Spring 2015

Enabling near-term prediction of status for intelligent transportation systems: Management techniques for data on mobile objects

Lasanthi Nilmini Heendaliya

Department: Computer Science

ENABLING NEAR-TERM PREDICTION OF STATUS FOR INTELLIGENT TRANSPORTATION SYSTEMS: MANAGEMENT TECHNIQUES FOR DATA ON MOBILE OBJECTS

by

LASANTHI NILMINI HEENDALIYA

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2015

Approved by

Dr. Ali Hurson, Advisor
Dr. Dan Lin, co-advisor
Dr. Bruce McMillin
Dr. Sanjay Madria
Dr. Sahra Sedighsarvestani

ABSTRACT

Location Dependent Queries (LDQs) benefit from the rapid advances in communication and Global Positioning System (GPS) technologies to track moving objects' locations, and improve the quality-of-life by providing location relevant services and information to end users. The enormity of the underlying data maintained by LDQ applications – a large quantity of mobile objects and their frequent mobility – is, however, a major obstacle in providing effective and efficient services. *Motivated by this obstacle*, this thesis sets out in the quest to find improved methods to efficiently index, access, retrieve, and update volatile LDQ related mobile object data and information. Challenges and research issues are discussed in detail, and solutions are presented and examined.

ACKNOWLEDGEMENT

My first and foremost gratitude goes out to my advisor, Dr. Hurson, for accepting me as his student, for giving me the opportunity to work with him, for guiding and assisting me become better at learning. Thank you! for being extremely patient and continuously encouraging me even when my productivity was low during certain testing times of my life.

Next, I like to express my gratitude to my co-advisor, Dr. Lin. Her support, guidance, and resources on my research have been priceless. You encouraged me and lifted me up when I had little strength and spirit. I would also like to thank my other committee members: Dr. Bruce McMillin, Dr. Sanjay Madria, and Dr. Sahra Sedigh for serving as my committee members. Their feedback, comments, and suggestions were always helpful.

This journey wouldn't have been possible if not for all those who financially supported me along the way. Missouri S&T CS department gave me the opportunity to expand my experience as a teaching assistance. Drs. Hurson, Lin, McMillin, and Sedigh graciously provided me with graduate assistantships. I was also lucky to receive the Philanthropic Educational Organization (P.E.O.) International Peace Scholarship, for most part due to Ms. Christena Sowers introducing me to the opportunity and helping me with the application process. The scholarship not only supported me financially, but also introduced me to family like friends: Ms. Debbie Estey, Ms. Marge Pundman, Ms. Noel Berryman, and many more unnamed P.E.O. members.

I was also fortunate to work with and have support from *Mike*, Michael Wisely. His great talents and experience in programming made it easier to convert my research ideas into implementations. I would also like express my appreciation to the Missouri S&T Computer Science Department staff members for their kind cooperation and excellent support every time I needed their assistance.

Above all, I am eternally gratefully to my family: my parents, my brothers, my husband, and sons. My mother, for her unconditional, limitless love and dedication to bring me to the position where I am now; My father, for his protection and giving me the freedom to be me; My brothers for their love and countless childhood memories; my husband for believing in me even when I myself, had no confidence in me; and lastly my beloved sons - *Evin* and *Javin* - for unknowingly agreeing to exchange their precious *mommy time* for "*mommy's Ph.D. time*".

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACRONYMS

ANR-tree	Adaptive Network R-tree ... 24
CPLQ	Continuous Predictive Line Query ... 4
CPS	Cyber Physical System ... 1
DIME	Disposable Index for Moving objects ... 27
GNNQ	Grouped Nearest Neighbor Query ... 7
GPS	Global Positioning System ... 5
GTR	Group Update Time Parameter R-tree ... 26
GTS	Ground Transportation System ... 2
IMORS	Indexing Moving Objects on Road Sectors ... 24
INOR-tree	Intersection Oriented Network R-tree ... 23
IR	Influence Region ... 70
KNNQ	K-Nearest Neighbor Query ... 6
LAQ	Location Aware Query ... 1
LDQ	Location Dependent Query ... 1
MBR	Minimum Bounding Rectangle ... 11
MOVNet	MOVing Objects in Road Networks ... 26
MQM	Monitoring Query Management ... 30
PDQ	Predictive Density Query ... 105
PLQ	Predictive Line Query ... 3
QI	Query Indexing ... 30
RGTR	Robust Group Update Time Parameter R-tree ... 26
RKNNQ	Reverse K Nearest Neighbor Query ... 6
RQ	Range Query ... 6
R^0Q	Ring Query ... 49
RUM-tree	R-tree with Update Memo ... 15
SD-tree	Shortest Distance-based Tree ... 32
SPLQ	Snapshot Predictive Line Query ... 4
SQM	Spatial Query Management ... 30
STRIPES	Scalable Trajectory Index for Predicted Positions in Moving Object Databases ... 17
TPR-tree	Time Parameterized R-tree ... 12
TPR*-tree	Time Parameterized R*-tree ... 13
TPR^Q-tree	Time-Parameterized Query R*-tree ... 67

1. INTRODUCTION

At least 35% of the working population in the United States currently commutes more than 30 minutes each day [1]. Time wasted on the road affects both individuals and society. For instance, individual who spends more time on the road will spend more money on fuel. In 2013, Americans wasted $124 billion due to traffic congestions. Without significant action to alleviate congestion, this cost is expected to reach $186 billion by 2030 [2]. Higher commuter times also increases the changes of encountering accidents.

Society demands information that helps optimize travel time and hence enhances the travel experience (e.g., real-time traffic information, shortest detoured path, closest hospital, or nearest gas station). These services, which address both individual's and societal needs, have escalated through technological innovation. According to the Statistics Portal, the estimated market size of stand-alone Global Positioning Systems (GPSs) in 2015 will exceed 35 million units [3]. This estimate did not include either GPS units embedded in vehicles or those available in mobile devices (e.g., smart phones and iPads®).

With the aid of this new technology, ground transportation is anticipated to become a Cyber Physical System (CPS). This CPS will be comprised of both a cyber infrastructure (computers, communication links, and sensors) and a physical infrastructure (roads and vehicles). Cyber infrastructure monitors, provides decision support to and controls the physical infrastructure. A significant fraction of CPSs (the modern counterparts of traditional physical infrastructure systems) includes components capable of both intelligent communication and effective control. Added intelligence in the form of sensors, embedded systems (either short- or long-range transceivers), and other computing and communication resources promises invasive operation flow, more robust infrastructures, increased autonomy, and improved safety.

Vehicles request information from a CPS through query requests. The queries addressing position-related information (e.g., find the best 5 restaurants in Chicago or find the 5 nearest restaurants to my current location) are known as *Location Aware Queries (LAQs)*. *Location Dependent Queries (LDQs)* are a subclass of LAQs. There is a subtle yet fundamental difference between these two queries. The result set of the first query is independent of the user's location and will be the same for

any query issuer, regardless of his/her location. For the second query, however, the result(s) will vary according to the user's geographic position.

Because LDQs are highly dependent on geographical location, efficient processing becomes quite challenging. It becomes even more challenging when the queried objects are moving (e.g., find the 5 taxis closest to my current location). This work is intended to address these challenges related to queries on objects moving within the ground transportation infrastructure.

1.1. QUERY PROCESSING UNDER ROAD NETWORK CONSTRAINTS

1.2. MOTIVATION

The Ground Transportation System (GTS) is considerably more complex than its counterpart in other CPS domains (e.g., power grids and water distribution systems). Contributing factors to this increase complexity includes:

- both the size and volatility of the underlying databases;
- the number of entities involved; and
- the human entities taking part in the process.

These characteristics are manifested in solutions that address both the frequent update of and the efficient access to large data repositories and technical constraints of the cyber infrastructure. Efficient access to frequently updated data is primarily addressed by means of efficient indexing and querying techniques. Both the efficient adaptation of previous techniques (when possible) and/or the introduction of new techniques is necessary to alleviate the GTS's added complexity.

Most work on mobile object indexing relax objects' movement to the Euclidean space [4–12]. Only a handful of work have considered road network constraints [13, 14], which is the more realistic modeling and implementation path to moving object-related applications; solutions available based on Euclidean space are incapable of providing accurate query results under added road-network constraints [15–17].

New query types could also possibly be introduced to address consumer demand for location based services. One such important query type is the *predictive queries*. By means of predictive queries, based on road conditions and route redirection, commuters have the ability to take proactive steps to make their travel

time more efficient and safer. For example, commuters may occupy the time with secondary tasks such as fueling and having lunch until a road condition, such as a traffic congestion, gets alleviated. Existing LDQs, however, do not support predictive queries under the road network constraints [18, 19]. This work intends to develop a new indexing scheme supporting predictive as well as current queries on objects moving under the road network constraints, and define and design predictive query algorithms.

1.3. OBJECTIVES

The following objectives are assembled to address both the aforementioned challenges as well as the foreseeable service requirements:

Develop a mobile data indexing structure under road-network constraints An indexing structure is developed to manage the ever-growing data associated with a transportation environment. The proposed indexing structure will support LDQs – both current and predictive.

Design and implement on-demand-based predictive queries A predictive query type – termed a *Predictive Line Query (PLQ)* is developed to exhibit the performance of the proposed indexing structure. Two types of queries are considered: snapshot query and continuous query.

Design and implement proactive-based predictive queries Research on PLQ is extended to both develop and accommodate the density query. The query algorithm will address issues that arise from considering road network constraints for PLQs.

The work proposed here provides formal definitions for the aforementioned query types under road network constrains. It is an attempt to bridge the gap in converting the ground transportation system into a CPS, allowing users to experience economically efficient and safe traveling.

The remainder of this thesis is organized as follows:

- Chapter 2 provides background information on the mobile object environment, Location Dependent Queries (LDQs) and some performance metrics which are being used to compare different approaches.
- Chapter 3 discusses past research on mobile data indexing schemes and querying techniques. The discussion on indexing schemes summarizes and

highlights strengths and drawbacks of existing approaches designed for Euclidean space and road network. The discussion on querying techniques focused on continuous query processing of LDQ and density queries.

- Chapter 4 first discusses the problem of mobile object indexing specific to the road network constraints. Then it discusses the design issues and implementation of the proposed indexing structure followed by its maintenance and query algorithms;

- Chapter 5 introduces a novel query type - the *Snapshot Predictive Line Query (SPLQ)* with three efficient query algorithms. The performance of the proposed algorithms is also presented in this chapter.

- Chapter 6 addresses the issues available with Snapshot Predictive Line Query (SPLQ) by introducing the *Continuous Predictive Line Query (CPLQ)*. It also, accompanies with three query algorithms and an extensive performance study.

- Chapter 7 redefines the density query to fit the needs of objects traveling under the road network constraints. The chapter discussion then presents an efficient query processing algorithm for the proposed query.

- Chapter 8 concludes the presented work and discusses future directions.

2. PRELIMINARIES

This chapter provides the preliminaries on both the mobile object environment and *Location Dependent Queries (LDQs)*. The background starts with introducing the moving object infrastructure in Chapter 2.1. A short introduction to LDQs supported in the moving object environment can be found in Chapter 2.2. Chapter 2.3 discusses some common performance metrics, which allow comparing similar approaches with each other.

2.1. MOBILE OBJECT APPLICATIONS INFRASTRUCTURE

The underlying infrastructure of moving object applications is composed of several components: *Moving Objects*, *Static Objects*, *Base Stations*, and, in most cases, the road network as shown in Figure 2.1 [20, 21].

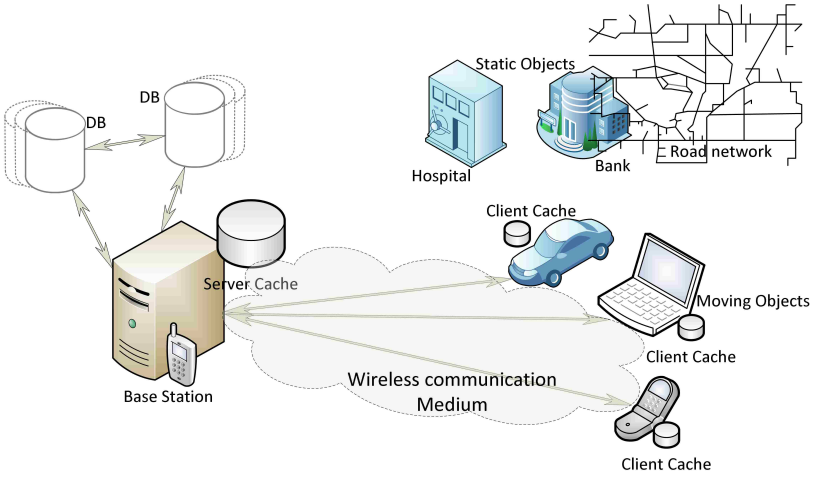


Figure 2.1 Moving Object Infrastructure

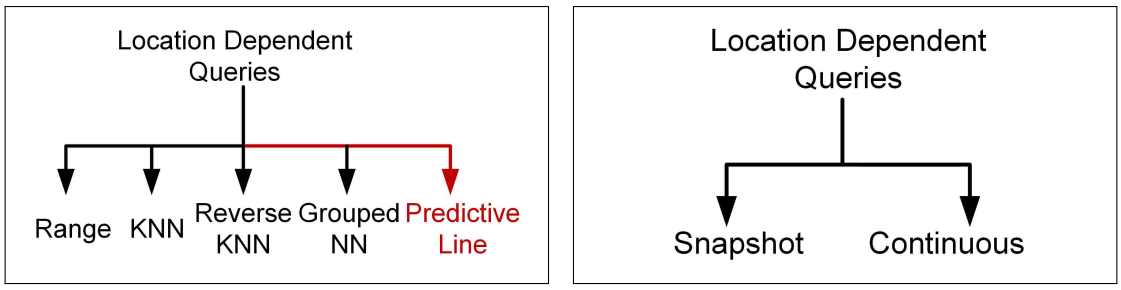
Moving objects are objects with a mobility capability. They are often referred to as either *Mobile Hosts* [22, 23] or *Mobile Clients* [4, 11]. Moving objects move on a fixed network bound to velocity constraints. the Global Positioning System (GPS) built-in to a moving vehicle or a cell phone carried by a person are tangible examples of moving objects. These objects are equipped with some type of GPS device to not only sense but also to measure both the speed and position of the vehicles they track. The devices may utilize a cache to retain recently used information for future use.

Static objects are geo-stationary objects with fixed position coordinates (e.g., buildings). Information about both moving objects and static objects are stored at base stations in reference to the road network. These base stations fulfill the clients requests based on the stored data. Most base stations maintain recently queried information in a cache to reduce the response time and/or the server side workload. Communication between base stations and moving objects occurs through wireless communication medium. The information exchanged in the communication includes query requests, query responses, and moving object information sensed by either the moving objects' GPS or road side sensors.

In a typical application, the user sends the query request to the base station, the base station retrieves the relevant information from the collection of databases, and the results are sent back to the user. Intermediate *Message Support Stations* located between base stations and moving objects are used to improve the strength of communication signals.

2.2. LOCATION DEPENDENT QUERY TAXONOMY

Location Dependent Queries (LDQs) are critical to the proper functionality of the mobile object applications infrastructure. They can be classified into several sub query types according to the information it provides (information based taxonomy as shown in Figure 2.2(a)) or according to their response frequency (frequency-based taxonomy as shown in Figure 2.2(b)).



(a) Information-based LDQ Taxonomy (b) Frequency-based LDQ Taxonomy

Figure 2.2 Location Dependent Query Taxonomy

Some common types of queries under the information based query taxonomy include *Range Query (RQ)*, *K-Nearest Neighbor Query (KNNQ)*, *Reverse K Nearest*

Neighbor Query (RKNNQ), and *Grouped Nearest Neighbor Query (GNNQ)*. The proposed *Predictive Line Queries (PLQs)* (in Red in Figure 2.2(a)) are also categorized under information-based LDQs.

RQs [5, 9, 24] search all objects within a user specified range (e.g., Show me all taxis within a 10 miles range from my current location). KNNQs [10–12, 24, 25] provides the K nearest neighboring objects to the issuer (e.g., Show me the 5 closest taxis to my current location). The information provided by a RQ is similar to that of a KNNQ. The difference comes from both the number of objects in the result set and the area that the data is looked for. The RQ restricts the range but relaxes the number of objects the issuer is interested in. The KNNQ is the opposite; it restricts the number of objects but relaxes the range.

Similar to the KNNQ, the RKNNQ [26] also specifies the number of objects it is looking for. However, the answers to KNNQ and RKNNQ are not necessarily the same because KNNQ is from the issuers perspective while RKNNQ is from the objects perspective (e.g., Show me 5 pedestrians whose nearest taxi driver is me). In both the example for KNNQ and RKNNQ, the taxi driver wants to know about five pedestrians. In the KNNQ example, those five pedestrians are the closest to the issuer. No other pedestrians are closer, but these pedestrians might find some other taxi driver closer to them. KNNQ does not consider the pedestrians view of the taxi drivers. The RKNNQ, on the other hand, queries for pedestrians who find the issuer within their five closest taxi drivers.

The GNNQ [27] provides a grouped answer to a set of KNNQs (e.g., the best place for n people from different companies/organizations to meet). The GNNQ provides an aggregated answer to a set of KNNQ. For example, consider people in `"n"` companies/organizations who are trying to schedule a meeting. The best place for everyone to meet would be at a site that reduces the total travel time. A GNNQ would find a solution to this kind of situation. Thus, the GNNQ response reduces the cost metric (the total travel time in the previous example) collectively to provide an aggregated answer.

Each aforementioned query type can be associated with a time parameter. In such cases, the queries aim to predict object positions at a specific future timestamp, e.g., some RQ extensions include predictive time slice queries, window queries, and moving queries. The predictive time slice query (also known as either a future RQ or a predictive RQ) finds all of the moving objects that will be inside the query range during the specified future time period between two given timestamps.

The window query is a generalized form of the time-slice query whose timestamps coincide. The moving query is a further generalization of a window query. The moving query specifies two ranges at two different timestamps: the initial time and the end time. These two ranges could be different in size and/or location. The range at the initial time gradually evolves into the range at the end time. In general, these two ranges can be considered to be one dynamically-shaped moving range. The query answer contains all of the moving objects that cross the moving range.

Categories of queries under the frequency-based query taxonomy include both snapshot queries and continuous queries. If the query result is provided only once per request then the query is identified as a *snapshot query*. This query expires when a result is produced. As the name implies, a *continuous query* processes a request continuously, informing the user of changes in the result set. Continuous queries do not expire with the first response. Instead, requesters may revoke their request when they are no longer interested in the service for that particular query.

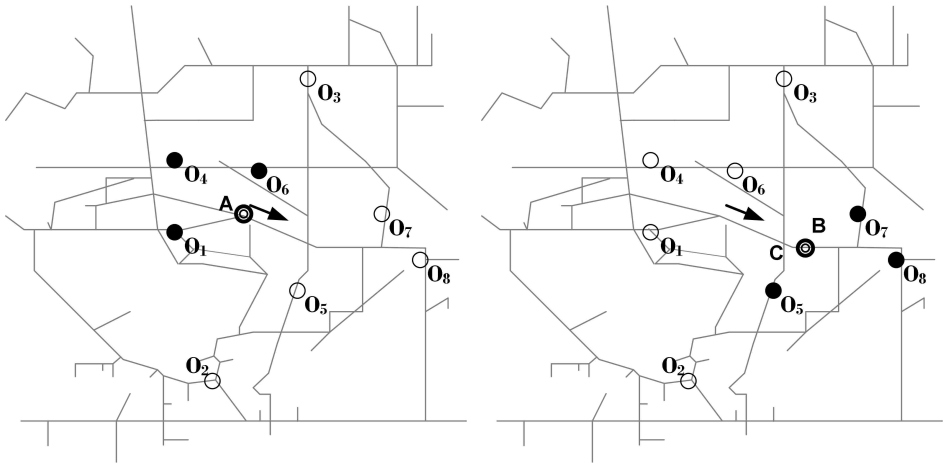
The sub categories of these two taxonomies can be co-related as well, e.g., snapshot RQs, continuous RQs, snapshot KNNQs, continuous KNNQs and so on. The most common and default frequency-based query category for information-based query types discussed in literature is the snapshot query category.

2.3. PERFORMANCE METRICS

Relevancy, response time, and information privacy are the primary features a user expects from an LDQ service provider. These are also the performance metrics used to determine the overall effectiveness of a moving object infrastructure. This research focuses on issues related to response time and accuracy; information privacy is beyond the scope of this work.

As a precursor the discussion on query performance issues, consider the scenario illustrated in Figure 2.3. Figures 2.3(a) and 2.3(b) show the results for the same KNNQ but at two different timestamps – $\{t_0$ and $t_1\}$. The positions of the issuer at t_0 and t_1 are A and B, respectively. The query requests the three objects that are nearest to the query issuer. Filled circles represent the query results.

Consider the query issued at point A. Due to processing delay, the corresponding response $\{O_6, O_1,$ and $O_4\}$, is received when the issuer is at point B. At this point, the result is no longer relevant to the user as the current position is not what the query was processed for. The service provider might be intelligent enough to predict the user's future position at time t_1. In such a case, it might be



(a) Query Issued at Point A (b) Answer Received at Point B

Figure 2.3 An Example of Location Dependency of an KNNQ Result

able to generate the results for the issuer at Point B accordingly: $\{O_5, O_7, \text{and } O_8\}$. The result still may not be valid for the issuer **if** the road network constraints have not been considered in the process. For instance, reaching the object O_5 from point B may not be practical under road network constraints such as one-way roads are taken into the consideration.

