenticated data structure technique and the signature aggregation technique is that the later guarantees higher concurrency processing for authenticated transactions, and it needs smaller number of verification objects which reduces the communication cost for sending those objects to the users. However, signature aggregation suffers from significant update overhead at the $\mathcal{DO}$ side and also costs more for verifying the received answer at the client side.

Addressing the problem of authenticated processing for predictive spatio-temporal queries is a challenging. The core difficulty of this challenge comes from the dynamic nature of moving objects updates about their locations, velocities, and directions. With each update, the data owner $\mathcal{DO}$ needs to refresh her signature on the outsourced data and resend the latest copy to the service provider $\mathcal{SP}$. Obviously, this will lead to a significant communication and computation overhead that overwhelms any gains from data outsourcing service.

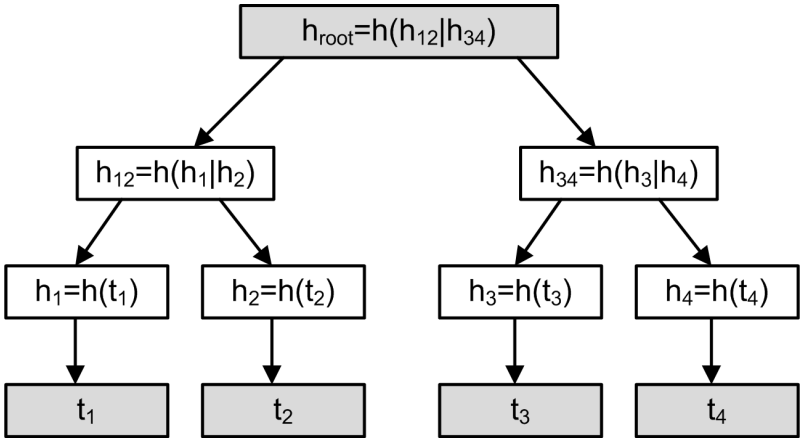


Figure 6.2: Example of Authenticated Index (MH-tree)

6.3 Future Plans

Spatial Prediction over Big-Data. Data management is evolving from two different perspectives that intersect at the point of common challenges and benefits. First is the enterprise perspective where integrating and analyzing available Big-Data sets is a serious challenge for existing data management systems. Second is the user perspective where hand-held mobile devices caused the production of massive volumes of data, e.g., trips trajectories, check-in locations, screen clicks/touches, camera usage, and queries, e.g., current/predictive kNN, range query. The future plan is to respond to this problem by building a formal bridge between the two perspectives. This can be done by designing platforms with novel data structures and query optimizers that are able to process big spatial data/queries. In specific, the target is to leverage the big-data platforms (i.e., Hadoop, Spark) to overcome the scalability challenge that most of the existing solutions suffer from.

Spatial Prediction for Forensic and Criminal Justice. Using technology for

improving the sector of forensic and criminal justice ranges from trace evidence to behavior analysis and crime scene examination. In some cases, court decisions may ask for supervising specific offenders using ankle bracelets. Each offender is given a tracking device and he is assigned to a set of allowed zones and another set of restricted zones that he has is not allowed to approach or enter, e.g., schools. In this direction, the plan is to leverage the knowledge in the area of trajectories analysis to mine for the offenders suspicious behavior and *predict probable future threats beforehand*. In addition, the future plan in this area is to design systems for criminal activities tracking and monitoring that can provide near real-time behavior analysis and alerts for possible crimes. To achieve this, there is a need to address the challenges of effectively streaming, and consuming offenders tracks on real time bases. Moreover, it is a core requirement to build novel data structures and query processors that guarantee very-low latency with highly accurate results.

Prediction of GeoSocial-Spread for Viral Events. Analysis of social networks graph leads to understanding the humans interaction behavior. In addition, it gives an image of what could be that behavior in real life, i.e., location-based interaction. This is achieved by combining the social and spatial analysis of users contributions. In this area, the aim to introduce a system that enables the analysis of users interactions in order to (a) identify viral events, (b) monitor the spread of a viral event, and (c) predict those users/places that could be affected by a viral event. The deployment of this system improves many real applications such as preventive health care.

References

[1] GO-Gulf. Smartphone Users Around the World Statistics and Facts. `http://www.go-gulf.com/blog/smartphone/`, January 2012.

[2] Yu Gu, Ge Yu, Na Guo, and Yueguo Chen. Probabilistic Moving Range Query over RFID Spatio-temporal Data Streams. In *CIKM*, pages 1413–1416, Hong Kong, China, November 2009.