The irrelevancy of a response is primarily due to the object's mobility characteristic; the information at the base station database, including both mobile object locations and the moving speed, become outdated quickly and frequently due to object's mobility. Such outdated information about mobile objects will result in obsolete and/or erroneous results. When accurate information is unavailable just-in-time, the user will often resort to resending the query request to the server. Both of these repetitive requests and their corresponding responses result in an inefficient use of communication, computational resources, and power resources. To provide the user with a relevant answer, the query execution should consider up-to-date information from both the query issuer and all other mobile objects.

Each mobile object periodically sends messages to update its up-to-date information. Assuming an update interval of 120 seconds, a system with 1 million mobile objects generates 30 million update messages per hour [10]. Such enormous amounts of messages can cause several additional performance issues to arise. These issues include inability to handle update messages quickly to provide up-to-date information for the next query request and to avoid communication overhead

from massive amounts of update request. In essence, such issues could disrupt service availability for current and future query requesters.

Addressing these issues require the development of efficient and effective update message handling techniques. Providing a fast response from up-to-date information is a matter of retrieving relevant information efficiently. A sequential search is an inefficient searching technique for a massive data collection. In a worst case scenario, all of the items in the data collection are accessed. This type of search is good for accessing data randomly. It is, however, inappropriate for spatially related data (as in mobile object related services). Thus, the data collection should be organized and stored in a manner that provides efficient access to relevant data.

Considering the aforementioned issues, the performance of a LDQ service should be evaluated based on how efficiently the information can be updated, relevant data items can be retrieved, and hardware can be utilized in providing accurate query response. Thus, update costs, search costs, and storage costs have become important performance metrics to analyze the different solutions and to compare different indexing and query types.

3. LITERATURE SURVEY

This chapter surveys three areas related to mobile data: mobile objects indexing structures, continuous query processing, and density queries. The following discussion emphasizes the strength and drawbacks of various approaches under each area. Indexing techniques in moving object databases are reviewed under two main categories based on how object movement is modeled. These are, indexing objects moving on Euclidean space and indexing objects moving under road network constraints. The discussion on continuous query processing is on query processing techniques found in current literature. Lastly, issues related to processing density queries are addressed.

3.1. INDEXING MOVING OBJECTS ON EUCLIDEAN SPACE

Much work has been done on indexing moving objects on Euclidean space [5–8]. Most of these approaches model the mobility of objects as a linear function of time, where the position at any given time t, denoted as $x(t)$, is defined as in Equation (1).

$$x(t) = x_{ref} + v(t - t_{ref}) \tag{1}$$

Here, x_{ref} represents a reference position at time t_{ref} ($t_{ref} < t$). The velocity vector is represented by v. The linear representation of moving objects makes predicting the object's future position simple and calculating it in constant time. Hence, fewer update messages are needed from mobile objects. Fewer update messages, while reduces the update cost, results in reduced accuracy in objects' actual location.

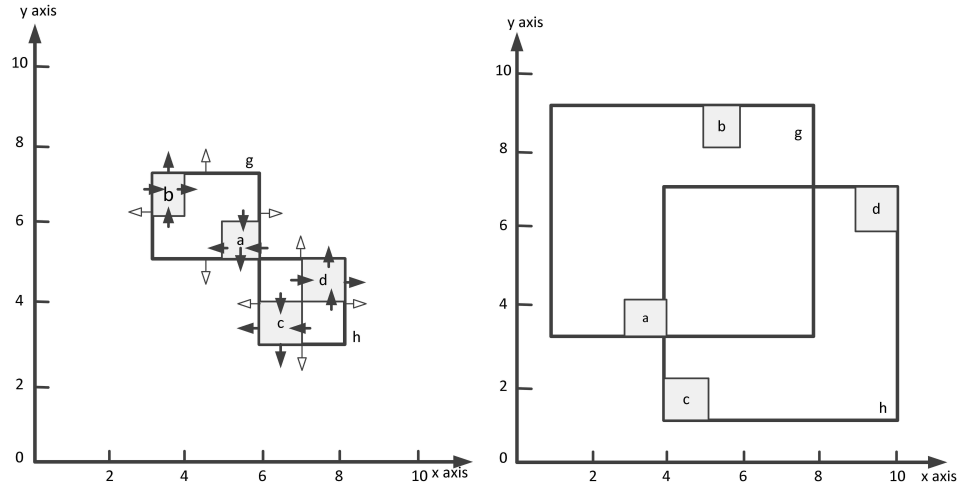
The indexing structures that utilize linear representation can be classified into several classes depending on their base indexing structure. Some commonly used base structures include the R-tree, the B^+-tree, and the quadtree. In addition to these structures, some hybrid approaches can also be found in the literature [28].

3.1.1. R-tree-based Indexing Structures. R-tree is a height balance tree structure. Each R-tree leaf node maintains mobile objects' attributes. Each object is represented by a tuple `(id, MBR)`. The `id` is the identifier for the object, and the *Minimum Bounding Rectangle (MBR)* is a rectangle which tightly bounds the object.

A non-leaf node of the R-tree maintains all of its children nodes' MBR along with a pointer to each child node.

The *Time Parameterized R-tree (TPR-tree)* [5] extends an R*-tree by maintaning mobile objects' velocity information. Each side of the MBR is embedded with a velocity component. The MBR's dimensions are updated according to these velocities and the mobile objects remain in the updated MBR. The velocities attached to opposite sides of the MBR represent the minimum and the maximum velocities along the two-dimensional Cartesian space along the object's moving direction; one pair of sides represents the x direction while the other pair represents the y direction.

This velocity information propagates to every MBR up to the tree root. For example, consider the time parameterized MBR depicted in Figure 3.1. The gray-colored rectangles in Figure 3.1(a) represent the leaf-node MBRs (a, b, c, and d) and the white-colored rectangles represent their parent MBRs (g and h). Leaf node and Parent MBRs' directions are shown by the filled-arrow attached to each MBR side and bt the hollow-arrow heads, respectively. The velocity magnitudes of each leaf-node's MBR side is 1. Depending on these leaf-nodes' velocities (+1s and -1s) and their parent MBR's, each side's speed becomes 1 as well. Figures 3.1(a) and 3.1(b) show the MBRs at time t_0 (construction time) and time t_1, respectively.



(a) MBR Orientation at Time t_0: Tight and No Overlap

(b) MBR Orientation at Time $t_1 > t_0$: Loose and Overlapped

Figure 3.1 Issues in Time Parameterized MBRs

A primary drawback of this approach is the unconditional expansion of MBRs as time elapses. As a result, the bounding rectangle may no longer be

"*minimum*". Additionally, an extensive amount of overlapping MBRs can also appear. Consider Figure 3.1(a) that shows both leaf node MBRs and parent MBRs at construction time. As time evolves, these non-overlapping MBR start to extensively expand and overlap, as shown in Figure 3.1(b). Consequently, such overlapped MBRs increase the search cost [5, 6].

Another TPR-tree specific drawback stems from its maintenance function (e.g. insertion and deletion) design. These functions are a direct, simple modification of an R*-tree developed for storing static data. For this reason, they maybe unable to render expected performances for dynamic data indexing in a Time Parameterized R-tree. The aforementioned behavior mismatch between the indexed objects (mobile) and the supported functions (static) is addressed in the *Time Parameterized R*-tree (TPR*-tree)* [6]. To overcome the mismatch, the TPR*-tree improves the insertion and the deletion functions of theTPR-tree.

As for the TPR-tree insertion function, when a new node is inserted into the TPR*-tree, the tree is traversed from the root to the best leaf node that can facilitate the new entry. The sub-tree is selected according to the lowest value of the predefined penalty metrics (e.g., both the perimeter and the overlapping area). When two or more sub-trees produce the same penalty value, each such sub-tree is explored further, until one selection becomes dominant[1]. This expanded search within a TPR-tree, however, can result in additional search costs. Research has shown that the overall overhead associated with obtaining the complete access path (using the proposed approach) is very minor [6].

If the selected node is already full, some of its entries, those that contribute to form MBR margins, will be removed. After these entries are removed, the node's MBR shrinks which produces a tighter MBR; but tree structure will remain unchanged. Once the MBR is tighten, reinserting a once removed entry into the same node might produce higher penalty than some other node. As a result of this higher penalty, new tree paths can emerge that lead to a different node than the one that the entry was originally placed. Any overflowed nodes found in the reinsertion process will be split. Each of these optimizations has lead the TPR*-tree to outperform the TPR-tree in the query performance [6] and to offset the search cost overhead mentioned in the previous paragraph.

However, both the TPR-tree and the TPR*-tree suffer from the unconditional expansion of MBRs. This issue has been addressed in several past studies [7, 8]. In 2002, Papadopoulos et al. [7] proposed handling moving objects separately

[1]In tiebreaker situations, every tied sub-tree is further explored until the best node is reached

according to their speeds. The approach proposed in [7] reduces the possibility of having large MBRs by handling moving objects in separate structures according to their speed. Furthermore, their work handled each dimension of the moving space separately; a separate R-tree was maintained for each dimension-speed category. As a result, this indexing structure must maintain several R-tree structures resulting in poor space utilization. More specifically, the space consumption of this approach was almost twice as that of a TPR-tree [7]. Determining the proper speed limit was also challenging in this approach.

Saltenis and Jensen [8] introduced an expiration time for the function parameters. The indexing technique they proposed is known as the R^{EXP}-tree. This tree embeds an expiration time for the function parameters. Thus, the MBR construction considers the object's life up to the expiration time. The corresponding MBR construction considers this short-term life result in four possible implementations: always-minimum, conservative, static, and update-minimum.

The always minimum-strategy ensures that the MBRs are always, i.e., not only at the construction time, but also during the object's entire life-time, tightly bound to enclosed objects. This way, the R^{EXP}-tree ensures to consider new information (e.g., a changes in both speed and direction) when updating the MBRs, each time that the object's information deviates from the reported information. Considering all of these changes during the object's lifetime is both difficult and impractical to implement; the entire future trajectories of each object for must be considered.

In conservative MBR construction, a perfect MBR is guaranteed only at construction time; it is not guaranteed subsequently. Static MBR construction defines MBR boundaries by considering both the lower and the upper position limits of the objects for the specified time. An update-minimum MBR is an improved version of the conservative MBR approach. At each update, the update-minimum approach reconstructed MBR in such a way that velocities of the bounding rectangle covers higher speed objects up to their expiration time. Regardless of the adoption method, all four MBR construction methods exhibit nearly the same performance characteristics [8].

None of the aforementioned MBR construction approaches remove objects as soon as they expire. This is because the removal requires the tree to be restructured, causing tree maintenance overhead. Instead, expired objects are removed each time this information is written back from the memory to the secondary disk. This could result in nodes being underutilized, as the node capacity is shared by both

expired and live entries. Delayed data removal lowers the tree maintenance cost (as the tree does not need to be restructured for each update), at the expense of lower space utilization.

The *R-tree with Update Memo (RUM-tree)* [4] also addresses the restructuring overhead of a tree structure. It handles the update message as an insertion followed by a deletion. The insertion is performed promptly upon receiving the update message. Deletion, however, is delayed, which results in multiple versions of one particular object. These unnecessary, older versions are removed by a process known as garbage cleaning. Garbage cleaning is activated when the RUM-tree's memory usage is going to overflow. One advantage of these approaches is the transparency of the expired object removal.

In another approach to reduce tree restructuring overhead, Dongseop et al. [25] proposed an *LUR-tree* that reviews the possibility of accommodating the new position of a moving object within the current MBR. It does so without following the typical update technique (deletion followed by an insertion), which can create unnecessary partitioning, Additionally, this method does not require any restructuring of the MBR if the object's new position falls within the current MBR.

If the new object's position falls outside the current MBR, one of the three following methods is proposed: traditional deletion and insertion, extension of the MBR, or reinsertion into the parent node. The first method (traditional deletion and insertion) is trivial. The second method (the extension of the MBR) maintains a slightly larger MBR. This expansion is more appropriate for situations in which objects move along the boundary roads in a zig-zag motion. In the reinsertion method, the updated object is inserted into the parent node. If the immediate parent node is full, the process propagates up to the root until a suitable candidate is found.

3.1.2. B^+-tree-based Indexing Structures. In general, B^+-tree based indexing structures convert an object's two-dimensional position to a one-dimensional position. The conversion is performed by means of a space filling curve (e.g. either the Hilbert or the Piano Curve). In this conversion process, the given space is considered as a two-dimensional grid. Every cell in the grid is visited only once. Each cell is then assigned a sequence value. This sequence value is the key used when indexing the objects in a cell in the B^+-tree.

Figure 3.2 illustrates the assignment of sequence values based on the Piano space filling curve. The number in each cell represents the sequence number of that particular cell. The indexing structures proposed in [10–12] have employed both B^+-trees and the aforementioned space filling concept. Technically, these structures are comprised of several B^+-trees. One tree is used to maintain the object information within one update interval, while another B^+-tree is used for the subsequent update interval. Maintaining separate B^+-trees provides the index structure with adequate time to clean up all of the object information at the first update interval. The second tree allows for handling the messages within a succeeding update interval. These B^+-trees are then used interchangeably.

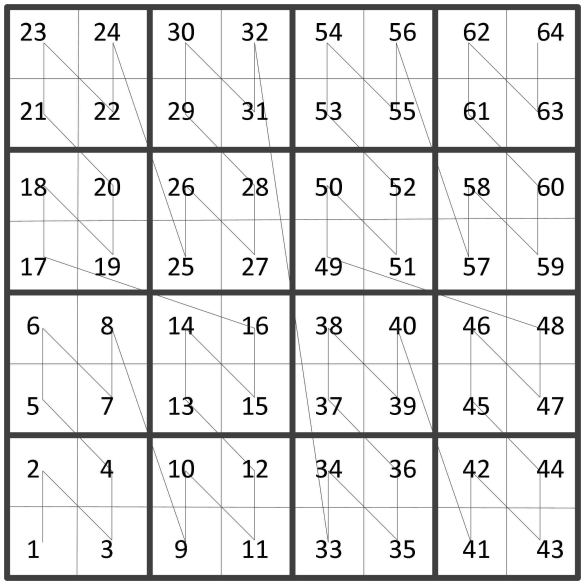


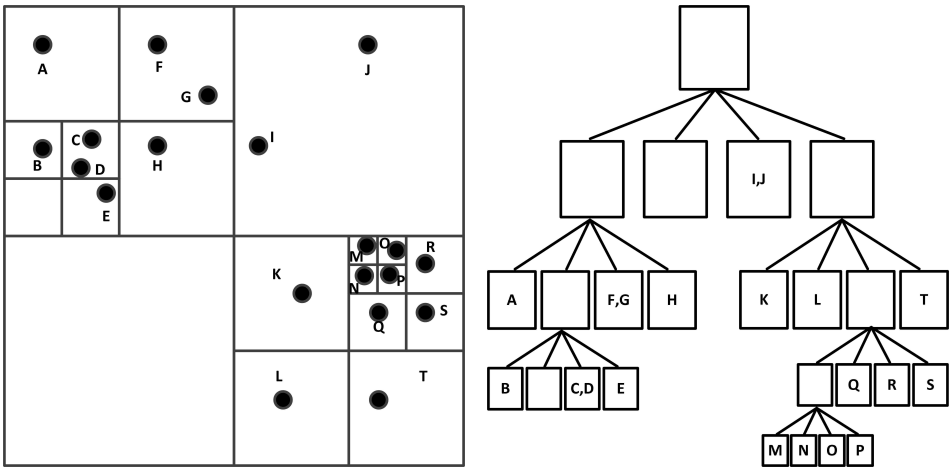
Figure 3.2 An Example of Space Filling Curve: Piano Curve

The B^x-tree [10] considers the global maximum speed when handling the objects' speeds. This global speed consideration demotes the performance of the B^x-tree, as query results would contain many false positives. The B^{dual} -tree [12] addresses this issue, by considering both an object's location and speed using a four-dimensional space-filling curve.

Both the B^x-tree and the B^{dual}-tree consider a normal distribution of objects on the Euclidean space. They are unable to perform well on skewed data distribution. Later studies tried to overcome the drawback of B^x-tree sensitiveness on skewed data. The ST^{2}B-tree improves the B^x-tree index to support skewed object

distribution [11]. Another study [29] kept the index unchanged and improved the query algorithm.

3.1.3. Quadtree-based Indexing Structures. Different version of quadtrees are available depending on the type of data supported by the data structure. Some commonly used quadtree types include the PR-quadtree (also known as PR bucket quadtree) and the PMR-quadtree [30, 31]. An example of a PR-quadtree space partition and tree construction is illustrated in Figure 3.3.



(a) Space Partition for PR-quadtree (b) Tree structure of the PR-quadtree

Figure 3.3 An Example of PR-quadtree

The PR bucket quadtree is often chosen over an R-tree to avoid the object's MBR representation. The quadtree is not required to define MBRs explicitly; Instead, the regions covered by each non-leaf node is found by repeatedly dividing the space into four quads until each cell's capacity is less than the tree node capacity. Thus, no overlapping would be found among the tree nodes. The leaves of the tree contain points (i.e., the object's position). This point-wise representation is easier to handle than MBR.

The *Scalable Trajectory Index for Predicted Positions in Moving Object Databases (STRIPES)* [9] extends the PR bucket quadtree to index mobile objects. STRIPES considers each moving dimension separately, applying the dual transformation for each dimension. The object's mobility representation is then altered from a line to a point (v, x_{ref}); where v and x_{ref} represent the object's velocity vector and the reference position at time t_{ref} ($t_{ref} < t$), receptively. These points are stored in a PR bucket quadtree. STRIPES maintains two similar PR bucket quadtree structures

for two consecutive time periods. These structures are used interchangeably in the following time periods, providing enough time for the previously used structure to flush old information while preparing for the next time period. Consequently, the updating object information is completely isolated from the query processing. As a result, maintenance cost can be neglected, as it is offline.

3.1.4. Hybrid Indexing Structures. The Q+R-tree [28] is a hybrid structure that combines a quadtree with the R*-tree. Each tree is built separately. The quadtree stores fast mobile objects while the R-tree maintains slow objects. The definitions of both fast and slow objects were based, primarily, on the sub-space they move on. For example, parking lots and the areas around homes and offices are considered slow movement regions. These areas can be identified according to either a map or historical data.

In contrast to the traditional R*-tree, Q+R-tree maintains neither a lower nor an upper bound for a leaf node. Instead, it stores all of the objects in one region in one MBR. This is done to reduce both the insertion and the update cost, as two MBRs will not overlap one another. Because the R*-tree is not used in the traditional manner, the purpose of using an R*-tree in this approach has not been explained clearly in the literature [28]. The index, however, performed better than did either individual quad-tree or the R*-tree.

The Q+R-tree is different from the other aforementioned approaches as it does not rely on the linear representation of a moving object. Instead, it expects update messages to maintain up-to-date position information.

3.1.5. Summary. This chapter addressed some of the recent indexing structures used for mobile objects in Euclidean space. A summary of the those approaches is presented in Table 3.1. The summary includes the types of queries, advantages, and drawbacks of each indexing structure discussed above. The approaches were categorized according to their base structure. The most common base structures included the R-tree, the B^+-tree, and the quadtree. In addition to those indexing structures, some hybrid versions of the common base were presented. Their key features and impact of the key features on indexing (within parenthesis) are compared with one another in Table 3.2.

A comparative analysis of these approaches is difficult due to the inconsistency of the experimental environment and the lack of a common benchmark. The experimental study conducted by Chen et al. [32], however, compared

Table 3.1 A Comparison of Different Base Structures as Moving Object Indexing Structures in Euclidean Space

Feature	R-tree	B+-tree	Quadtree
Space partition	Dynamic (handling space upgrades/degrades are easy), possible overlapped subspaces (high search cost), height balance tree (high restructuring cost)	Static (handling space upgrades/degrades needs redesign), no overlaps (less search cost)	Dynamic (handling space upgrades/degrades are easy), no overlaps (less search cost)
Tree Structure	Dynamic, height balance tree (high restructuring cost)	Static, height balance tree (less restructuring cost)	Dynamic, height balance tree (high restructuring cost)
Moving Object Representation	Velocities and attached to MBRs (unconditional MBR expansion, but easy to simulate the mobility)	Objects' 2D position converted to a 1D sequence number (add extra processing overhead for updates and query execution)	Dynamic (handling update messages usually demands tree restructuring)

several of the aforementioned index structures to one another. This study includes the TPR-tree, the TPR*-tree, the RUM-tree, the STRIPES, the B^x-tree, and the B^{dual} . Among these, the B^+-tree based indexes (i.e., the B^x- and the B^{dual}-tree) demonstrated the best update performance with a reasonable query cost. Both the TPR-trees and the TPR*-trees gave the best query performances at the expense of the worst update performance. With regard to storage, both the TPR-tree and the TPR*-tree consumed the least amount of storage. Both the B^x-tree and the B^{dual}-tree consumed storage close to that of the TPR-tree and TPR*-tree.

Chen et al. [32] also concluded that the TPR-tree is better for an environment that requires a greater number of queries with fewer updates. This conclusion was further extended to include two more environments: environments that require both fewer queries and higher updates and environments whose behavior is unknown. It was shown that the B^x-tree is better in the former environment, while STRIPES is superior for an unknown environment [32].