[3] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD*, pages 479–490, Maryland, USA, June 2005.

[4] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE. *GeoInformatica*, 9(4):343–365, December 2005.

[5] Haojun Wang, Roger Zimmermann, and Wei-Shinn Ku. Distributed Continuous Range Query Processing on Moving Objects. In *DEXA*, pages 655–665, Krakow, Poland, September 2006.

[6] Abdeltwab M. Hendawi and Mohamed F. Mokbel. Predictive Spatio-Temporal Queries: A Comprehensive Survey and Future Directions. In *MobiGIS*, California, USA, November 2012.

[7] Hoyoung Jeung, Qing Liu, Heng Tao Shen, and Xiaofang Zhou. A Hybrid Prediction Model for Moving Objects. In *ICDE*, pages 70–79, Cancn, Mxico, April 2008.

[8] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, and Christian S. Jensen. Path Prediction and Predictive Range Querying in Road Network Databases. *VLDB Journal*, 19(4):585–602, August 2010.

[9] Abdeltawab M. Hendawi and Mohamed F. Mokbel. Panda: A Predictive Spatio-Temporal Query Processor. In *ACM SIGSPATIAL GIS*, California, USA, November 2012.

[10] Abdeltawab M. Hendawi, Jie Bao, and Mohamed F. Mokbel. iRoad: A Framework For Scalable Predictive Query Processing On Road Networks. In *VLDB*, Riva Del Garda, Italy, August 2013.

[11] Rui Zhang, H. V. Jagadish, Bing Tian Dai, and Kotagiri Ramamohanarao. Optimized Algorithms for Predictive Range and KNN Queries on Moving Objects. *Information Systems*, 35(8):911–932, December 2010.

[12] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu 0002. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *SIGMOD*, pages 611–622, Paris, France, June 2004.

[13] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest and Reverse Nearest Neighbor Queries for Moving Objects. *VLDB Journal*, 15(3):229–249, 2006.

[14] Katerina Raptopoulou, Apostolos Papadopoulos, and Yannis Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases. *GeoInformatica*, 7(2):113–137, June 2003.

[15] Yufei Tao and Dimitris Papadias. Time-parameterized Queries in Spatio-temporal Databases. In *SIGMOD*, pages 334–345, Wisconsin, USA, June 2002.

[16] Yufei Tao and Dimitris Papadias. Spatial queries in dynamic environments. *TODS*, 28(2):101–139, 2003.

[17] Tian Xia and Donghui Zhang. Continuous Reverse Nearest Neighbor Monitoring. In *ICDE*, page 77, Georgia, USA, April 2006.

[18] Jimeng Sun, Dimitris Papadias, Yufei Tao, and Bin Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal. In *ICDE*, pages 202–213, MASSACHUSETTS, USA, March 2004.

[19] James Kang, Mohamed F. Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. In *ICDE*, pages 806–815, Istanbul, Turkey, April 2007.

[20] Ken C. K. Lee, Hong Va Leong, Jing Zhou, and Antonio Si. An Efficient Algorithm for Predictive Continuous Nearest Neighbor Query Processing and Result Maintenance. In *MDM*, pages 178–182, Ayia Napa, Cyprus, May 2005.

[21] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, pages 443–454, Paris, France, June 2004.

[22] Hae Don Chon, Divyakant Agrawal, and Amr El-Abbadi. Range and kNN Query Processing for Moving Objects in Grid Model. *MONET*, 8(4):401–412, 2003.

[23] Yong-Jin Choi and Chin-Wan Chung. Selectivity Estimation for Spatio-temporal Queries to Moving Objects. In *SIGMOD*, pages 440–451, Wisconsin, USA, June 2002.

[24] Antonio Corral, Manuel Torres, Michael Vassilakopoulos, and Yannis Manolopoulos. Predictive Join Processing between Regions and Moving Objects. In *ADBIS*, volume 5207, pages 46–61, 2008.

[25] Yufei Tao, Jimeng Sun, and Dimitris Papadias. Selectivity Estimation for Predictive Spatio-Temporal Queries. In *ICDE*, pages 417–428, Bangalore, India, March 2003.

[26] Yufei Tao, Jimeng Sun, and Dimitris Papadias. Analysis of predictive spatio-temporal queries. *TODS*, 28(4):295–336, December 2003.

[27] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and Querying Moving Objects. In *ICDE*, pages 422–432, Birmingham U.K, April 1997.