The Euclidean space mobility representation is mostly suitable when the objects have random movement behavior (e.g., animals and sensors). This random movement, however, is not practical for all types of objects. For example, vehicles are confined to the underlying road networks. Thus, these indexes might not be able to effectively support mobile object indexing under road network constraints.

3.2. INDEXING MOVING OBJECTS ON ROAD NETWORK

Knowing the fact that mobile objects move constraint to the underlying infrastructure allows the server to provide more precise information to mobile users [15–17]. The constraint, however, makes both the mobility patterns assumed in the Euclidean space and their approaches invalid.

Research based on Euclidean space mobility patterns is mainly based on two primary assumptions: the linear movements and the constant/random speeds of the objects. The linear movement can no longer be accepted, as the roads cannot be assumed to be straight lines. Instead, these mobile objects move along a path through the road network where more direction changes can exist. At the same time, the speed may not be either steady or random. Rather, the mobile objects might be forced to change speeds depending on road speeds, weather condition, road condition, and so forth.

The mobile object indexing under the road network has been addressed under two categories: historical positions of moving objects [33–35] and real-time

Table 3.2 Indexing Schemes for Moving Objects in Euclidean Space

Indexing Name	Base Indexing Structure	Supported Queries	Query Cost	Update Cost
TPR-tree [5]	R-tree	Time slice, window, moving	Better than R*-tree [5] Good [32]	Better than R*-tree [5]
TPR*-tree[6]	R-tree	Window	Better than TPR-tree [6] Good [32]	Better than TPR-tree [6]
Dual Space [7]	R-tree	window	better than TPR-tree [7]	better than TPR-tree [7]
R^{EXP}-tree[8]	R-tree	Time slice, window, moving	Better than TPR-tree [8]	Worse than TPR-tree [8]
LUR-tree[25]	R-tree	Range and kNN	Slightly worse than R*-tree[25]	Better than R*-tree[25]
RUM-tree[4]	R-tree	Range	Worse than R*-tree [4]	Better than R*-tree [4]
B^x-tree[10]	B$^+$-tree	Range, kNN, Continuous range, continuous kNN	Better than TPR-tree [10] Reasonable [32][2]	Better than TPR-tree [10] Good [32][2]
ST^{2}B-tree[11]	B$^+$-tree	Range and kNN	Better than TPR*-tree [11]	Better than TPR*-tree and almost similar to B^x-tree [11]
B^{dual}-tree[12]	B$^+$-tree	Range and kNN	Better than TPR*-tree, STRIPES, B^x-tree [12] Reasonable [32][2]	Better than TPR*-tree, worse than STRIPES and B^x-tree[12] Good [32][2]
STRIPES[9]	PR Quad-tree	Time slice, window, moving	Better than TPR*-tree [9]	Better than TPR*-tree [9]
Q+R-tree[28]	PR Quad-tree +R-tree	Range	Better than Quad and R-tree [28]	Better than Quad and R-tree [28]

positions of moving objects [21, 36, 37]. The latter category will be discussed in detail in the following sections as the other category is out of this article's scope.

The real-time handling of moving objects under road-network constraints involves indexing the objects' information with respect to the road network (known as composite structures) [21, 36, 37]. Composite structures use spatial indexing methods (i.e., an R-tree, an R*-tree and a PMR quadtree), grids, tables, and/or hash tables with reference to the corresponding road segments to store the road network and the moving objects, respectively. The general idea of a composite structure is illustrated in Figure 3.4. As the figure depicts, the triangle represents the road network indexing structure.

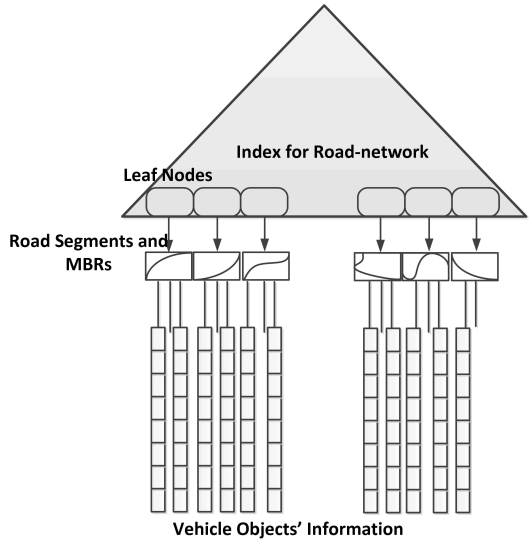


Figure 3.4 An Example of Composite Structure

This structure is usually arranged as a hierarchical tree structure. The leaf-level of the tree points to the road segments. The mobile object information is stored under these road segments.

When the road network is indexed in the R-tree family indexes (e.g., either R-tree or R*-tree), each leaf-node MBR represents the box that fully covers the corresponding road segment. The objects moving on a particular road segment are stored under that road segment. When an object reaches the end of the road segment, it is removed from the current road segment and stored under the new road segment. There are two main drawbacks to this indexing. First, a significant amount of dead space exists in a road segment's MBR. The dead space is the redundant space within the MBR but outside the road segment. This redundancy

can lead to false overlaps of MBRs which, in turn, increase the search cost. This drawback, however, does not exist in quadtree based indexing structures [21, 37], because the quadtrees, by nature, divide the space into non-overlapping quads.

The second drawback is due to the mandatory search cost involved in maintaining update messages. MBRs do not capture road segment connectivity. Thus, to locate the next road segment that the object is going to move to, the tree is required to be searched. To resolve this issue, Bok et al. [18] and Feng et al. [19] proposed two separate (but similar) techniques. Bok et al. [18] proposed the Intersection Oriented Network R-tree (INOR-tree). The premise is to store multiple edges connected to the same intersection in the same MBR. By doing so, some object updates can be done within the same MBR when the objects travel from one edge to another.

Feng et al. [19] suggested storing a fraction of each contributing edge to form a junction in the MBR. This indexing structure is known as a cross region (CR)-tree. The primary difference between the INOR-tree and the CR-tree is that the CR-tree maintains the fraction of the edge, whereas the INOR-tree defines new junctions where the edge is split by the MBR margin. However, because the CR-tree stores both the edge and its corresponding fraction, edges are duplicated if they are covered by multiple MBRs. This duplication leads to reduced space utilization.

Nevertheless, for a particular road map with the exact same MBR, both methods consume approximately the same amount of storage. The INOR-tree stores an edge several times; the CR-tree splits an edge into a number of sub-edges and stores them separately. If the number of repeated edges and the number of splits are the same, the amount of space required in both methods is approximately the same.

One disadvantage of the CR-tree and the INOR-tree is that modifying the road network adversely impacts the index structure. Both methods are also equally complex when compared to single road segment-based indexing (i.e., determining the proper road segment units, finding the MBR, and so forth). This complexity might be one reason that these novel indexing schemes have not been explored extensively.

The single edge storage unit, however, has been used in many composite structures. These structures can be classified into several categories depending on their storage method: disk-based, memory-based, and hybrid-based indexing schemes.

The disk-based indexing schemes consider both network and vehicle information stored in a secondary disk. Thus, their primary performance metric is the number of disk pages accessed. One primary advantage of disk-based index structures is higher scalability. These index structures, however, might not handle updates efficiently due to the communication delays associated with access to the secondary storage.

The memory-based indexing structures store information in memory. Their common performance metric is CPU time since memory access is faster than the disk access query processing and vehicle update handling methods are faster with CPU technology than with disk-based approaches. However, they typically show low scalability as the memory space is limited.

The hybrid-based indexing structures utilize both secondary disk and memory to store information; hence, both page accesses and CPU time can be employed to report performance. Most of the time, secondary storage maintains road network information while memory maintains vehicle information. Hence, these approaches might be scalable at a network size but not with the amount of vehicles in the network.

3.2.1. Disk-based Indexing Structures. Some of the disk-based composite structures include Indexing Moving Objects on Road Sectors (IMORS) [15], the Adaptive Network R-tree (ANR-tree) [17], the R-TPR$\pm$-tree [36], and the TPR-uv [38]. Each of these structures uses an R-tree-like structure whose leaf nodes point to road segments. Each indexing scheme has demonstrated the ability to support different queries: range [15], predictive range [17], predictive traffic flow [36], and continuous queries [38].

IMORS [15] stores road information using an R*-tree. A leaf node of the tree points to a list of mobile objects that are moving on a road segment (the road sector block). IMORS uses a separate data block to maintain the mobile objects' information (e.g. both the position coordinates and the velocity). Each entry in the data block is bidirectionally connected to the corresponding object information in the road sector block. This allows the user to locate the corresponding road segment information from the object information and vice versa. A search is initiated at the data block, and followed by the pointer to the road sector block, to find the old road sector block that corresponds to the object in an update. The R*-tree is searched and the bi-directional connection is updated to find the new road sector block for both an update and an insertion.

The *Adaptive Network R-tree (ANR-tree)* [17] is comprised of both an R-tree and an in-memory direct access table. The R-tree stores segments of the road network. Leaf nodes in an ANR-tree, similar to IMORS, also maintain the information for moving objects on the corresponding road segment. Additionally, the ANR-tree introduces a grouping concept, known as *adaptive units*, for the objects within a road segment. An adaptive unit groups those objects with similar moving patterns. This similarity is defined according to both the moving direction along the road segment and two threshold values (the speed threshold and the distance threshold). One road segment could have several adaptive units, depending on the presence of different moving patterns. Each adaptive unit maintains both the entry time to and the predicted exit time from (the trajectory bounds) the segment. These times will be used during the query process. The direct access table of a segment contains the number of objects stored, the trajectory bounds, and the pointer to each adaptive unit disk page.

The R-TPR$\pm$-tree [36] has also applied concepts similar to the ANR-tree. For example, the R-TPR$\pm$-tree selects the R-tree for storage. It also considers similar mobility patterns of objects. The similar patterns in the R-TPR$\pm$-tree, however, are defined only by the direction in which they are moving along the road segment.

In addition to the R-tree, the R-TPR$\pm$-tree [36] maintains a set of TPR$\pm$-trees. These TPR$\pm$-trees are attached to the leaf nodes of the R-tree. Each tree maintains the objects moving on the corresponding road segment. The root of the TPR$\pm$-tree points to two children TPR-trees; Each represents the objects moving in the same direction. In doing so, the R-TPR$\pm$-tree [36] tries to reduce the expansion of MBRs by separating the objects according to their moving direction. Thus, the expansion of the MBR boundaries is not as severe as it was with the TPR-tree. A considerable number of mobile objects must be present on the road segment, however, to realize the advantage of TPR-tree adoption over other list-based structures (e.g. the direct access table in the ANR-tree). Furthermore, it might not perform well for the road networks with shorter road segments.

The TPR^{uv}-tree [38] is comprised of an R-tree, a direct access table, and adjacent lists. Similar to the indexes previously discussed, the R-tree indexes road network information. A direct access table is connected to each leaf node in an R-tree. Each entry in this table maintains information regarding that particular road segment. This information includes the road ID, the speed limit, adjacent lists for each end node, and a pointer to the mobile objects. The indexing structure handles

the update messages through the adjacent lists. The TPR^{uv}-tree provides an almost constant time update cost [38].

The *Group Update Time Parameter R-tree (GTR)* [39] and the robust *Robust Group Update Time Parameter R-tree (RGTR)* [40] are index structures that support efficient updates through group-wise execution. Both insertion and deletion messages are buffered and performed in groups. They neglect the updates sent by vehicles with constant velocities. The user, however, is required to send the previous velocity information in the update messages to determine the velocity's steadiness. Thus, these models increase the communication cost when compared to a traditional update message environment. The difference between GTR and RGTR appears in their data structures. The RGTR discards the buffer which was keeping track of update messages in the GTR. Instead, updates are performed instantly, as an insertion followed by a deletion. The additional cost is traded off by introducing compressed object representation, making it possible to accommodate more objects in one tree node.

3.2.2. Memory-based Indexing Structures. One common approach to improve update costs includes memory-employed indexes: fully memory-based [21, 37, 41, 42] and hybrid [24, 43] (will be covered in Chapter 3.2.3). The memory-employed indexes, usually, employ lists [21, 41] or tables [37, 42] to maintain the mobile objects' information. The road network could be indexed as either an R-tree [42] or a PMR quadtree [21, 37]

Implementation of the R-tree family index in memory is similar to that of disk-based indexes. The PMR quadtree, however, organizes the entire space into a PMR quadtree. A leaf of the tree contains the covered edge IDs. Connectivity among the edges is maintained in a table.

3.2.3. Hybrid Indexing Structures. Hybrid approaches typically store the network information on the secondary disk, while mobile objects are maintained in memory (e.g., MOVNet [24]). The MOVing Objects in Road Networks (MOVNet) employ both an R*-tree to store static road network information and an in-memory grid structure to store object positions. The grid structure divides the entire mobile area into cells. Each cell maintains a list of mobile objects whose current location falls into that cell. Mobile object updates are directly handled in the grid cells. The R*-tree is accessed whenever road network distance is required. For example, in a query process, the corresponding Euclidean space query is performed on the

grid first. The R*-tree is then searched to obtain the corresponding road segments. Finally, a partial map is constructed in the memory to consider the network's distance.

Disposable Index for Moving objects (DIME) [43] focuses on reducing update costs. DIME manages several indexing structures, both in secondary disk and in memory, to maintain an object's position. When the objects on secondary disks must be updated, a new in-memory index is created. This index maintains only the objects whose positions were updated. Entries on the initial on-disk tree structures are not updated with the new location information. Instead, they are flagged as obsolete. When the in-memory entries receive position-updates another new index is created to maintain them. Essentially, no indexing structure is modified for an update message; a new index is created instead and the old indexes are deleted.

There are several drawbacks to this method. One is that all objects' positions may eventually be placed in memory, although DIME begins with a secondary disk index structure. A lower update cost will be penalized at query processing as both the obsolete and valid entries are filtered out at that time.

3.2.4. Summary. The previous section discussed the primary indexing approaches of moving objects on road networks indexing Table 3.3 summarize these approaches. The summarization considered the following key features: the storage (e.g, disk based, memory based and hybrid), the structures used in indexing the road network information (e.g., R-tree, R*-tree, and table), structures used to handle road mobile object information (e.g., list and table), supported query type(s) (e.g., Range, KNNQ, and RKNNQ), the query response frequency (snapshot or continuous), and support for the predictive information (predictive or non predictive).

From Table 3.3, we can observe the following: First, R-tree family indexes are common structures that have been employed to store road network information. Secondly, the mobile objects are indexes with reference to the corresponding edge they move on. Thirdly, the common types of queries supported include the range and the KNNQ. Additionally, most of the queries are snapshot queries and non-predictive.

Table 3.3 Indexing Schemes for Moving Objects Under Fixed Network

Indexing Name	Storage	Road network	Mobile objects	Supported Queries	Continuity	Predictivity
IMORS [15]	Disk	R*-tree	Data Blocks for each R-tree leaf	Range	Snapshot	Current
ANR Tree [20]	Disk	R-tree	Adaptive Unit(s)[a] for each R-tree leaf.	Range	Snapshot	predictive
R-TPR$^{\pm}$-tree [36]	Disk	R-tree	TPR$^{\pm}$-tree[b] for each R-tree leaf	Navigation	Snapshot	predictive
TPR^{uv} [38]	Disk	R-tree and a adjacent list for each edge	direct access table	KNN	Continuous	Current
GTR[c] [39]	Disk	R-tree	Object list for each leaf node of R-tree	Range	Snapshot	Current
RGTR[d] [40]	Disk	R-tree	Object list for each leaf node of R-tree	Range	Snapshot	Current
SR*-tree [41]	Memory	R*-tree and a network connectivity table	Object list	Range	Continuous	Current
DLM [21]	Memory	PMR Quad	List	Reverse KNN	Continuous	Current
Prediction Distance Table [42]	Memory	B^{+}-tree	Hash, on destination	Range	Snapshot	Predictive
SI and ET [37]	Memory	PMR-Quad tree and an hash table on edge id (ET)	ET	NN	Continuous	Current
MOVNet [24]	Hybrid	R*-tree (disk)	Array of objects (memory), objects are grouped into grid cells	KNN and Range	Snapshot, continuous[e]	Current
DLME [43]	Hybrid	-	R*-tree and B^{+}-tree	Range	Continuous	Current

[a] similar to a one-dimensional MBR in the TPR-tree
[b] a modification of TPR-tree that handle each direction separately
[c] Addition to the Road network and moving objects, contains a list of obsolete information while table is maintained.
[d] the difference of RGTR from GTR is that the additional obsolete information table of GTR has been discarded.
[e] A query algorithm for the continuous versions using the same index structure as presented in [44]

3.3. QUERY PROCESSING UNDER ROAD NETWORK CONSTRAINTS

This section focuses on continuous queries on mobile objects. The term *continuous* refers to the continuous monitoring of an issued query. This continuity, however, can come in two forms: continuous monitoring of static query and continuous monitoring of dynamic queries (more often called moving queries). The former types of query are queries whose parameters do not change over the time (e.g., the value of K and the issuer's position in KNNQs and the value of the range and the issuer's position in RQs), but the query result might be changed due to the movement of querying objects. The later category is referred to the queries whose parameters also might changed over time (e.g., most commonly, the issuer's position). In either case, frequent and large amounts of update messages should be handled in an efficient manner for up-to-date query results.

Regardless of the category of the query, a naive approach for answering a continuous query is to re-conduct the same query every timestamp till the expiration of the life time of the continuous query. This may involve lots of unnecessary efforts if there is no change of the query results at consecutive timestamps. Upon closer examination, there is a need to update the query results only when an object in the current result becomes invalid or a new object joins the result due to the change of previously considered information in the query processing.

Taking this observation into the consideration, most approaches on continuous queries have been developed to process queries in two main phases: the initial and the maintenance phase. The initial step generates results for new queries. The maintenance step maintains the result obtained in the initial phase. It considers the influence of the objects' mobility on the query answer.

The simplest method to find out the influenced mobile objects would be to evaluate each update message received from all objects. However, this could directly affect the performance in two ways: a higher processing cost (in terms of both time and resources) and an increased communication cost. Thus, past research has focused on efficient query processing techniques to reduce these costs.

A common technique for handling high frequent update messages is defining safe regions for moving objects in which the movement of objects within the safe region does not alter the result. The safe region could be calculated by either the object itself or the server. The calculated safe regions can then be managed by the object [45, 46] and server [47]. If the safe regions are known to the object, the object initiate an update message when the object crosses the safe region. In some

situations the objects update message itself cannot confirm the alteration of query result(s). In such situations, the server probes object information selectively.

The details of the aforementioned techniques could be different in static and dynamic queries. Thus the details of continuous query processing techniques in the literature will be addressed separately. These techniques are specifically addressed when handling the updates of the mobile objects and the data structures used to maintain both queries and query results.

3.3.1. Continuous Monitoring on Static Queries. Prabhakar et al. [48] proposed an approach that supports static range queries on mobile objects whose movements are relaxed for the Euclidean space. They considered query indexing opposed to object indexing. As a result, every query is indexed in an R-tree based structure called *Query Indexing (QI)*.

The initial query response is obtained by searching overlapped queries against each object position. Once the initial result is obtained, the maintaining phase compares only the update messages from the users against the QI. This approach might scale up with the number of queries as the queries are indexed considering their spatial closeness. However, the approach might not scale up with the number of objects. Specially, the initial phase would yield a higher search cost when there is a higher number of objects because the algorithms compares each object against the QI.

In order to reduce the number of update messages, an individual object maintains its own safe region calculated by the server. The safe region is calculated as the shortest distance between an object and a query boundary. This calculation increases the overall computational cost at the server side. Moreover, the advantage of the safe region might lessen, as the distance between query and object is diminished. Another disadvantage of this safe region concept is the requirement of recalculating the safe regions upon both receiving a new query and expiring a query.

Cai et al. [45, 46] also addressed the same continuous query type, the static continuous range query. Two methods are proposed named *Spatial Query Management (SQM)* [45] and *Monitoring Query Management (MQM)* [46]. The difference between SQM and MQM comes in the differences of the mobile objects' computational capability. In fact, MQM considered heterogeneous mobile objects where SQM does not.

In both of these approaches, the entire space is divided into disjoint spaces called subdomains. The safe regions in both SQM and MQM are defined in terms of these subdomains. Safe region of objects in the MQM are comprised of one or more subdomains depending on their computational capability. Safe regions in the SQM, on the other hand, have one subdomain for each object. This safe region calculation is significantly simple compared that of in QI [48].

The partial areas of a query region overlapped on each safe region (called monitoring regions) are maintained in a binary partition tree (BP-tree). Mapping between the monitoring region and the query is maintained in the relevance table.

The initial phase of the continuous query is performed on this BP-tree. It compares the query region of the newly issued query in the BP-tree and calculates the monitoring regions. The objects in the relevant subdomains are then informed with these monitoring regions.

The maintaining phase is mainly handled by the query candidates. They report their influence on the query results and the server updates the query issuer based on those reports. Thus, the work load at the server is much less than the work load handled by [48]. As the computational work-load is distributed among the mobile objects, this method is more apt to be scaled up based on the number of queries and the number of moving objects.

The approach proposed by Hu et al. in [47] also considers the safe region in order to reduce the communication cost. The main difference of this approach is that the object's actual positions are not being indexed, but the safe regions for each object are. In addition to object's safe region index structure, an in-memory query-index is also maintained, which keeps track of query information and their results.