[28] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, Texas, USA, May 2000.

[29] Agne Brillingaite and Christian S. Jensen. Online Route Prediction for Automotive Applications. In *ITS*, London, United Kingdom, October 2006.

[30] Jon Froehlich and Jhon Krumm. Route Prediction from Trip Observations. In *SAE*, Michigan, USA, April 2008.

[31] Hassan A. Karimi and Xiong Liu. A Predictive Location Model for Location-Based Services. In *GIS*, pages 126–133, Louisiana, USA, November 2003.

[32] John Krumm. A markov model for driver turn prediction. *SAE*, 2193(1), 2008.

[33] Sang-Wook Kim, Jung-Im Won, Jong-Dae Kim, Miyoung Shin, Junghoon Lee, and Hanil Kim. Path Prediction of Moving Objects on Road Networks Through Analyzing Past Trajectories. In *KES*, pages 379–389, Vietri sul Mare, Italy, September 2007.

[34] John Krumm. Real Time Destination Prediction Based on Efficient Routes. In *SAE*, Michigan, USA, April 2006.

[35] John Krumm and Eric Horvitz. Predestination: Inferring Destinations from Partial Trajectories. In *UbiComp*, pages 243–260, California, USA, September 2006.

[36] Meihui Zhang, Su Chen, Christian S. Jensen, Beng Chin Ooi, and Zhenjie Zhang. Effectively Indexing Uncertain Moving Objects for Predictive Queries. *PVLDB*, 2(1):1198–1209, 2009.

[37] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, New Jersey, USA, May 1990.

[38] Simonas Saltenis and Christian S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, pages 463–472, California, USA, February 2002.

[39] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. Indexing Land Surface for Efficient kNN Query. In *VLDB*, pages 1020–1031, Aucklan, New Zealand, August 2008.

[40] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, Berlin, Germany, September 2003.

[41] Charu C. Aggarwal and Dakshi Agrawal. On Nearest Neighbor Indexing of Nonlinear Trajectories. In *PODS*, pages 252–259, California, USA, June 2003.

[42] Su Chen, Beng Chin Ooi, Kian Lee Tan, and Mario A. Nascimento. ST^{2}B-tree: A Self-tunable Spatio-temporal B$^+$-tree Index for Moving Objects. In *SIGMOD*, pages 29–42, Vancouver, Canada, June 2008.

[43] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B$^+$-tree Based Indexing of Moving Objects. In *VLDB*, pages 768–779, Toronto, Canada, August 2004.

[44] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. The B^{dual}-tree: Indexing Moving Objects by Space Filling Curves in the Dual Space. *VLDB Journal*, 17(3):379–400, May 2008.

[45] Jens Dittrich Lukas Blunschi and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. *SSTD*, pages 189–207, 2009.

[46] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. MOVIES: indexing moving objects by shooting index images. *GeoInformatica*, 15(4):727–767, 2011.

[47] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE Data Engineering Bulletin*, 33(2):46–55, 2010.

[48] H. Tropf and H. Herzog. Multidimensional Range Search in Dynamically Balanced Trees. *Angewante Informatik*, 23(2):71–77, 1981.

[49] Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD*, pages 637–646, Paris, France, June 2004.

[50] Su Chen, Christian S. Jensen, and Dan Lin. A Benchmark for Evaluating Moving Object Indexes. *VLDB Journal*, 1(2):1574–1585, August 2008.

[51] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, June 2003.

[52] Shaojie Qiao, Changjie Tang, Huidong Jin, Teng Long, Shucheng Dai, Yungchang Ku, and Michael Chau. PutMode: prediction of uncertain trajectories in moving objects databases. *Applied Intelligence*, 33(3):370–386, 2010.

[53] Abdeltawab M. Hendawi. Predictive query processing on moving objects. In *In proceedings of the Data Engineering Workshops (ICDEW)*, Illinoi, USA, April 2014.

[54] Abdeltawab M. Hendawi, Mohamed Ali, and Mohamed F. Mokbel. A Framework for Spatial Predictive Query Processing and Visualization. In *MDM*, Pennsylvania, USA, June 2015.

[55] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.

[56] Abdeltawab M. Hendawi, Jie Bao, Mohamed F. Mokbel, and Mohamed Ali. Predictive Tree: An Efficient Index for Predictive Queries On Road Networks. In *ICDE*, Seoul, South Korea, April 2015.

[57] John Krumm, Robert Gruen, and Daniel Delling. From Destiantion Prediction To Route Prediction. *Technical Report. Microsoft Research*, 2011.