The query-index is constructed in a similar manner as that in the BP-tree [45, 46]. In fact, for the query-index also the entire space is partitioned into disjoint areas (into a grid). The cells are indexed in the query index. Each cell points to the information about query regions (partially) overlapped with the cell (similar to the monitoring region in [45,46]), also known as the quarantine area. These quarantine areas are indexed in the query-index.

The safe region of an object for one particular query is the quarantine area or its complement. The decision is made based on whether the object is in the quarantine area or not, respectively. The overall safe region of an object is the intersection of individual query-safe regions.

Objects update the server when it crosses these safe regions. In some situations, these messages themselves are unable to decide the changes to the query result. If that happens, the server probes some selected object's exact location information to resolve the uncertainty.

The experimental results [47] shows that the scalability of the execution time against the number of registered queries. This scalability has been achieved as the pointer in a grid cell points to all the queries whose quarantine area overlaps with the cell. In this manner, unnecessary query consideration can diminished. As a result, the overall execution time is also reduced.

Wang and Zimmermann proposed algorithms [44] for continuous range query by extending the indexing structure that they proposed for MOVNet [24]. Additional information maintained in MOVNet, shows how each cell in the grid maintains a list of vertices which are connected to other cells. The distance between each connected node pair within the cell is also maintained.

Once a range query is issued, edges relevant to the cell of the issuer are retrieved from the secondary disk. Their previous approach[24] retrieved edges overlapped by every cell within the query range. From the retrieved edges, a tree, Shortest Distance-based Tree (SD-tree), can be created making the query point the root of the tree. Paths between nodes in the tree gives the shortest distance among them and the distance to any of the nodes does not exceed the query range. When the query issuer is moved, the tree is rotated, which requires expanding the tree along the subtree that the query pointer moved. Moving the query issuer also requires trimming in the other subtrees. In cases of other vehicle's movements, their new position is checked with the SD-tree and added or removed from the old result accordingly.

The main drawback of this approach is the amount of data managed in the memory. Only the road network is stored in the secondary disk in the R-tree structure. The information about the vehicles, connectivity and the SD-tree are stored in the main memory. This becomes severer when the number of queries in one cell gets increased. At that point, each query needs to maintain an SD-tree which consumes more memory.

Mouratidis et al. proposed a method to process continuous queries in [49] for nearest neighbors. In this method, the safe region is a circle whose radius is the distance to the nearest neighbor. They considered the service area to be a grid. The process starts from the cell which contains the query point. Then the cell with the query point accesses the cells closest to by iterating through its neighbors

clockwise, starting with the left. This process repeats until it finds K objects. The main difference in this approach is the safe region is considered per query; not for mobile objects. Thus, in this method, mobile objects do not pay a penalty for keeping track of safe regions. However, the safe region is bigger compared to safe regions maintained by mobile objects. The larger the safe region is, the higher the update consideration chances are.

In addition to the two most common query types, *KNNQ* and *Range* (discussed above) and some other query types have also been considered under the continuous static query monitoring. For example top K queries [37], RKNNQ [50], and *Detour queries* built to find the shortest detoured route [51]. All work together to reduce the communication cost by reducing the update messages from the mobile objects.

In static query monitoring, the most common approach is to define a safe region for each object. In most approaches the safe region is calculated by the server, which preserves the confidentiality of the query issuer. The object is responsible for sending an update message whenever that object crosses the boundary. In this way, it tries to reduce the communication cost and unnecessary update processes.

3.3.2. Continuous Monitoring on Moving Queries. When the moving queries are monitored, the query itself also moves along with the querying objects. Thus, most approaches proposed under the static query do not fit well, e.g., safe region calculations. The following discusses the approaches on moving query monitoring techniques.

Stojanovic et al. proposed an algorithm for processing dynamic continuous range queries [52]. The algorithm was based on three steps: filter step, pre-refinement step and refinement step. The filter step and the pre-refinement step produced the initial query response. The refinement phase explorers possible overlaps of the query range along the path of each object. The detailed information on these overlaps (e.g., time period(s) and location(s)) are stored in two tables: Continuous Range Query Table (CRQT) and Mobile Object Table (MOT). The refinement step periodically ensures the validity of the entries in these tables.

One main drawback of this algorithm is the huge amount of duplicated data managed in memory. Hence, this method might not scale up well with the number of objects.

In contrast to the approach proposed in [52], Gedik and Liu proposed a distributed approach for spatial queries to reduce the work load at the server [53]. The server broadcasts query information upon receiving a new query. Each object then decides whether it is in the monitoring region of the query by examining the query information. If the object determines that the query is in its monitoring region, the mobile object will maintain the query information. The information will be maintained until the query leaves the monitoring region, or vice versa.

The mobile object estimates the query issuer's position depending on the query information such as velocity, time and position. If this estimated position difference suggests a possible change in the query result, the latest information is passed over to the server. Since the updates are sent only when an object identifies a potential query result change, the communication cost is reduced. However, the main drawback of this approach is that it violates the confidentiality of query issuers.

Both SEA-CNN: Shared Execution Algorithm [54] and SCUBA: Scalable Cluster-Based Algorithm [55] present scalable approaches to the KNN and range queries, respectively. SEA-CNN groups queries based on their searching regions and locations. The SCUBA, on the other hand, groups not only queries but also mobile objects into groups according to their common spatial relationship. In this approach, when compared to SEA-CNN, both objects and queries could be included in a group.

The performance of these approaches is better when steady clusters are present. The relative speed of objects is within the clusters influence for having steady clusters. When the relative speeds are significantly high, the objects which were in the same group might not fall into the same group in the next update. In this case, not only do unsteady clusters increase the cluster maintenance cost, but also it increases the query execution cost as the number of merged clusters is high.

Liu and Hua [56] proposed algorithms to process dynamic RQ and KNNQ under the network consideration. The supported information for algorithms are maintained in-memory at the server. Such information includes query issuers' details, query information, and the road segments that overlap with each query.

The process of range query begins with obtaining its snapshot query answer. The objects in the snapshot query answer keep track of the query position and the query range. With this information, the objects update the server when the object itself moves out of the query range or vice versa. Upon receiving the update message from the object, the server updates its query result accordingly.

Additionally, the server updates object's new query list and their information to maintained.

The KNNQ processing follows a similar method as that of RQ. The KNNQ is handled as a range query where the range is the distance to the furthest vehicle from the query issuer. If a new object has the potential to be in a query result, current positions of all objects in the previous result are considered and insert the new vehicle in the appropriate position in the list. This approach shows better performance when the range is higher as the number of messages to be sent is less. However, similar to the approach proposed by Stojanovic et al. [52], this approach also maintains a considerable amount of data on memory. These data include duplicates of object information. For example, each query maintains the list of all the moving objects in the response and each such moving object maintains all the queries which were affected.

3.3.3. Summary. The previous sections discussed the approaches to continuous monitoring of LDQs (both static and dynamic). The most common types of queries developed for continuous monitoring queries include RQ and KNNQ. Table 3.4 summarizes these approaches.

In static query monitoring, the most common approach is to define a safe region for object. In most approaches the safe region is calculated by the server, which preserves the confidentiality of the query issuer. The object is responsible for sending update message whenever that object crosses the boundary. This way it tries to reduce the communication cost and unnecessary update processes.

Despite, in moving query monitoring some approaches broadcast the query information and each mobile object decides its effect on the query and safe regions are maintain accordingly. However this violates the privacy of query issuers compared to the approaches in static query monitoring.

Table 3.4 Continuous Monitoring on Queries Under Fixed Network

Query Algorithm	Continuity	Query Types	Advantages & Disadvantages
Prabhakar et al. [48]	Static	Range	Introduces the indexing queries instead of objects; safe regions reduces the update cost:
Cai et al. [46]	Static	Range	Distributed workload among clients gives scale up at server level; supports heterogeneity of clients; clients work load is not uniform; more dynamic environment creates higher communication cost
Hu et al. [47]	Static	Range and KNNQ	Safe region reduces communication cost; higher client privacy preserving; grouping queries on cell basis reduces the unnecessary query consideration and hence provides the scalability.
Mouratidis et al. [57]	Static	KNNQ	Safe regions reduces communication cost; work load at the clients is high; confidentiality violation
Wang and Zimmermann [44]	Static	Range	Less execution cost; Higher storage cost due to duplicates
Mouratidis et al. [49]	Static	KNNQ	Less work load for the clients
Stojanovic et al. [52]	Moving	Range	Distributed approach reduces server work load; Carry significant false positives for individual comparison step, privacy violation of query issuer's
Xiong et al. [54]	Moving		Grouping information on spatial relationship gives the scaling up
Nehme and Rundensteiner [55]	Moving		Grouping information on spatial relationship gives the scaling up; group maintenance cost is high on dynamic environments
Liu and Hua [56]	Moving	Range and KNNQ	Higher storage cost due to duplicates; need processing capability at the client to decides its effect on queries and send update messages

3.4. DENSITY QUERIES

A density query is issued by a user to discover dense regions on the global space. Denseness is determined according to mobile object concentration per area. If the concentration exceeds a particular threshold, then that area is defined as a dense area. The threshold could be specified by the system or the query issuer.

One can interpret density queries as a variation of other common types of queries (e.g., an aggregated range query, which provides the number of objects in the range but not necessarily details of individual objects). However, significant differences can be noticed in a density query compared to a range or KNNQ. One main difference would be the RQ and KNNQ allows the reference point to be specified by the issuer, which does not happen in density queries. Thus, to query dense areas using either a RQ or a KNNQ, one is required to know possible dense areas to issue the query a priori.

The density query process is also greatly different than RQ and KNNQs. In these queries, since the reference point is provided the search space can easily be pruned. In the dense queries, on the other hand, the entire space is required to be monitored in order to identify dense regions. Additionally, depending on the objects' distribution among the space, it could be possible to have more than one dense regions. These features makes the density query different than the other types of queries.

Several research projects have been conducted to address the aforementioned issues. The first work on processing a density query was proposed by Hadjieleftheriou et al.[58]. They proposed two versions of density queries: Snapshot Density Queries (SDQ) and Period Density Queries (PDQ). The SDQ provides the density information for a specific time instance in future, where PDQ provides the validity period of the response addition to the density information.

In these approaches, the entire space is divided into a grid and density regions are reported in terms of cells (i.e., whether a cell is dense or not). However, the cell based density definition is unable to captures all possible dense areas within the global area. For example, consider a dense area distributed among couple of cells. If these cells are considered individually, each cell could be below the threshold. Thus, no dense area would be identified, even though a dense area is present globally.

The aforementioned problem, the answer loss problem [29], has been resolved by Jensen et al. [29]. They have given a new definition for the density query, named effective density query (EDQ). The EDQ also divides the moving space into

a grid. However, they have relaxed the rigid, cell based density identification (as it was in [58]). Instead, the density areas occur at any place, in any shape, and size can be identified.

Another approach for addressing density queries is presented by Ni and Ravishankar [59]. Their definition of density query is named pointwise dense regions (PDR). In their definition also, the dense region could be in any shape and any size. The searching area, however, is always assumed to be greater or equal to twice the cell width of the cell. Since this assumption guarantees that a 4-cell block is searched, the PDR query is also able to avoid the answer loss problem.

To obtain a PDR query result, Ni and Ravishankar [59] proposed two algorithms: an exact method and an approximate method. In the exact method, each cell maintains the objects information for each query time in a histogram. The histogram information is used in the first phase of filtering. Then each candidate cell is applied on a range query. In the approximate method, the density distribution is approximated using a polynomial function, where an exact algorithm is not used. As expected, the approximated method performs faster compared to the exact method, despite the reduced accuracy.

All the aforementioned query algorithms consider the snapshot version of the query. Due to the violation of assumptions on the objects' path, speeds and so on, results can get changed as the time advances. The methods proposed in [60, 61] have considered these features and proposed continuous density query algorithms.

Similar to the snapshot queries, these query definitions are also associated with two thresholds: a density threshold and an area threshold. The definition of these thresholds, however, are different than that of snapshot queries. The continuous density query definition considers these thresholds global to the entire space, but not per query. The entire space is divided repeatedly into quadrants until the area is less than the given area threshold. Each area can then be labeled as either dense or non-dense - not both.

The main drawback of this approach is the less flexibility because different users might be interested in different degrees of density. At the same time, in practice, these threshold values are spatially and temporally not uniform. For these reasons, this method might not be able to provide a better service to the users.

In addition to the aforementioned density query algorithms, to which movements are considered on the Euclidean space, only a couple of works have been conducted on density queries restricting the movements to road network [62].

The density definition has been modified and now takes road topology into consideration. Thus, density is given per road segment where it was per area in the Euclidean space density definitions. In fact, Lai et al. [62] propose Effective Road-Network Density Query (e-RNDQ). The e-RNDQ also considers the density and road segment length threshold. Furthermore, the distance between any two neighboring objects in a dense road segment should not exceed the given distance threshold. This condition prevents having skewed object distribution in a query result.

Given the aforementioned conditions, [62] have proposed a cluster based algorithm in obtaining the query result. However, these clusters are on the current position of the objects. Thus, the proposed query definition and algorithm can identify current dense road segments, but not future density.

3.4.1. Summary. The previous section discussed the approaches on density queries. Few works have addressed density queries. Table 3.5 summarizes the characteristics, and pros and cons of each approach.

3.5. SUMMARY

This chapter summarized, compared, and contrasted past work on mobile data indexing and querying. The discussion showed that the most existing indexing schemes relax the objects mobility to Euclidean space which is not practical with objects such as vehicles. Some other, but recent, works have developed indexing schemes supporting objects moving on road network.

Nevertheless, the discussion on querying mobile data showed that only a handful of query types have been developed to service road network based mobile data inquiries using the aforementioned road network considered mobile indexing scheme. Furthermore, the discussion discloses the shortfall of future/predictive information delivering services for mobile users.

Table 3.5 Density Queries

Query Algorithm	Query Types	Moving Space	Advantages	Disadvantages
Hadjieleftheriou et al. [58]	Snapshot	Euclidean	Supports two types of queries: dense regions for a period of time and dense regions for a point in time; provides future densities	Size and the shape of the query area is not flexible, but is fixed to the grid cell size; answer loss problem: some dense areas are unable to be captured. Object distribution is not interpreted from the result (evenly, skewed, etc.)
Jensen et al. [29]	Snapshot	Euclidean	Support dense regions for a point in time , provides possible future densities, addresses answer loss problem, size of the query area is flexible	Shape of the query area is fixed; object distribution is not interpreted from the result (evenly, skewed, etc); answer is not complete as overlapped density areas are not provided
Ni and Ravishankar [59]	Snapshot	Euclidean	Addresses answer loss problem; provides density distribution information, provide future densities	The query area threshold must be grater than the two units of the grid cell size
Hao et al. [60] and Wen et al. [61]	Continuous	Euclidean	Issuers are up to date with the current density	The issuer has no control over the density threshold, thus less flexible.
Lai et al. [62]	Snapshot	Road Network	Reported dense areas are evenly distributed according to the user provided distance threshold	Provides the current density information not future

4. INDEXING UNDER ROAD NETWORK CONSTRAINTS

When objects' mobility is constrained to the road network, storing mobility information corresponding to the road network provides intelligence to both the data and the overall system. The added information, however, increases the amount of data that needs to be managed in order to support queries efficiently. This facet prompts for further challenges in traditional indexing structures utilization.

The utilization becomes even more challenging when the mobility information is required to support predictive queries. The main reason for this is that most moving object management techniques [4–6, 25] deal with model objects moving freely in Euclidean space but not under road-network constraints. One type of Euclidean space based approaches relies on a snapshot of the object's position at each timestamp [4,25], while the other type relies on object's position using a linear function [5, 6, 8–12]. Approaches in the first category are not capable of supporting predictive queries. Approaches in the second category, however, are capable of predicting future positions by assuming that the object moves along a straight line at the most recently reported velocity. This assumption is not realistic under road-network constraints; roads are more often curvy than straight, and maintaining a steady velocity is difficult. Therefore in general, queries generated based on the aforementioned approaches lack accuracy with respect to predictive queries under road network constraints.

This chapter discusses a solution to the aforementioned challenge and introduces a novel, efficient, and effective indexing structure. The indexing structure manages moving objects under road-network constraints that support predictive queries. In particular, we propose a new indexing structure called the R^D*-tree*; here, D stands for direction.

The rest of this chapter is organized as follows: Chapter 4.1 introduces the proposed index structure – the R^D-tree. Chapter 4.2 presents the tree maintenance algorithms and Chapter 4.3 briefly presents the queries supported by the indexing structure.

4.1. THE R^D-TREE INDEX STRUCTURE

The R^D-tree indexes two types of data: road-network information and object location information. The road network is represented as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$,

where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each edge $e = \{\mathbb{v}_1, \mathbb{v}_2\} \in \mathcal{E}$ represents a road segment[2] in the network where $\mathbb{v}_1, \mathbb{v}_2 \in V$; $\mathbb{v}_1$ and $\mathbb{v}_2$ are starting and end nodes of the road segment, respectively. Furthermore, each edge is associated with two parameters: l and s, where l is the length of the edge and s is the maximum possible speed on that edge.

A moving object O is represented by the tuple $\{o_{id}, x_t, y_t, o_e^t, o_e^{t+1}, o_v^t, o_{gd}, t\}$ where o_{id} is the unique object ID, x_t and y_t are the coordinates of the moving object at the latest update timestamp t, o_e^t is the current road segment that the object is on, o_e^{t+1} is the next road segment that the object is heading to, o_v^t is the object's velocity (or speed), and o_{gd} is the object's travel destination; it is assumed that most moving objects are willing to disclose their tentative traveling destinations to the service provider (server) in order to obtain high-quality services, albeit the destination may change during the trip.

Figure 4.1 illustrates the overall structure of the R^D-tree. The R^D-tree is designed as a disk-based structure since its potential need for a huge amount of storage space to store large amount of vehicles and complex road maps. Such storage may not be available at the service provider end that usually support multiple types of services simultaneously. The R^D-tree is composed of an R*-tree [63] and a set of hash tables. The road-network information is indexed by the R*-tree. Each entry in the non-leaf node is in the form of $(node\_MBR, child\_ptr)$, where $node\_MBR$ is the MBR covering the MBRs of all entries in its children pointed to by the $child\_ptr$. Leaf nodes in R*-tree pointing to hash tables represent vehicles at each road segment. Each entry in the leaf node is in the form of $(edge\_MBR, obj\_ptr)$, where $edge\_MBR$ is the MBR of a road segment and obj_ptr links to a hash table storing objects moving on this edge.

Each hash table has an N_d hash bucket, where N_d is the number of traveling directions. Each bucket has two linked lists that provide a finer grouping for objects based on their traveling directions. Moving objects with similar traveling directions are hashed to the same hash bucket and stored in one of the sorted linked lists maintained in that hash bucket. Moreover, for easy update, each object also has a pointer directly linked to the edge that it is currently moving on. The details of the construction of the hash table and link lists will be elaborated shortly.

The critical issue in constructing the hash table is to determine an effective hash function which groups objects with similar traveling directions. The object's traveling direction is determined by the angle between the horizontal line and the

[2]Road segments and edges may be used interchangeably throughout this paper

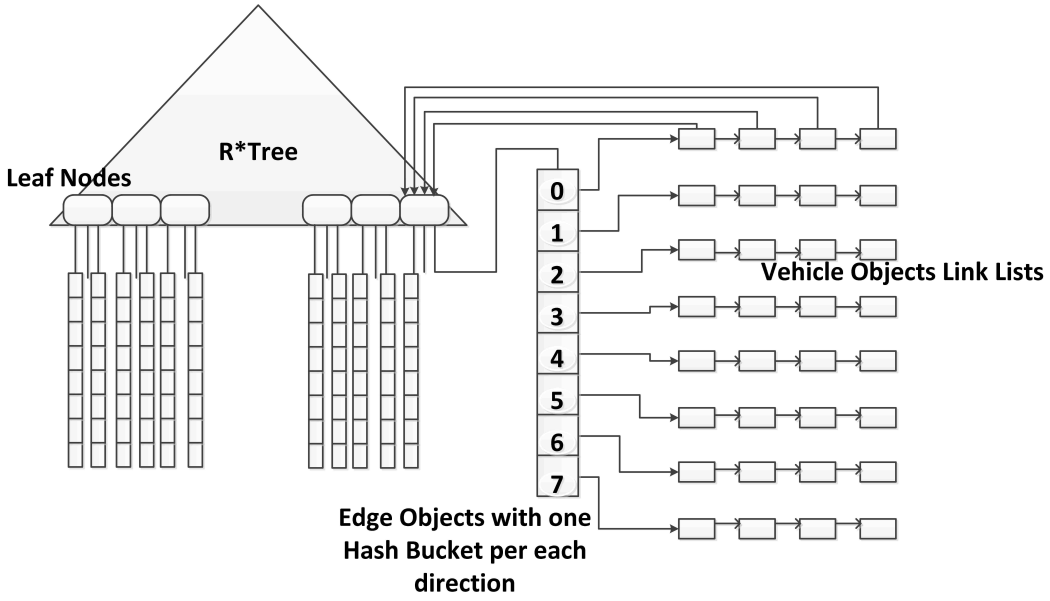
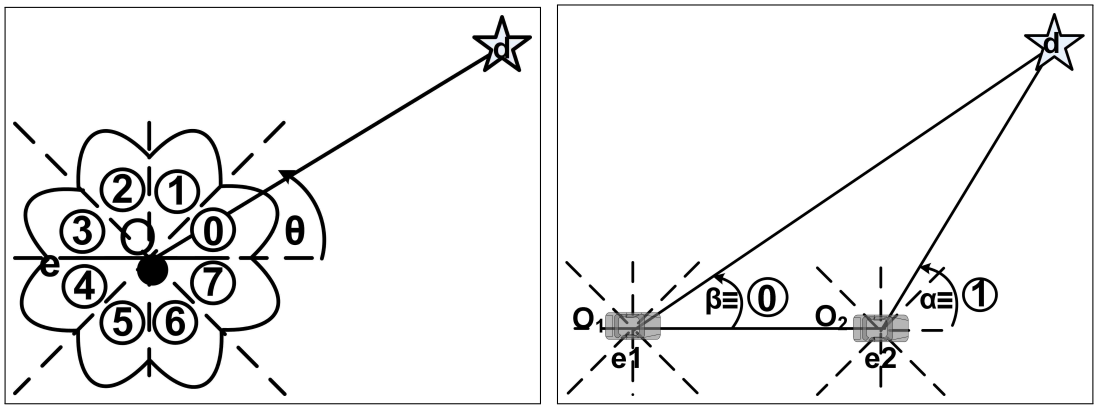


Figure 4.1 Index Structure for the Road Network with $N_d = 8$

line connecting the object's current position to its destination. For example, in Figure 4.2(a), object O's traveling direction is indicated by θ, and its destination is indicated by the star. After equally partitioning the 360 degree space into 8 directions, in this example, object O's traveling direction falls into the direction 0. This is treated as a hash value. The formal definition of an object's hash value is given in Definition 1.



(a) Object Traveling Direction w.r.t. Current Road Segment

(b) Objects with the Same Destination but Different Traveling Directions

Figure 4.2 Object Traveling Direction Calculation

Definition 1. *Let O be a moving object currently on road segment o_e^t with traveling destination o_{gd}. Let θ denote the angle between the horizontal line of the coordinate system and the line connecting o_{gd} and the midpoint of o_e^t. O's hash value $H(O)$ is defined by Equation (2), where N_d is the number of buckets in a hash table.*

$$H(O) = \lfloor \theta \; / \; \frac{360}{N_d} \rfloor \tag{2}$$

This strategy has to the following rationale: Consider the two objects O_1 and O_2 moving on the same road segment with the exact same destination depicted in Figure 4.2(b). These two obtain two different directions, 0 and 1, respectively, simply because of their minor difference in their current positions. From the querying perspective, these two objects are expected to be stored together since they are very likely to have similar or the same travel path. Therefore, to ensure the same hash value, the middle point of the road segment was used, instead of the current position, in computing the angle.

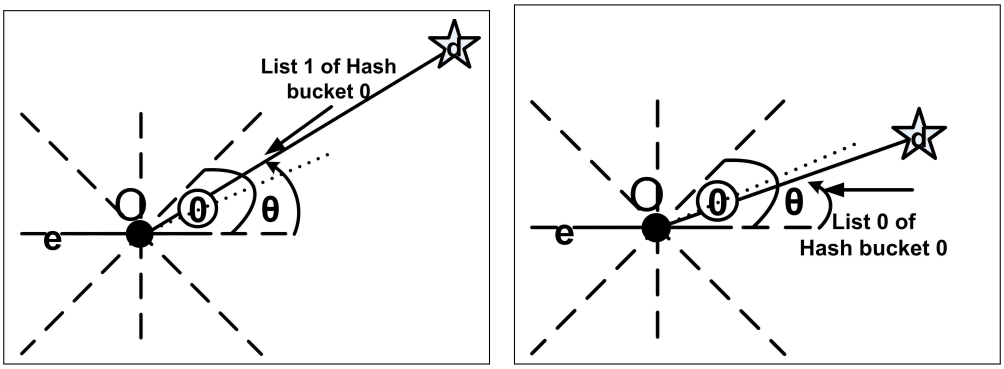
Once the hash bucket is selected, the object is stored in one of the linked lists. Selecting the corresponding linked list is based on the geographical direction in a finer granule. In this process, the central angle, considered for hash bucket selection, is further divided into two angles to maintain a linked list for each subdivision. The equation for selecting the linked list is shown in Equation (3).

$$list\ index = \begin{cases} 0 & \text{if } 0 \leq \theta - [\frac{360 \cdot H(O)}{N_d}] \leq \frac{360}{2 \cdot N_d}; \\ 1 & \text{if } \frac{360}{2 \cdot N_d} \leq \theta - [\frac{360 \cdot H(O)}{N_d}] \leq \frac{360}{N_d}; \end{cases} \tag{3}$$

Figure 4.3 shows the two vehicle destination positions, ending up in a different linked list of the same hash bucket. The dotted lines represent the margins of the linked list's area and dashed lines represent that of the hash bucket's area. Further, in each list, objects are arranged in a descending order of the Euclidean distance between their destinations and the mid point of the current edge. Such arrangement will help speed up queries as discussed in the next section.

Example

When θ is 30 degrees and N_d equals to 8, $H(O)$ will be $\lfloor 30 \,/\, (360/8) \rfloor$ = 0. Thus, applying θ, $H(O)$, and N_d in Equation (3), $\theta - [\frac{360 \cdot H(O)}{N_d}]$ becomes 30. Hence, the list index is 1. That means object O will be stored in the first linked list in the first hash table slot.



(a) (b)

Figure 4.3 Two Examples for a Linked List Selection

4.2. INSERTION, DELETION, AND UPDATE IN R^D-TREE

4.2.1. Insertion. When an object enters into the system, it sends an insertion request containing object's ID o_{id}, destination o_{gd}, current road segment o_e^t, current position (x_t, y_t) and velocity o_s. First, the R^D-tree is searched to locate the leaf node containing the current road segment. After that, a hash value is computed based on the road segment o_e^t and the destination o_{gd}, and the object's current information is inserted into the corresponding hash bucket linked to the leaf node.

4.2.2. Deletion. An object exiting the system sends a request to delete itself from the system. The request contains the object's ID o_{id}, previously reported road segment o_e^t, and the previously reported destination o_{gd}. Similar to the insertion process, first, the R^D-tree is searched to find the leaf node containing o_e^t that the object was on. Once the leaf node is located, the hash value is computed according

to Definition 1 using o_{gd}. Once the the corresponding hash bucket is located, the object is deleted from the R^D-tree.

4.2.3. Update. An object position update can be seen as a deletion followed by an insertion. The update process, however, is optimized with some modifications. An update request contains the object ID o_{id}, previous road segment o_e^t, and destination o_{gd}, current road segment o_e^{t+1}, current position (x_{t+1}, y_{t+1}) and velocity o_s^{t+1}, and the new destination o_{gd}^* (if changed).

First, similar to the way the object was searched in the delete operation, the object is searched in the R^D-tree. The update procedure is as follows:

- If the object's previous and current road segment as well as the travel desitnations are the same, i.e., $(o_e^t == o_e^{t+1}) \wedge (o_{gd} == o_{gd}^*)$, only the object's new position (x_{t+1}, y_{t+1}) and the new velocity o_s^{t+1} information need to be updated in the hash bucket.
- If the above condition is not present, the object's old information is deleted and followed up with the following insertion steps:
  - Check if the object is still on the same road segment but with a new destination, i.e., $(o_e^t == o_e^{t+1}) \wedge (o_{gd} \neq o_{gd}^*)$. If so, the update is conducted under the same leaf node.
  - Otherwise, if $(o_e^t \neq o_e^{t+1})$, R^D-tree is traversed to locate the leaf node containing o_e^{t+1}. Once found, a hash value is computed based on the new road segment o_e^{t+1} and destination o_{gd}. The object current information is inserted to the corresponding hash bucket linked to the leaf node. Note that, here o_{gd}^* is considered instead of o_{gd}, if the travel destination has changed.

4.3. QUERYING R^D-TREE

R^D-tree can support traditional types of queries, such as RQ and KNNQ. Concerning the road network constraint, we refine the range query to the line query. Instead of locating objects in a certain rectangular or circular range, the line query estimates the moving objects which may enter the query road segment (i.e., a line) at the query time. The motivation of such line query is that people are usually more interested in the traffic condition of a particular road that they need to pass by, rather than the traffic condition of a wide range which may contain

roads irrelevant to the query issuers' traveling routes. The formal definition of the predictive line query and the detailed query algorithms are discussed in Chapter 5 and 6.

4.4. SUMMARY

This chapter proposed a novel, efficient, and effective indexing structure, namely R^D-tree, that supports queries on mobile objects under road network constraints. The tree comprised of an R*-tree and a hash table in which the objects are stored depending on their geographical moving direction. The proposed indexing structure facilitates efficient query processing on objects' predictive information. Storing based on the moving direction promotes moving objects with similar traveling directions to be stored together in separate hash buckets. Thus, when the predictive queries are processed the required information can be accessed efficiently.

5. PREDICTIVE LINE QUERIES : SNAPSHOT QUERY

This chapter presents an advanced and a novel query type named *PLQs* that predicts traffic jams ahead of time. PLQs help commuters plan their trips more effectively and efficiently, and enhance their location-based experience. Specifically, this chapter presents one of the two versions of PLQs – the *SPLQ* – proposed in this work; the other version, the *CPLQ*, is presented in Chapter 6. Furthermore, this chapter presents three query algorithms for the SPLQ with an increasing number of heuristics and, hence, a coinciding increase in pruning power.

Existing traffic related queries are mostly queries on real-time traffic information, which is not sufficient to help commuters plan their trips ahead of time. For example, by the time traffic information is received, it may be too late for a commuter to select an alternative route to avoid a newly formed traffic jam on his/her current travel route.

As an example, consider Figure 5.1, where a user (commuter) is interested in the traffic condition of the highlighted road segment in the near future. A predictive query result will help the user to make adjustments on his/her travel plan based on impending traffic conditions. The proposed service is capable of answering queries like ‘‘`What will be the traffic condition on Highway 44 near St.Louis in half an hour"?`

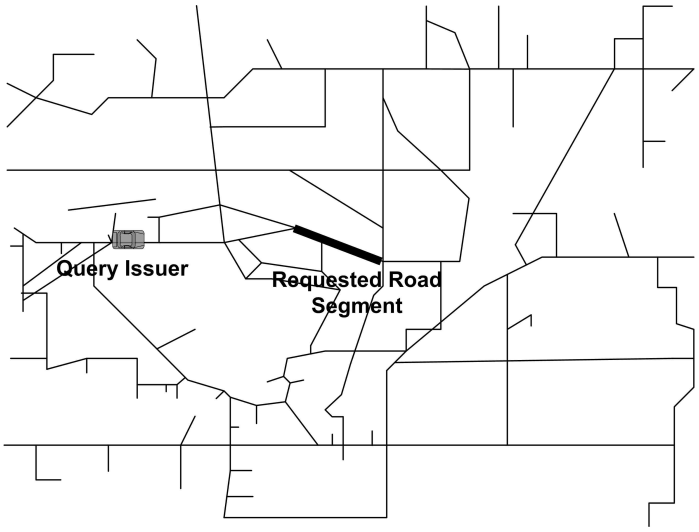


Figure 5.1 An Example of a Predictive Line Query (PLQ)

The formal definition of the PLQ is found in Chapter 5.1. Chapter 5.2, 5.3, and 5.4 present the three algorithms proposed for CPLQ, respectively. These query algorithms are supported by the R^D-tree[3] that was designed to support predictive queries under road network constraints. A cost analysis of the algorithms can be found in Chapter 5.5 and, finally, Chapter 5.6 presents the performance study of these algorithms.

5.1. DEFINITIONS

Definition 2. *[Predictive Line Query (PLQ)] A Predictive Line Query retrieves all moving objects which will be on the query road segment e_q at the query time t_q, where $t_q > t_c$; t_c is the current time at which the query is issued.*

The Predictive Line Query (PLQ) is a one-time snapshot query. It does not consider possible changes of the predicted traffic condition when the query issuer moves closer to the querying road. In order to provide timely and up-to-date information to the query issuer, we model moving objects as a linear function of time, which has proven effective in many prior works [5, 6, 10–12]. Vehicles are assumed to report their locations and velocities to the server whenever there is a significant change of their moving functions.

Definition 3. *[Ring Query (R^0Q)] A Ring Query $R^0Q = (e_q, r_1, r_2)$ retrieves moving objects whose current locations are in the ring defined by the concentric circles with the mid point of the query road segment e_q as center and r_1 and r_2 as radii, where $r_1 = v_{min} \cdot (t_q - t_c)$ and $r_2 = v_{max} \cdot (t_q - t_c)$. t_q is the query request time and t_c is the query issuing time; $(t_q > t_c)$.*

A Graphical explanation of the ring query is illustrated in Figure 5.2. Its formal definition is given in Definition 3. The ring query aims to define a more restricted search range than the general rectangular or circular *Range Queries (RQs)*, so that fewer intermediate results are generated. The basic idea is to find the current positions of the furthest vehicle and closest vehicle which may enter the query road segment e_q at the query time t_q, and then use their current distance to the e_q to define concentric circles as the query ring. More specifically, the furthest candidate vehicle is currently at a distance $v_{max} \cdot (t_q - t_c)$ from e_q, while the closest candidate vehicle is currently moving at a $v_{min} \cdot (t_q - t_c)$ distance, both moving towards e_q. Here, v_{max} and v_{min} are the maximum and minimum speed limits

[3] See Chapter 4 for a detailed discussion.

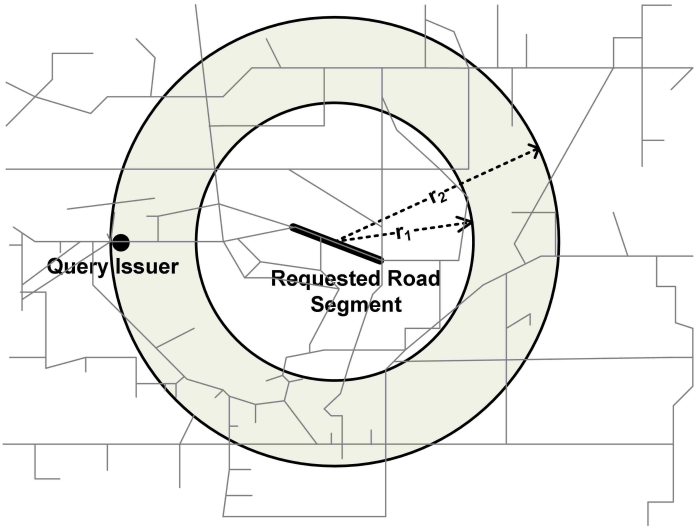


Figure 5.2 The Initial Filtering with a Ring Query

respectively. The area covered by the R^{0}Q is $\pi \cdot (v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2$, while that of the RQ is $\pi \cdot v_{max}^2 \cdot (t_q - t_c)^2$. The smaller range, given by the R^{0}Q, reduces the number of objects that needs to be accessed in the index.

5.2. BASIC ALGORITHM

The first PLQ algorithm is the basic algorithm. It consists of two phases: the filtering phase and the refining phase. The filtering phase retrieves candidate objects using a *Ring Query (R^0Q)*. The second phase refines the results by estimating the candidate objects' traveling routes.

Given a PLQ, the basic algorithm first computes its corresponding ring query. Once the query ring is determined, a search is initiated in the R^D-tree to find the road segments that intersect with the query ring. For each such road segment, its hash table is checked to find objects currently moving on it. In fact, it is not necessary to access the entire hash table, but only access the hash buckets which contain objects with traveling directions toward the query road segment. Afterwards, the following calculation is performed.

1. First compute the angle θ_q between the horizontal line and the line connecting the mid points of the current road segment o_e^t (Refer Chapter 4.1) and the query road segment e_q.

2. Then, plug θ_q Equation (2) to obtain a hash value H_q.

In this version of the algorithm, both linked lists of the obtained hash buckets are accessed. Figure 5.3 illustrates the idea, where the hash value is 0. As evident in Figure 5.3, the query θ_q is located at the border of the hash bucket 0. To obtain more accurate query results, one more bucket, adjacent to H_q, is considered when θ_q is close to the border with less than θ_x degree (which is set at a 15 degree as default). In the example, both buckets 0 and 1 are considered in the query, which gives four linked lists: Linked lists 0 and 1 of both buckets.

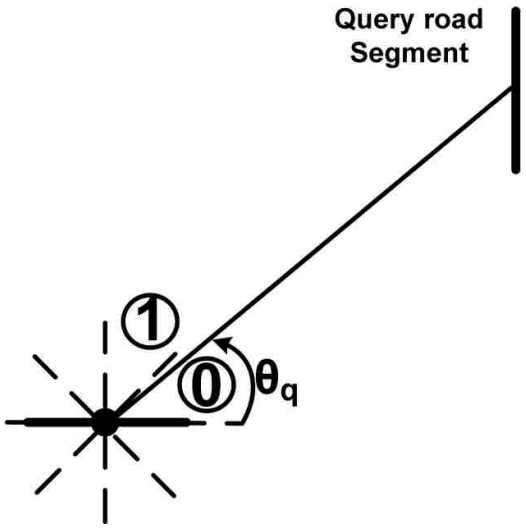


Figure 5.3 Marginal Query Angle Selection in Basic Algorithm

After obtaining a set of candidate objects from the ring query, the second phase of the query processing eliminates objects which cannot possibly enter the query road segment e_q by examining objects' tentative traveling routes. The shortest route of an object's destination is computed when an object initially registers in the system or issues an update of its destination. During the query, we check to see if the shortest route of the candidate object contains the query road segment at the query time. If so, the candidate object will be included in the final result. It is worth noting that being a prediction, the query results may not be 100% accurate.

The query algorithm is summarized in Algorithm 1. Lines 5-11 are the first phase. When a user (moving object) sends a query request, he/she does not need to always specify the query time. The algorithm estimates the time taken for the query issuer to enter the query road segment as the query time t_q when it is not provided (line 2). The function 'getDirection()' returns two consecutive

Algorithm 1 Basic Algorithm for Predictive Line Query
Inputs: $(x_t, y_t)_q$ – current location of the query issuer, e_q – query road segment, t_q – query time, t_c – Query issuing time
Output: $Result$ – a set of objects that may be on e_q at t_q

```
1: if t_q =NULL then
2:    t_q = timeToEnter(v, e, t_c)
3: end if
4: Result = ∅
5: Edges = RingQuery(e_q, v_min · (t_q − t_c), v_max · (t_q − t_c), )
6: if (Edges <> null) then
7:    for each e_i ∈ Edges do
8:       Direction = getDirection(e_i, e_q)
9:       Result = Result ⋃ getVehicles(e_i, Direction)
10:   end for
11: end if
12: for each object o_i in Result do
13:   if not getVehiclesContainPaths(e_q, o_i, t_q) then
14:      Result = Result − {o_i}
15:   end if
16: end for
```

hash buckets with the hash value of the direction to the query road segment. The function 'getVehicles()' checks the hash table of the particular edge and only retrieves moving objects with the hash values given by 'getDirection()'. Candidate objects are stored in a set $Result$. Line 12-16 are the second phase. The estimated traveling route of each candidate object in $Result$ is checked. If the traveling route does not contain the query road segment at the query time, the object will be removed from $Result$.

5.3. ENHANCED ALGORITHM

The enhanced algorithm also consists of two phases as the basic algorithm. The improvement is at the 'getVehicles()' function in Algorithm 1, where the potential destinations of objects are considered for pruning purposes. Observe the example shown in Figure 5.4. The road segment AB is a candidate road segment retrieved form the ring query in Figure 5.2. O_1 and O_2 are two objects whose destinations are d_1 and d_2 respectively. Remaining traveling routes of both objects from point B onwards are shown as bold lines. As shown, the route of O_1 ends before the querying road segment, which means that O_1 will not pass by

the querying road segment unless it changes its destination later on. Based on currently available information, the query results should only include O_2.

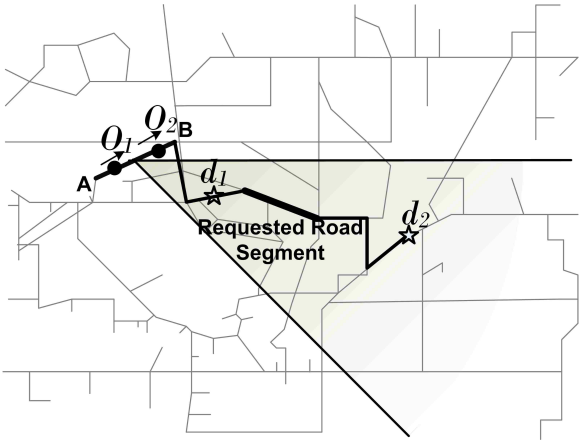


Figure 5.4 An Early-Destination-Pruning Heuristic Example

The above observation leads to the conclusion that it may not be necessary to examine all objects in the linked list. Thus, objects are stored in descending order according to the distance between current edge and their destinations. In the above example, object O_2 will be stored before O_1 in the linked list. When accessing the list of vehicles, the destination of the vehicles is also considered. The search stops when reaching the object whose destination is earlier than the querying road segment.