[58] Guttman and Antonin. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, Massachusetts, USA, June 1984.

[59] Jhon Krumm. How People Use Their Vehicles: Statistics from the 2009 National Household Travel Survey. In *SAE*, Michigan, USA, April 2008.

[60] Stanford. Large Network Dataset Collection. `http://snap.stanford.edu/data/#road`, January 2013.

[61] Dimitris Papadias, Jun Zhang, and Nikos Mamoulis. Query Processing in Spatial Network Databases. In *VLDB*, pages 802–813, Berlin, Germany, September 2003.

[62] Mohamed Ali, John Krumm, and Ankur Teredesai. ACM SIGSPATIAL GIS Cup 2012. In *ACM SIGSPATIAL GIS*, pages 597–600, California, USA, November 2012.

[63] JOSM. An extensible editor for OpenStreetMap (OSM). `http://josm.openstreetmap.de/wiki`, January 2014.

[64] Mohamed F. Mokbel, Louai Alarabi, Jie Bao, Ahmed Eldawy, Amr Magdy, Mohamed Sarwat, Ethan Waytas, and Steven Yackel. MNTG: An Extensible Web-based Traffic Generator. In *SSTD*, pages 38–55, Munich, Germany, August 2013.

[65] Abdeltawab M. Hendawi, Jie Bao, and Mohamed F. Mokbel. Predictive Tree Source Code and Sample Data. URL:http://www-users.cs.umn.edu/~hendawi/PredictiveTree/, August 2014.

[66] Xike Xie, Reynold Cheng, and Man Lung Yiu. Evaluating Trajectory Queries over Imprecise Location Data. In *SSDBM*, pages 56–74, Chania Crete, Greece, June 2012.

[67] Tian Li, Wang Le, Zou Peng, Jia Yan, and Li Aiping. Continuous monitoring of skyline query over highly dynamic moving objects. In *MobiDE*, pages 59–66, Beijing, China, June 2007.

[68] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, pages 634–645, Maryland, USA, June 2005.

[69] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, pages 631–642, Tokyo, Japan, April 2005.

[70] Osman Abul, Francesco Bonchi, and Mirco Nanni. Never walk alone: Uncertainty for anonymity in moving objects databases. In *ICDE*, pages 376–385, Cancun, Mexico, April 2008.

[71] Hidetoshi Kido, Yutaka Yanagisawa, and Tetsuji Satoh. Protection of location privacy using dummies for location-based services. In *ICDE*, pages 1248–1248, Tokyo, Japan, April 2005.

[72] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, pages 31–42, California, USA, May 2003.

[73] Panos Kalnis, Gabriel Ghinita, Kyriakos Mouratidis, and Dimitris Papadias. Preventing location-based identity inference in anonymous spatial queries. *TKDE*, 19(12):1719–1733, 2007.

[74] Mohamed F. Mokbel, Chi-Yin Chow, and Walid G. Aref. The new Casper: query processing for location services without compromising privacy. In *VLDB*, pages 763–774, Seoul, Korea, September 2006.

[75] Jhon Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399, 2009.

[76] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: anonymizers are not necessary. In *SIGMOD*, pages 121–132, Vancouver, Canada, June 2008.

[77] Ali Khoshgozaran and Cyrus Shahabi. Private Information Retrieval Techniques for Enabling Location Privacy in Location-Based Services. In *SSTD*, pages 59–83, Aalborg, Denmark, July 2009.

[78] Weiwei Cheng and Kian-Lee Tan. Query assurance verification for outsourced multi-dimensional databases. *Journal of Computer Security*, 17(1):101–126, 2009.

[79] Man Lung Yiu, Eric Lo, and Duncan Yung. Authentication of moving kNN queries. In *ICDE*, pages 565–576, Hannover, Germany, April 2011.

[80] Stavros Papadopoulos, Yin Yang, and Dimitris Papadias. Continuous authentication on relational streams. *VLDB Journal*, 19(2):161–180, 2010.

[81] Haibo Hu, Jianliang Xu, Qian Chen, and Ziwei Yang. Authenticating location-based services without compromising location privacy. In *SIGMOD*, pages 301–312, Arizona, USA, May 2012.

[82] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, Illinois, USA, June 2006.

[83] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *SIGMOD*, pages 407–418, Maryland, USA, June 2005.

[84] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable Verification for Outsourced Dynamic Databases. *PVLDB*, 2(1):802–813, 2009.