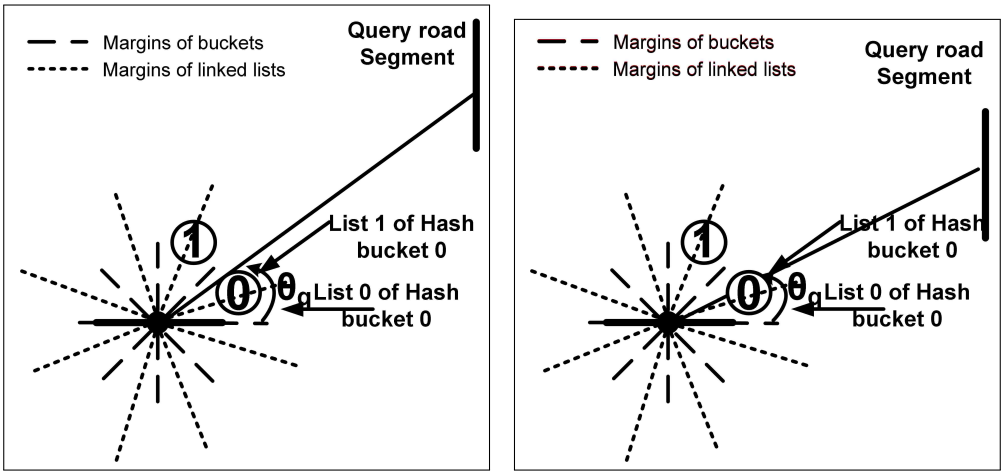
5.4. COMPREHENSIVE ALGORITHM

The comprehensive algorithm aims to further improve the accuracy of the query results obtained by the enhanced algorithm. The idea is to choose a more confined set of objects by carefully selecting traveling directions towards the querying road segment. In the previously discussed basic and enhanced algorithms, the traveling directions, i.e., the number of hash buckets, being considered is either one or two according to the closeness (15 degrees in our experiments) to the margins of an area. Thus, the total area considered is the area made by the central angle $2 \cdot (360/N_d)$ or $(360/N_d)$.

The comprehensive algorithm introduces a method which considers an area equal to that of exactly one central angle $360/N_d$. This restricted area reduces the number of individual vehicles considered compared to that of the other two algorithms of the R^D-tree. Besides, in this way, the area considered is nearly

symmetric on the line to the query road segment. Thus, chances of neglecting possible candidates are less. As the end result, this method will result in accurate results with lesser number of page accesses.

Execution of the ring query in the comprehensive algorithm is the same as that of the basic algorithm. For each road segment retrieved from the ring query, relevant hash value H_q is obtained from Equation (1). Based on this hash value, the relevant linked list index is found from Equation (3). The second linked list is the list which is closest to the first list. In this case, the second linked list could either be from a bucket adjacent to H_q or from the same bucket, depending on which linked list covers the closest central angle. Figure 5.5(a) and 5.5(b) show examples of obtaining the second list from the same hash bucket and an adjacent hash bucket respectively.



(a) Both Linked Lists from Same Hash Bucket

(b) Linked lists from Two Adjacent Hash Buckets

Figure 5.5 Two Examples for Two Linked List Selections

Vehicles are retrieved from the selected two sorted linked lists considering the remaining distance to the destination and to the querying road segment (the same way as the enhanced algorithm did). Retrieved vehicles are then applied to the second phase of the query algorithm as explained under the basic algorithm.

5.5. QUERY COST ANALYSIS

In this section, we analyze the query cost in terms of the number of disk page accesses. For clarity of the presentation, Table 5.1 summarizes the notations in the following discussion.

Table 5.1 Terms and Their Descriptions

Term	Description
v_{max}	Maximum speed limit of the entire map
v_{min}	Minimum speed limit of the entire map
t_q	Querying time
t_c	Current time
$count_{e_ttotal}$	Total number of road segments
$count_{e_page}$	Number of road segments per disk page
$Area_{map}$	Total area of the map
$count_{moPerRdSeg}$	Average number of vehicles moving on a road segment
$count_{mo-page}$	Maximum number of vehicles per disk page
N_d	Number of hash buckets

Given a query, the disk access cost includes two aspects: (1) the number of disk pages ($Count_q$) visited to find the road segments covered by the ring query; (2) the number of disk pages ($Count_v$) visited to find the vehicles that may be the query answers.

$Count_q$ is determined by the area covered by the query ring, which is:

$$\pi(v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2 \tag{4}$$

Assuming that the road segments are distributed evenly throughout the entire area, the average number of road segments per unit area $count_{e-unitArea}$ is $count_{e-total}$ / $Area_{map}$. Thus, the average number of road segments in ring area is:

$$\frac{count_{e-total} \cdot \pi(v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2}{Area_{map}}. \tag{5}$$

Let the maximum number of road segments per disk page, a system design parameter, be $count_{e-page}$. Then the number of disk pages required for road segments in the ring area is:

$$Count_q = \frac{count_{e-total} \cdot \pi(v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2}{Area_{map} \cdot count_{e-page}}. \tag{6}$$

For the second part of the query cost, vehicles are assumed to be distributed uniformly throughout the road segments. Let $count_{moPerRdSeg}$ denote the average number of vehicles moving on a road segment. The number of vehicles moving on

road segments covered by the ring area can be estimated as follows:

$$\frac{count_{e-total} \cdot \pi(v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2 \cdot count_{moPerRdSeg}}{Area_{map}}. \quad (7)$$

The total number of hash buckets is N_d. Since only one bucket in the hash table is considered during one query process, the maximum number of disk pages for vehicles are expressed as:

$$Count_v = \frac{count_{e-total} \cdot \pi(v_{max}^2 - v_{min}^2) \cdot (t_q - t_c)^2 \cdot count_{moPerRdSeg}}{Area_{map} \cdot count_{mo-page} \cdot N_d}. \quad (8)$$

At the end, the total number of disk page accesses can be estimated by summing up the cost in (6) and (8):

$$Cost_{disk} = Count_q + Count_v \quad (9)$$

5.6. PERFORMANCE STUDY

Experiments were conducted on moving object data sets generated by the Brinkhoff's generator [64]. Real road maps of US states were provided to the generator. The number of moving objects ranged from 10K to 100K. The object speeds ranges from 30mph to 60mph. The California state map was used as the default, which contains 53, 112 road segments. We generated predictive queries by randomly selecting query road segment and predictive time length.

Performance of our proposed R^D-tree with the comprehensive query algorithm was compared with recent related work, i.e., the R-TPR$\pm$-tree [36] which supports predictive queries on moving objects under road network constraints. In addition, performance of all three algorithms – Basic, Enhanced, and Comprehensive – were also compared in order to study the effect of the individual improvements. For notational convenience, the Basic, Enhanced, and Comprehensive algorithms will be referred to as R^{D}B-tree, R^{D}E-tree, and R^{D}C-tree, respectively, throughout the rest of the discussion. .

All four algorithms were evaluated by varying three parameters: the number of moving objects, the predictive time length, and the road topology. The performance was measured in terms of I/O cost (the number of disk-page accesses), CPU time, and query accuracy. CPU time does not include initial bulk loading of the road map or objects, but considers only the query processing time. Query accuracy was examined by comparing the number of objects in the predictive query results

with the actual number of objects on the query road segment at the query time. Each test case was run for 250 queries, and the average cost is reported. Parameters and their values are summarized in Table 5.2, where default values are highlighted in bold.

Table 5.2 Simulation Parameters and Their Values for Snapshot PLQ Algorithm

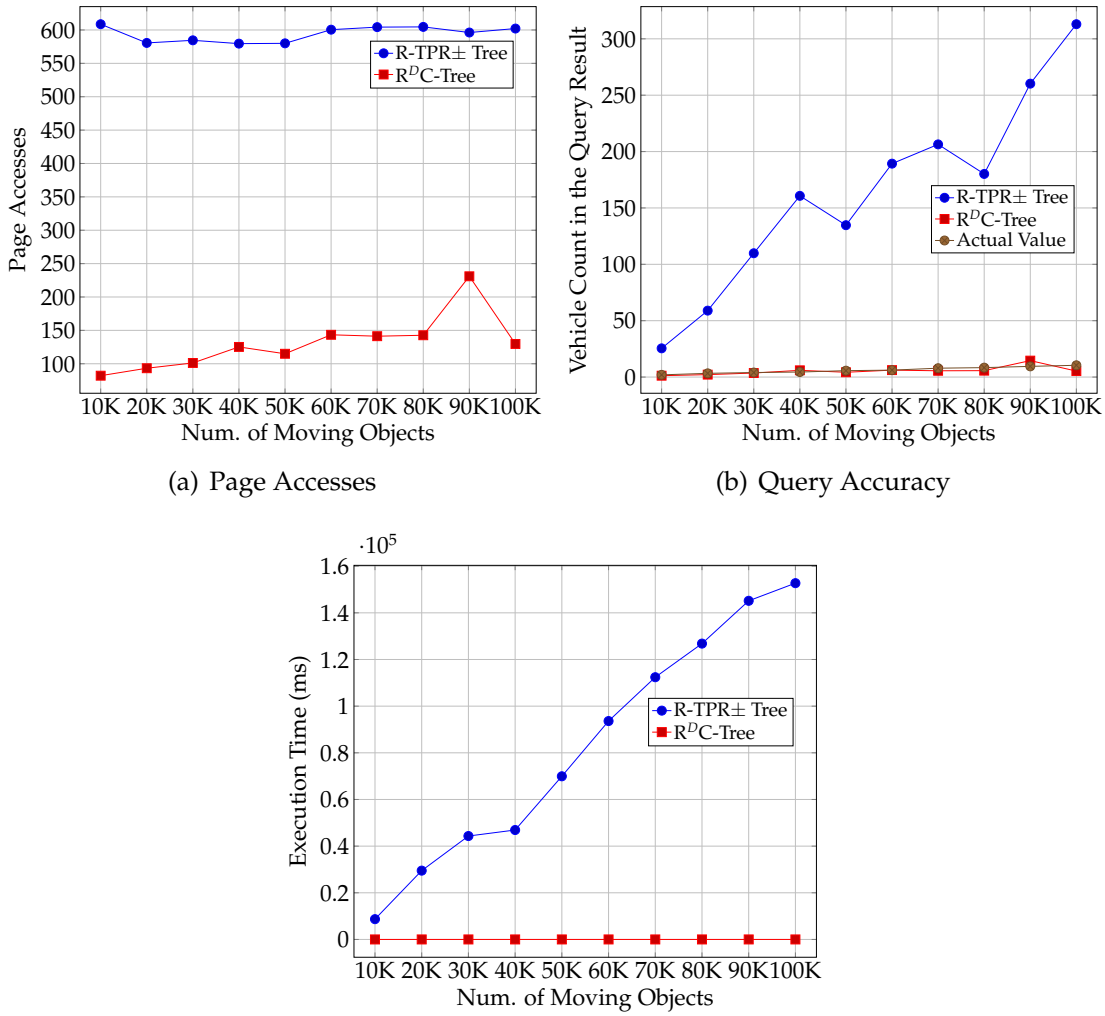
Parameters	Values
number of moving objects	10K, 20K, ..., **50K**, 60K, ..., 100K
predictive time length (in minutes)	10, 20, **30**, 40, 50, 60
road maps	CO, AR, NM, **CA (California)**

5.6.1. Effect of the Number of Moving Objects. Both the R-TPR±-tree and the R^D-tree were tested for different sizes of moving object data sets generated using the default road map, the CA map. Figure 5.6 shows the results of the R-TPR±-tree and the R^D-tree with the comprehensive algorithm (the R^DC-tree), while Figure 5.7 compares the R^{D}B-tree, the R^{D}E-tree, and the R^{D}C-tree. R^{D}C-tree outperforms in all three performance metrics.

Figure 5.6(a) shows that the proposed R^DC-tree requires about slightly more than a 50 % less page accesses than the R-TPR±-tree. The reasons are mainly three-fold. First, the R^DC-tree uses the ring query to retrieve candidate objects which are usually less than objects retrieved using the range query. Second, the R^DC-tree arranges objects according to their traveling directions. Finally, the objects are ordered according to the distance of the destination from the edge. Thus, the R^{D}C-tree greatly reduces unnecessary page accesses. As a result, the query only needs to check objects that probably will be on the query road segment, i.e. those objects heading the query road segment.

With respect to the accuracy, the R^DC-tree also significantly outperforms the R-TPR±-tree as shown in Figure 5.6(b). The number of query results returned by the R^DC-tree is very close to the actual number of objects on the query road segment. However, that of the R-TPR±-tree is a considerable diversion from the correct result. It should also be noted that in most of the test cases, the accuracy of the R^{D}C-tree is slightly less than the actual result. The reason for this kind of behavior is due to the restricted number of hash buckets considered in the query processing.

The powerfulness of the R^DC-tree pruning techniques is more visible in Figure 5.6(c). The graph shows the performance in terms of CPU time. Note that



(a) Page Accesses

(b) Query Accuracy

(c) Execution Time

Figure 5.6 Query Performance of R^{D}C-tree and R-TPR±-tree with Varying Number of Moving Objects

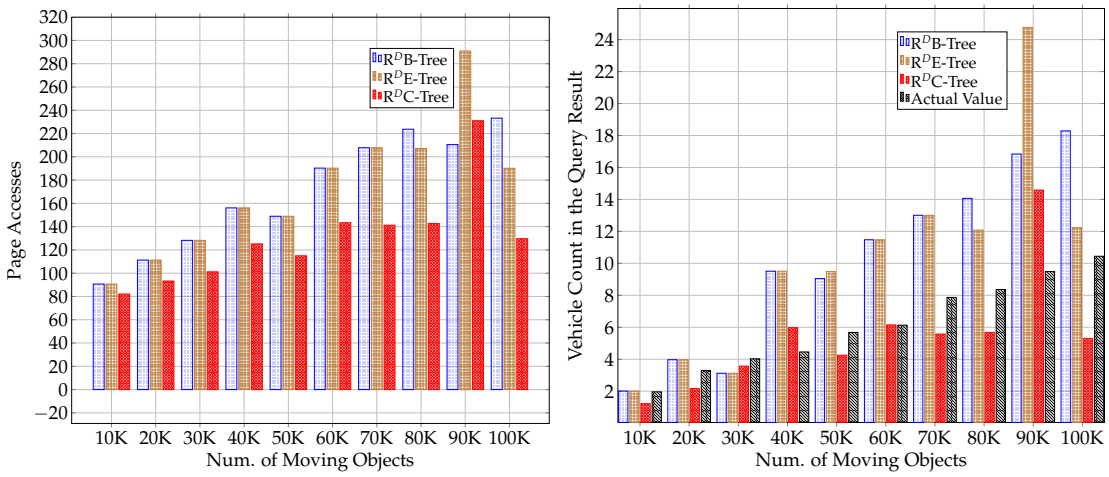
the execution time is in logarithmic scale. Since the R^DC-tree prunes the search space more efficiently, the number of individual edges and vehicles considered are much less than that in the R-TPR±-tree, which leads to less CPU time.

The performance of three query algorithms on the R^D-tree is shown in Figure 5.7. In particular, Figure 5.7(a) depicts the comparison in terms of page accesses. The R^{D}C-tree yields the smallest number of page accesses compared to the other two algorithms. The other two algorithms consume a similar number of page accesses when the number of vehicles are less than 70k but diverge afterwards.

When the number of vehicles is small, on average, the number of vehicles per hash bucket is also small. The performance difference between the R^{D}B-tree and

R^DE-tree can be seen only when vehicles with destinations beyond the query road segment are stored in multiple disk pages. In other words, the early-destination pruning metric helps reduce disk page accesses when vehicles being pruned are stored in different disk pages.

Figure 5.7(b) compares the predicted number of vehicles in the query results obtained from the three query algorithms with the actual number of vehicles on the road segment at the query time. We can observe that the predicted number by the R^DC-tree is always closest to the actual value, while the R^DB-tree and R^DE-tree perform similarly in most cases. This again indicates the superiority of the R^DC-tree.



(a) Page Accesses

(b) Query Accuracy

(c) Execution Time

Figure 5.7 Query Performance of R^D-tree Query Algorithms with Varying Number of Moving Objects

In terms of the execution time, the R^{D}E-tree and the R^{D}C-tree are very similar while the R^{D}B-tree is slowest. This is mainly attributed to the sorting feature introduced in the enhanced query algorithm that prunes vehicles which cannot reach the query segment.

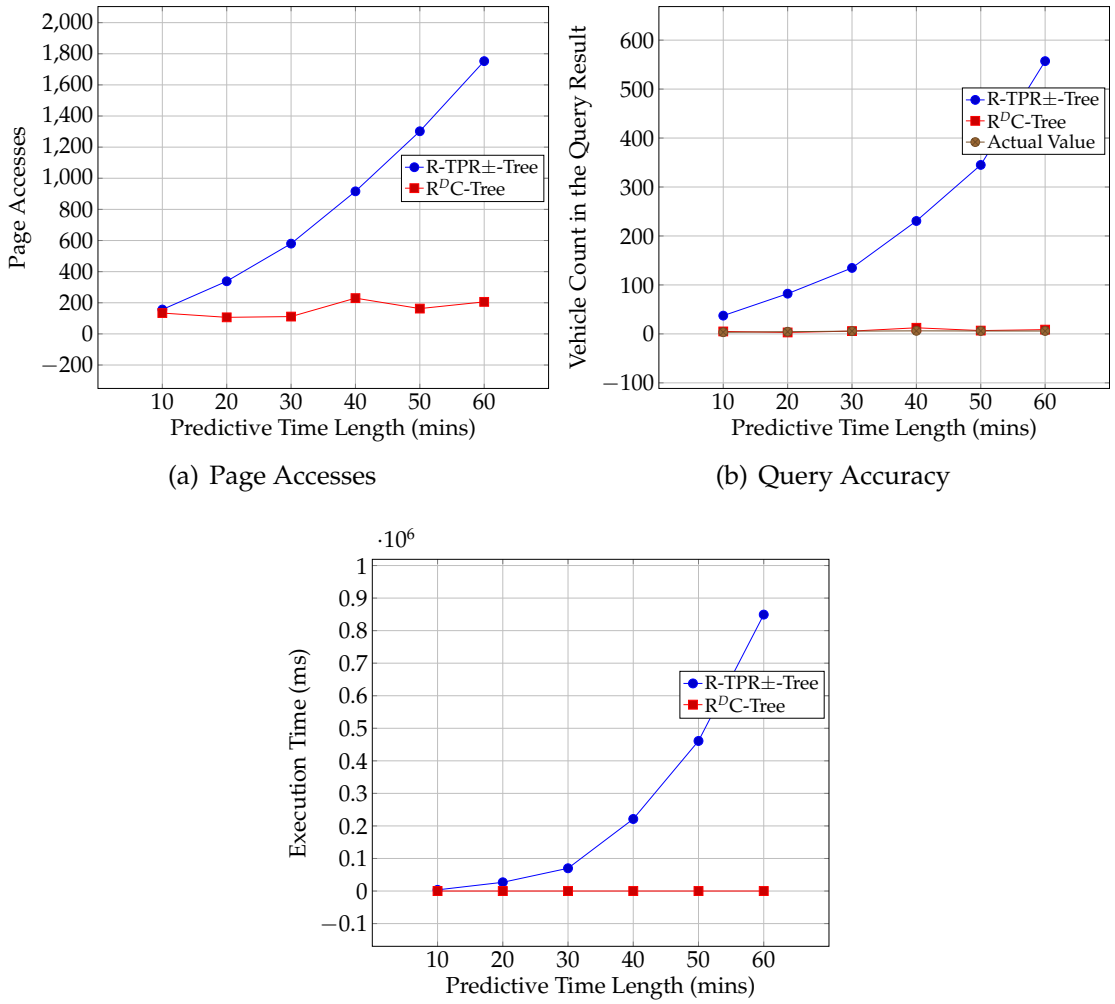
5.6.2. Effect of the Predictive Time Length. The effect of the predictive time length was studied by varying it from 10 minutes to 60 minutes. Figure 5.8 shows the performance comparison of the R-TPR$\pm$-tree with the R^DC-tree. As shown in Figure 5.8(a), both trees access more disk pages when the time length increases. This is because the longer the time to look into the future, the bigger the query range will be, which results in more page accesses. We also observed that the query cost using the R^DC-tree only slightly increases whereas the query cost using the R-TPR$\pm$-tree increases drastically.

The advantage of the use of ring query by the R^D-tree is more prominent when the query time length is longer. The area of a query ring increases less significantly than the area of a query circle. Therefore, the number of objects need to be retrieved in the R^DC-tree also increases very slowly.

Figure 5.8(b) compares the accuracy of the R-TPR$\pm$-tree and R^DC-tree with the actual query result. The results obtained by the R^DC-tree query algorithm are very close to the actual values, and the accuracy is relatively stable for different query time lengths. Any minor inaccuracy may be caused by the difference of the estimated traveling routes and the actual routes taken by some objects.

The accuracy in the R-TPR$\pm$-tree is much lower compared to the R^D-tree. Especially when the predictive time length is longer, e.g., 60 minutes, the R-TPR$\pm$-tree query algorithm returns a number more than 10 times the actual number of objects on the query road segment. The R-TPR$\pm$-tree query algorithm works well when the predictive time length is extremely short so that the query range mainly covers road segments next to the query road segment, and objects in the query range can at most move to the next road segment at the query time. When the predictive time length is long, such estimation introduces lots of errors.

The execution time for the R-TPR$\pm$-tree increases gradually with the predictive time length. As shown in Figure 5.8(c), the execution time of R^{D}C-tree is relatively steady for all predictive time lengths. Moreover, the R^{D}C-tree is about 250 times faster than the R-TPR$\pm$-tree when the predictive time length is 10 minutes. The performance gap between the two algorithms is further enlarged with the increase of the predictive time length.



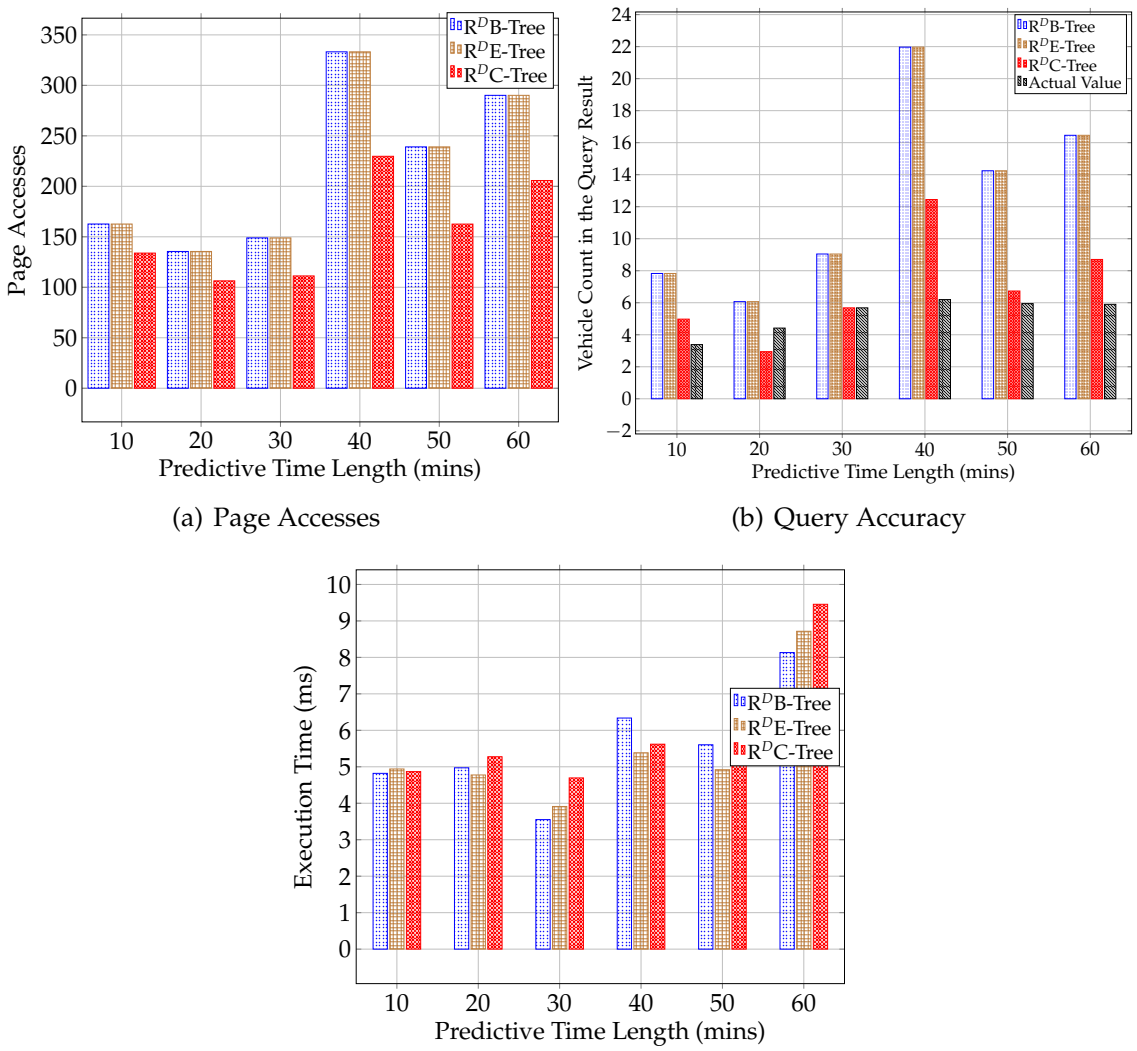
(a) Page Accesses

(b) Query Accuracy

(c) Execution Time

Figure 5.8 Query Performance of R^{D}C-tree and R-TPR±-tree with Varying Predictive Time Length

Next, we compare the performance of our three query algorithm (see Figure 5.9). Overall, the R^{D}C-tree performs best. The R^{D}C-tree has the fewest page accesses and the best accuracy. However, it does require slightly more execution time due to the complexity of the bucket selection algorithm. The R^{D}B-tree and the R^{D}E-tree behave very similarly in all cases. As previously discussed in Section 5.6.1, 50 K moving objects do not represent a large enough sample to illustrate the advantage of the sorted list in the enhanced algorithm. In addition, we can see that



(a) Page Accesses

(b) Query Accuracy

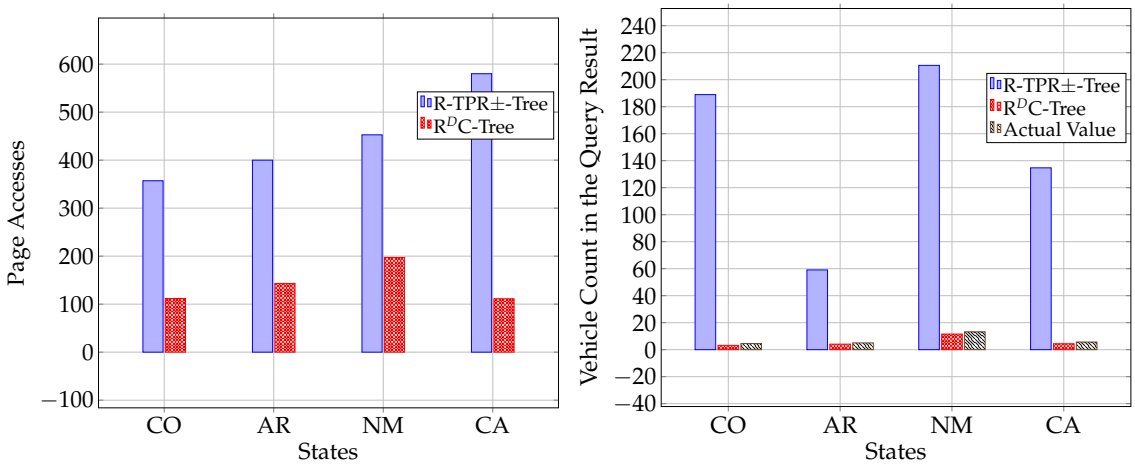
(c) Execution Time

Figure 5.9 Query Performance of R^D-tree Query Algorithms with Varying Predictive Time Length

the cost increases slightly in all algorithms. This is mainly due to the increment of the query area caused by the increase of predictive time length.

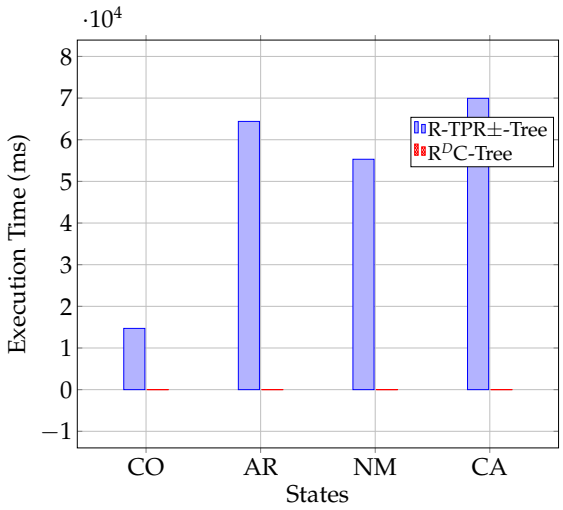
5.6.3. Effect of the Road Topology. The effect of road topology was evaluated by testing different road maps: Colorado (CO), Arkansas (AR), New Mexico (NM), and California (CA). The average road segment length in these maps is different, which are 0.152 miles in CO, 0.101 miles in AR, 0.92 miles in NM, and 0.81 miles in CA. Figure 5.10 shows the results for the R^DC-tree and the R-TPR±-tree. Observe that the R^DC-tree significantly outperforms the R-TPR±-tree

in all cases. Moreover, the performance of the R^D-tree is relatively independent of the road topology, while the R-TPR±-tree performs worse when the road segment becomes shorter. In the R^D-tree, longer road segments result in more objects per hash bucket, and hence slightly affects the performance.



(a) Page Accesses

(b) Query Accuracy



(c) Execution Time

Figure 5.10 Query Performance of R^{D}C-tree and R-TPR±-tree for Different Road Topologies

In contrast, the R-TPR±-tree performs better for maps with lengthier road segments. The possible reason is that each TPR-tree in the R-TPR±-tree groups objects better when the road segment is longer. As shown in Figure 5.10(b), the R-TPR±-tree contains a significant amount of false positives in the query result. The

actual value is even 75 % less than that of R-TPR±-tree. As shown in Figure 5.10(c), the execution time of the R-TPR±-tree is in the range of 10 k to 100 k milliseconds while that of R^D-tree ranges from 1 to 10 milliseconds.

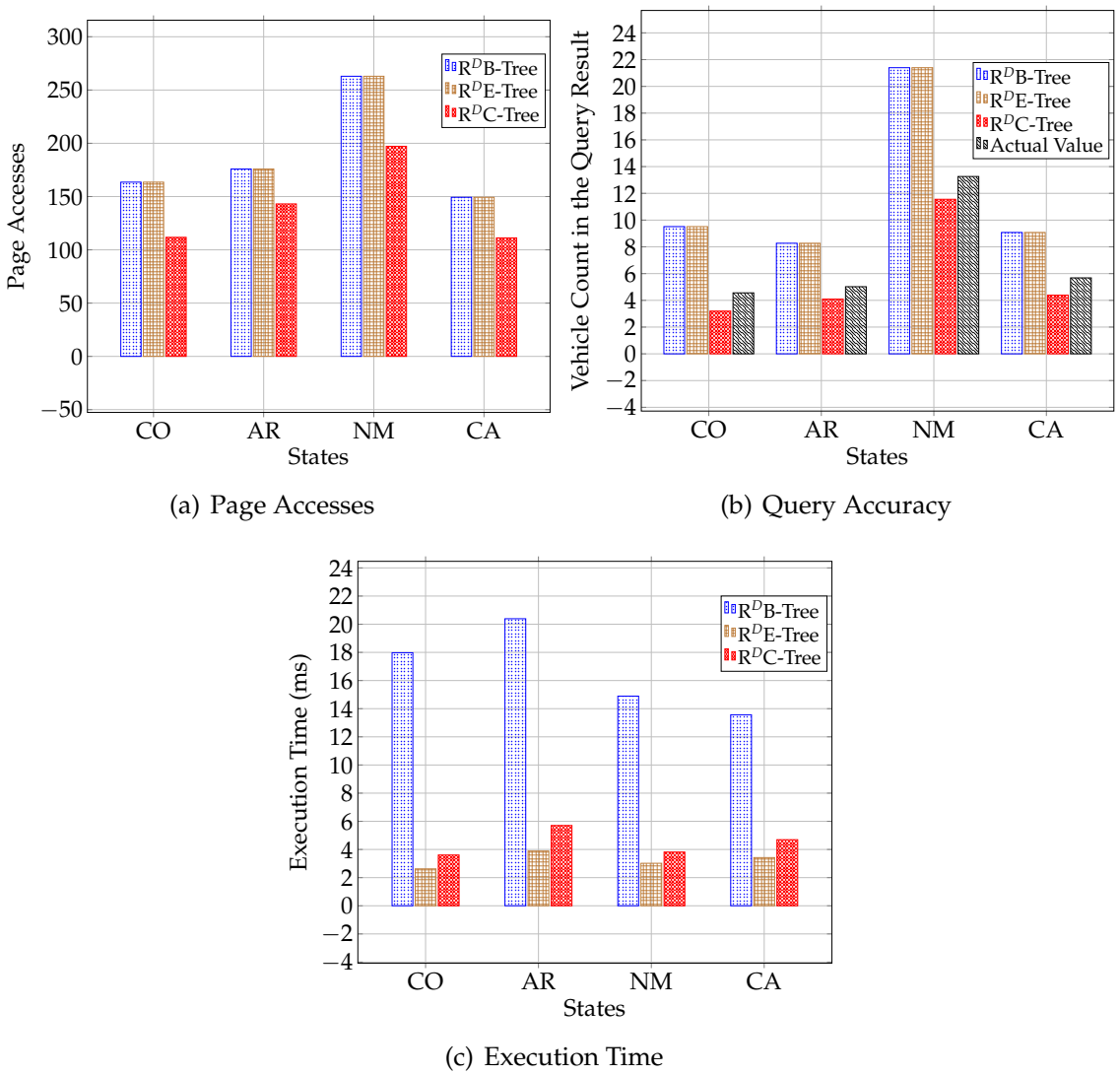
This significant increase of execution time in the R-TPR± is due to the individual consideration of road segment, obtained from the range query specified by a circle. Since, the ring query applied in the R^DC-tree reduces the number of edges, the number of individually considered edges are also less and hence the processing power reduces significantly.

Figure 5.11 shows the performance of all three versions of the R^D-tree query algorithms. Again the R^DB-tree and the R^DE-tree perform similarly in terms of page accesses and query accuracy due to the same reason as previously discussed. However, in terms of execution time, the R^DE-tree is much faster than the R^DB-tree due to the use of sorting list for pruning vehicles with early destinations. Since the R^DC-tree inherits all the pruning power of the other two versions, it achieves overall best performance.

5.6.4. Update Cost. We also examined the update cost in the tree versions of the R^D-tree and the R-TPR±-tree. Figure 5.12 shows the average cost after all objects have been updated once. In the experiment, 50 pages of buffer was used. We can see that all three versions of R^D-tree gives the same update cost. That is because the update algorithms in the R^D-tree do not depend on the differences of the query algorithm. Additionally, R-TPR±-tree also behaves similar to R^D-tree. This is possibly due to the similarity of the update algorithms. In both trees, the update cost includes two portions. One is for the search in the R*-tree to locate the road segment, and the other is for the search in either the hash table in the R^D-tree or the TPR-tree in the R-TPR±-tree to find the actual object.

5.7. SUMMARY

This chapter presented a novel query type named *PLQs* that predicts traffic of a user given road segment and three query algorithms for snapshot PLQ processing. Each algorithm has two phases: the filtering phase and the refining phase. All three algorithms share the refining phase. The differences come in the filtering phase. The filtering phase utilizes a novel concept to efficiently extract the road segments from the R^D-tree that might contain the objects which will be on the querying road segment. The algorithms capitalize R^D-tree's key feature - storing objects moving towards the same destination- to get the objects from the



(a) Page Accesses

(b) Query Accuracy

(c) Execution Time

Figure 5.11 Query Performance of R^D-tree Query Algorithms for Different of Road Topologies

extracted edges. In fact, the algorithms finds the geographical direction of the querying road segment for each extracted road segment and select objects from the R^D-tree's hash bucket that would be traveling with the same direction. The three algorithms selects objects from hash bucket differently. The first algorithm finds out the best matching hash bucket and get objects in that hash bucket. The second algorithm uses the same hash bucket selection with a sorted list of objects according to the objects destination. It then selects objects only whose destination is after the querying road segment. The third algorithm selects the best matching two object lists which can be from one or two hash buckets. The refinement

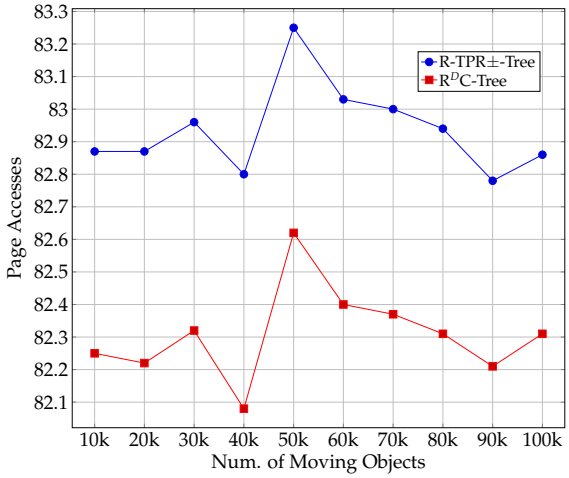


Figure 5.12 Update Cost

phase examines the objects traveling routes and refines the objects retrieved in the filtering phase.

Gathered experimental results shows that the ring and the destination based grouping has improved the performance of the algorithms compared to that of range query based algorithm in RTPR-±-tree.

6. PREDICTIVE LINE QUERIES : CONTINUOUS QUERY

Route calculation and planning based on *current traffic condition*, e.g. traffic jam prediction based on Snapshot Predictive Line Query (SPLQ), while enhances commuters' driving experience, may not be optimal due to the dynamic nature of the query objects and their influence over each other. This chapter proposes a solution to this problem by constructing a *continuous* traffic prediction system. The problem is formalized as a *CPLQ*. The primary contributions of this chapter are:

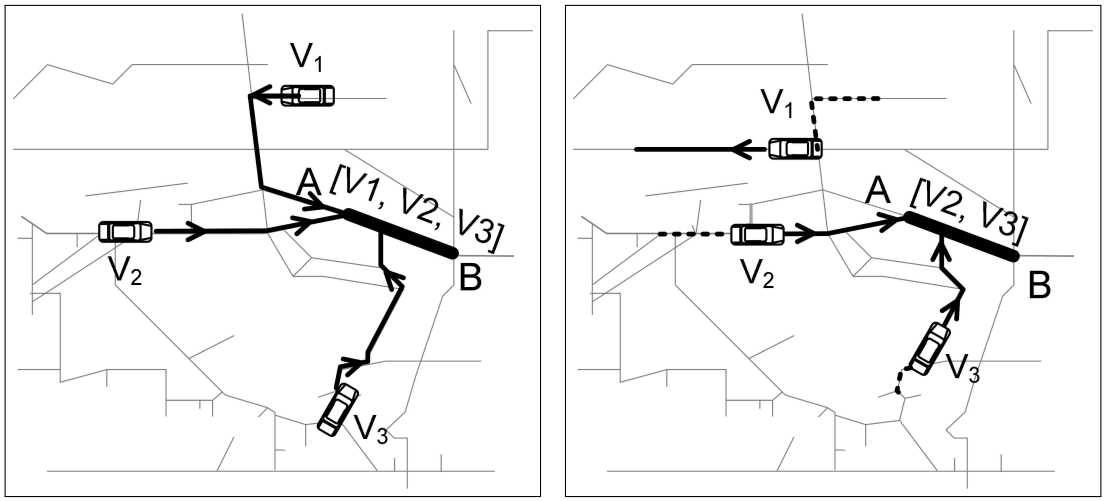
- The *Continuous Predictive Line Query (CPLQ)* that allows a user to specify a road that he/she would like to know the traffic condition of. The query then returns predicted traffic condition of the querying road at the estimated time that the user may pass by using a SPLQ. If there is any significant change of the prediction results on the querying road due to location updates of other vehicles, the updated query result will be automatically sent back to the user;
- A novel data structure, the *Time-Parameterized Query R*-tree (TPR^Q-tree)*, that indexes queries and efficiently handles the query result updates that evolve with time is designed to speed up the query processing and to reduce the query maintenance cost; and
- Three query algorithms that leverage the TPR^Q-tree and achieve increased efficiency for predictive traffic queries. We have carried out both theoretical and empirical study. Our experimental results demonstrate the effectiveness and efficiency of our approach.

Consider the following example. Bob plans to travel from Rolla to St. Louis which is about 100 miles (i.e., about 2-hour driving). Assume a traffic jam on his way to St.Louis when he sets off. If the navigation system computes the travel route for Bob based on current traffic condition, the route will probably include a detour to bypass the traffic jam. However, its possible that the traffic jam to be cleared an hour later while Bob is already on his detour route; Bob doesn't need to take the detour in the first place if the navigation system was able to calculate the route with predicted traffic condition.

Scenarios, such as the aforementioned, inspire designing a traffic prediction system that can provide better insight to travel planning. Moreover, the traffic prediction should be proactive/pervasive in that once the user initiates a traffic

condition prediction query, the system should continuously monitor the prediction results and report any changes that may be caused by the dynamic traffic.

Figure 6.1 illustrates an example of continuous traffic prediction. Specifically, Figures 6.1(a) and 6.1(b) show snapshots of three vehicles at time t_1 and t_2 respectively. The query road segment is $\overline{AB}$, and the current travel plans of the vehicles are highlighted by bold lines. As shown in Figure 6.1(a), three vehicles V_1, V_2, and V_3 may enter the querying road $\overline{AB}$. However, as time passes, vehicle V_1 changes its travel plan by making a right turn earlier at time t_2. As a result, only two vehicles (V_1 and V_3) may enter the query road, which requires an update of the previous query results.



(a) Vehicles at Time t_1 (b) Vehicles at Time t_2

Figure 6.1 Dynamic Nature of Continuous Traffic Prediction Information

None of the existing approaches can be directly adopted to build the above envisioned system. The closest related work that can provide traffic information include *range queries* and *density queries*. A range query reports traffic information in a given circular or rectangular area [44, 46, 47], but also contains traffic information on irrelevant roads rather than just the routes that the query issuer may pass by.

The density query [29,58,59,61] outputs even coarser information that include regions with more than a certain threshold value of vehicles. Moreover, most of the solutions to these query types assume an environment that objects move freely, which is not the case when the road network constraints are applied. Very few works [15, 36] can be found that consider road network constraints. Those

few, however, only support queries on current traffic condition but not on traffic prediction.

The rest of this chapter is organized as follows: Chapter 6.1 formally defines the problem; Chapter 6.2 introduces our proposed data structures, followed by Chapter 6.3 which elaborates the query algorithms; Then, Chapter 6.4 presents an analytical cost model; and Chapter 6.5 reports the experimental results.

6.1. DEFINITIONS

Definition 4. *[CPLQ] A continuous predictive line query* $CPL = (e_q, t_q, t_c, \rho)$ *continuously monitors the moving objects which will be on the query road segment* e_q *at the query time* t_q*, and returns query results whenever the number of query results differ more than a threshold* ρ*. Specifically, let* R_i *denote the query results at time* t_i $(t_c \leq t_i \leq t_q)$*, the CPLQ returns the query results in the form of* $\{(R_1, t_c), (R_2, t_2), \cdots, (R_k, t_q)\}$*, and* $|R_{i+1}| - |R_i| > \rho$*.*

The Continuous Predictive Line Query (CPLQ) is developed based on the PLQ introduced in Chapter 5. As an example, `CPLQ = (AB, 0800, 0730, 20)` means that the user issued a query at 7:30a.m. (i.e., t_c) and is interested in the traffic at road AB at 8:00 a.m. (i.e., t_q). The query issuer expects the server to report changes in prediction results if the difference of the number of vehicles on the querying road is more than 20.

Note that it is not necessary for the query issuer to specify the threshold parameter. Instead, the threshold can be automatically chosen by the server according to past experience (traffic flow records) to reflect significant traffic changes. Moreover, the server can also provide the query issuer the traffic information in an easy-to-understand form like "may have traffic jam" or "traffic flow will be good" based on the raw number of query results and the number of lanes on the specific road.

CPLQs have an inherent temporal aspect in which the query continuously monitors moving objects on the requested query road segment. A naive approach to answer a continuous query is to reprocess the same query at every timestamp till the query lifespan expires. Such an approach is computationally inefficient if there are no change of the query results at consecutive timestamps.

A better approach is to update the query results only when an object in the current result becomes invalid or a new object joins the query result due to its moving function changes. Given the possibility of large number of moving object

updates per timestamp, this requires an indexing structure that facilitates quick identification of which updates affect which CPLQ in order to achieve efficient query performance.

6.2. TPR^{Q}-TREE

A new data structure named *Time-Parameterized Query R*-tree (TPR^{Q}-tree)* is proposed to efficiently answer CPLQs issued by users and to efficiently index road networks and moving objects. In what follows, we describe the two data structures in detail.

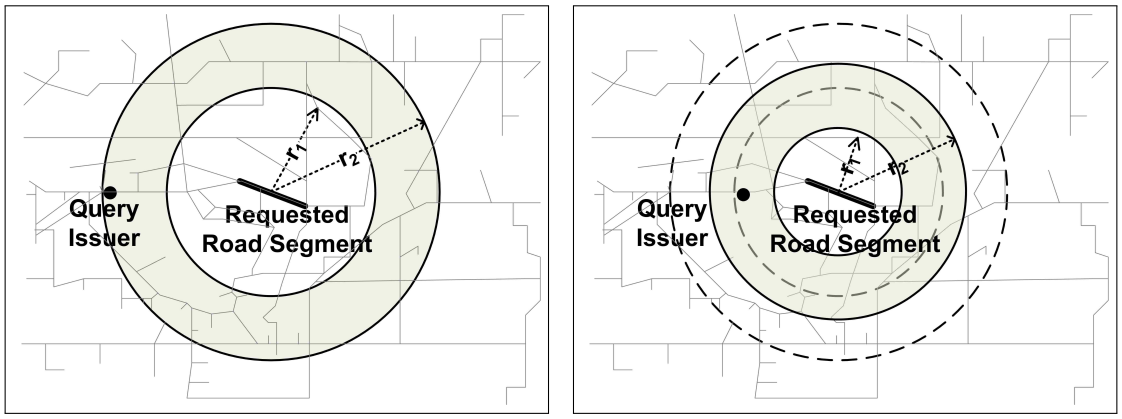
The TPR^{Q}-tree does not simply index the query road segment of a CPLQ, but instead, indexes an *Influence Region (IR)* for each CPLQ. The Influence Region is the region which covers majority of moving objects that may enter the desired query road segment at the future query time. In other words, if objects in the IR update their movement functions, the query results may be affected.

To better understand the concept of IR, consider Figure 6.2 that illustrates a query Q that aims to predict moving objects entering a predefined road segment; Figure 6.2(a) shows the IR of Q at query issuing time and Figure 6.2(b) the IR after 30 minutes have elapsed. As shown, the IR has a ring shape with its inner radius denoting the road distance traveled by the object at minimum speed > 0 and its outer radius denoting the road distance the object can travel at its fastest moving speed in 30 minutes (the query interval)[4]. All moving objects covered by this ring can potentially enter the query road segment.

As time evolves and gets closer to the future query time, the remaining time to reach the queried road segment shortens, resulting the IR to shrink as shown in Figure 6.2(b). More specifically, at the query issuing time, the CPLQ considers all objects within a 30 minutes radius to reach the road segments. After 10 minutes of the query issuing time, the CPLQ only considers objects that are within 20 minutes to enter the road segments. This example also exemplifies the temporal nature of IR.

Modeling the shrinking IR requires it to be stored as a parameterized ring which has moving speed attached to both inner and outer radius. The inner radius is associated with a minimum moving speed towards the query road segment while the outer radius is associated with a maximum speed towards the query

[4]Outliers such as objects stopped at gas stations, hence minimum speed equal to zero, are excluded from calculations.



(a) Influence region at query issuing time: t_c (b) Influence region at a later timestamp: t'_c

Figure 6.2 Shrinking Influence Region at Time t_c and $t'_c(> t_c)$

road segments as shown by the arrows in Figure 6.3. The time-parameterized IR is formally defined as follows.

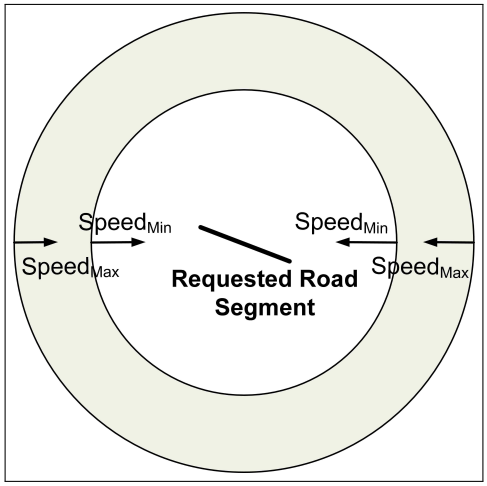
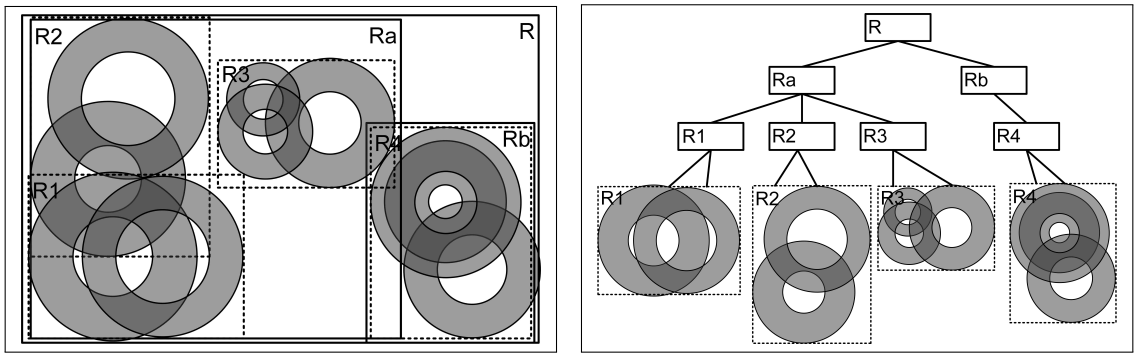


Figure 6.3 Shrinking Speeds of the Influence Region

Definition 5. *[Influence Region (IR)] Let $Q = (e_q, t_q, t_c, \rho)$ be a CPLQ. The Influence Region (IR) is a time-parameterized ring in the form of $IR = (c, r_1, speed_{min}, r_2, speed_{max})$, where c is the middle point of the querying road e_q, r_1 and r_2 are the radius of the inner and outer circles respectively, and $speed_{min}$ and $speed_{max}$ are the shrinking speed of the inner and outer circles respectively. The radii are computed as follows: r_1=$RoadDist(speed_{min} \cdot (t_q - t_c))$ and r_2=$RoadDist(speed_{max} \cdot (t_q - t_c))$.*

Figure 6.4 illustrates the structure of a TPR^Q-tree. The base structure of the TPR^Q-tree is the R^*-tree. There are three types of nodes in the TPR^Q-tree, the leaf

nodes, immediate parent node of the leaf nodes, and higher-level internal nodes. These are elaborated as follows:



(a) Leaf nodes in the TPR^{Q}-tree (b) An Overview of the Entire TPR^{Q}-tree

Figure 6.4 The Structure of the TPR^{Q}-tree

- Leaf nodes: An entry in the leaf node of the TPR^{Q}-tree stores information of a group of CPLQs. The information includes each query's parameters (e_q, t_q, t_c, ρ), the corresponding Influence Region (IR), a list of query issuers, and a pointer to the query results.

- Intermediate parent nodes: Each entry in the parent node of the leaf nodes stores a pointer to the leaf node and a time-parameterized MBR that bounds all the IRs of the queries in the leaf node. The time-parameterized MBR has a speed attached to each edge as shown in Figure 6.5. The speed of each edge is the minimum speed among the speeds of outer rings of all IRs in the MBR. The moving direction of each edge is pointing to the center of the MBR so that the MBR shrinks as time passes and bounds the shrinking IRs. The time-parameterized MBR is stored as a six-tuple $(x_1, y_1, x_2, y_2, speed, t_u)$ where (x_1, y_1) is the coordinates of the left lower corner of the MBR, (x_2, y_2) is the right upper corner of the MBR, $speed$ is the speed of each edge, and t_u is the latest time that the parameters of the MBR is updated.

- Higher-level internal nodes: An entry in higher level internal nodes contains a pointer to the child node and a time-parameterized MBR that bounds MBRs in the child node. Each edge of the MBR is associated with a minimum speed among the speeds of its child MBRs and each edge is moving towards the center as well.

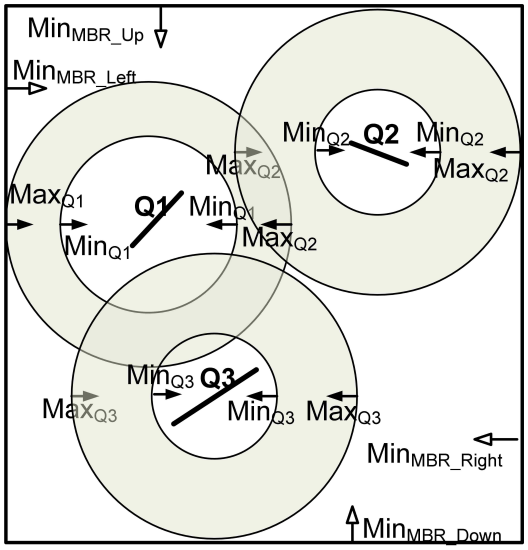


Figure 6.5 Shrinking Speeds of the MBR

There are three types of basic operations in the TPR^Q-tree: inserting a new query, deleting an existing query, and updating an existing query.

An outline of the insertion algorithm is shown in Figure 6.6. Given a new CPL query, its IR (denoted as IR_{new}) is calculated (line 1). The TPR^Q-tree is then searched to find the proper leaf node to store the new query. The algorithm to identifying the leaf node (chooseSubTree() on line 2) will be described shortly in the next paragraph. At the end of the search, if the same query is found in a leaf node (since other users may have already issued the same query), the results stored with the query will be directly returned to the user and the ID of this user will be appended to the list of query issuers.

Note that two queries are considered the same if they are querying traffic of the same road segment at the same near-future timestamp. Moreover, without affecting the service quality much, the query cost can be significantly saved by requiring users to specify the query time at a lower resolution of the time (e.g., every 10 minutes instead of every second) so that the probability of having the same queries at the same timestamp will be increased. If the new query does not exist in the tree, it will be inserted into the identified leaf node (line 5-10) and the predictive line query algorithm will be executed to obtain the initial query results for this new query (line 11).

It is worth noting that the insertion at the leaf node may trigger updates to its parent nodes all the way up to the root node in that the speed and the size of the MBRs of its ancestor nodes may need to be adjusted to ensure the newly inserted

query is enclosed. In addition, if the insertion encounters a node that is full (line 10), the node will be split. The node splitting algorithm is similar to that in the R*-tree. The only difference is that the speeds of the MBRs after the splitting need to be re-calculated.

Procedure TPR^Q-tree Insert
Input : Q
Input : Q_{Result}

1. $IR_{new} \leftarrow Q.getIR(Q.time())$
2. $node \leftarrow TPRQ.chooseSubTree(TPRQ.root, IR_{new}, Q.time())$
3. **if** $node = DATA$ **then**
4. $\quad Q_{Result} \leftarrow node.result()$
5. **else**
6. $\quad numOfChildren \leftarrow node.numChildren()$
7. $\quad$ **if** $numOfChildren < QueryRTree.MAX$ **then**
8. $\quad\quad node.addAChild(Q)$
9. $\quad$ **else**
10. $\quad\quad NodeSplit(node, Q, Q.time)$
11. $\quad Q_{Result} \leftarrow RD.snapshotQuery(Q)$
12. **return** Q_{Result}

Figure 6.6 Description of the TPR^Q-tree Insert Operation

The chooseSubTree() algorithm is explained in Figure 6.7. This process starts from the root. For each node being examined during the search, the chooseSubTree() algorithm first computes the MBR of each entry of this node at the current timestamp based on the shrinking speed of the corresponding MBR. The entries with the MBRs that fully cover IR_{new} will be considered first (i.e., $areaEnl = 0$). If none of the MBRs fully cover IR_{new}, the entry with the MBRs that needs the minimum enlargement to include IR_{new} will be considered. If there are several candidate entries, the entry with the MBRs with the smallest area will be chosen to break the tie. At the end of the search, the algorithm returns either an existing query (if the same query is found) or a leaf node for inserting the new query.

The deletion algorithm is introduced next. A CPL query needs to be deleted from the TPR^Q-tree either when the query issuer passes the querying road segment or when the issuer withdraws the query before the query expires. Figure 6.8 outlines the deletion process.

Procedure chooseSubTree
Input : $parent, IR_{new}, time$
Output : $atreenode$

```
1.  minAreaEnl ← BIGNUMBER
2.  minArea ← BIGNUMBER
3.  for each children ∈ parent do
4.      areaEnl ← findAreaEnlargement(children, IR_new, time)
5.      area ← findArea(children, IR_new, time)
6.      if (minAreaEnl > areaEnl) or (minAreaEnl = areaEnl and (minArea > area)) then
7.          newNode ← children
8.          minAreaEnl ← areaEnl
9.          minArea ← area
10. if newNode not LEAF then
11.     return chooseSubTree(newNode, IR_new, time)
12. else if newNode = LEAF then
13.     duplicate ← newNode.findDuplicate(IR_new, time)
14.     if duplicate = nil then return children
15.     return duplicate
```

Figure 6.7 Description of the ChooseSubTree Operation

Given a query to be deleted, the first step of the deletion process is to locate this query. The search starts from the root of the TPR^Q-tree. At each level, the entries with the MBRs that fully cover the query's IR will be considered (line 1), and their children nodes will be checked in the same way until the leaf nodes are reached. Then, check each located leaf node to identify the one that contains the query to be deleted. After deleting the query from the leaf node, the MBRs of the leaf node may need to be re-calculated, and the update may propagate to the ancestor nodes of this leaf node all the way to the root of the tree. In addition, if the deletion causes a node underflow (containing entries fewer than half of the capacity), the under-flow treatment will be applied (line 6). The under-flow treatment considers merging with a sibling node first. If the merging can not be done due to the relatively full occupation of the sibling nodes, entries of the under-flowed node will be deleted and reinserted into the tree.

A query update is processed as follows. First, we search the TPR^Q-tree to locate the leaf node containing the query. If the query with the new parameters is still covered by the MBRs of the leaf node, we will update the query parameters as well as the speed of the MBRs of this leaf node if the speed needs to be changed to the new query parameters. If the new query can no longer be included in the

current leaf node, we delete the query and treat it as a new query to be inserted into the tree.

Procedure TPR^Q-tree Delete
Input : Q

```
1.  parentNode ← TPRQ.search(Q)
2.  if parent ≠ null then
3.      parentNode.remove(Q)
4.      updateMBR()
5.      if parentNode.numChildren() < QueryRTree.MIN then
6.          underflowTreat(parentNode)
```

Figure 6.8 Description of the TPR^Q-tree Delete Operation

6.3. CONTINUOUS PREDICTIVE LINE QUERY ALGORITHMS

In this section, we present the CPLQ algorithm which consists of two phases: the initial phase and the maintenance phase. The initial phase computes the query result that is valid at the query issuing time. The maintenance phase maintains the query results as time passes.

6.3.1. Initial Phase. Upon receiving a new query from a user u, the TPR^Q-tree will be updated as discussed in Chapter 6.2. Recall that if the new query coincides with a previously stored query in the TPR^Q-tree, there is no need to execute this query again. Instead, the stored query results will be directly returned to the user u, and hence repeated query execution is avoided. This is one of the advantages of the TPR^Q-tree. In practice, it can be expected that many people might be interested in some particular road segments. That could be because the road segments often have traffic congestion issues, or they are the hubs for many popular destinations. Thus, in this kind of situation, using the TPR^Q-tree to group the same users with respect to the same query helps save on query cost.

If the new query cannot be found in the TPR^Q-tree, we will first insert the new query into the tree, and then execute a snapshot predictive line query [65] to identify those moving objects that may enter the query road segment at the query time based on their current movement functions. These initial query results will be reported to the user and stored along with the new query in the TPR^Q-tree. Due to

the characteristics of mobile objects, the initial query results will need to be revised during the subsequent maintenance phase until the query expires.

6.3.2. Maintenance Phase. The query results computed at the initial phase may need to be updated upon changes of some vehicles' travel plans as shown in Figure 6.1.

If a vehicle changes its moving direction or speed dramatically, the vehicle will send an update to the server. Upon receiving the update message, the server performs two tasks. The first task is to update the object in the R^D-tree [65]. The second task is to check if the update affects existing queries by answering the following two questions: (1) Is this object currently included in any existing query result? (2) Is this object going to be in some queries results' after the update? Given an object update and one query, there are four cases for the above two questions:

1. The object is included in the query result, and is still the query result after the update.
2. The object is included in the query result but will no longer be a valid query result after the update.
3. The object is not included in the query result but will become the query result after the update.
4. The object is not included in the query result and will also not be the query result after the update.

Among these four cases, only the second and third cases influence the query results. In the second case, we need to remove the updated object from the affected query results; while in the third case, we need to add the object to the affected query results. The challenge is how to efficiently categorize each update message into one of the four cases against all existing CPL queries. A brute-force approach that scans all the queries and checks if the object is in or is expected to appear in their query results is obviously time consuming since an object may just affect a small set of existing queries. Therefore, to reduce unnecessary comparisons, we leveraged the proposed TPR^Q-tree and proposed three query maintenance algorithms with increasing performance achievements: (1) solo-update maintenance; (2) solo-object maintenance; (3) batch-object maintenance.

The details of the three maintenance algorithms are presented in the following subsections.

6.3.2.1. Solo-update (SU) maintenance. The solo-update (SU) maintenance algorithm considers the update of an object information as two parts separately: the deletion of the old object information and the insertion of new object information. Correspondingly, the SU algorithm conducts two searches on the TPR^Q-tree for each object update. The first search looks for a set of CPL queries (denoted as Q_{old}) which the object belonged to at the object's previous update timestamp t_{old}; the second search looks for a set of CPL queries (denoted as Q_{new}) to which the object will belong to after its update at the current timestamp t_{new}. Note that in the case when a new object joins the system (an insertion only), the first search will be skipped and only the second search will be executed. In contrast, when an object exits the system (a deletion only), only the first search will be executed.

To obtain the query set Q_{old}, we start the search from the top of the TPR^Q-tree. For each entry of the visited internal tree node, we compute its MBR at t_{old}. Recall that MBRs and influence regions stored in the TPR^Q-tree are associated with shrinking speed. Therefore, to obtain the MBR at t_{old}, we need to expand it on all the four directions by $MBR_{speed} \cdot (t_u - t_{old})$, where t_u is the last time that the MBR is updated. Then, we check if the old object position falls into the expanded MBR. If so, that means this object may be included in the CPL queries stored under the children leaf nodes of this entry. Therefore, we will further check the children nodes of this entry in the similar way.

When the search reaches the leaf node, we do not simply scan all the query results associated with each query in this node because it could be time consuming. Instead, we take the advantage of the object travel destination and the influence region to prune queries that definitely do not contain the old object position. First, we prune the CPL queries whose query road segments are not on the traveling direction of the object according to its old travel destination. Then we compute the influence regions of the remaining queries at t_{old}. The center of the old influence region is the same as the one stored in the tree, while the inner radius (r_{old_inner}) and the outer radius (r_{old_outer}) are computed based on the inner/outer speed multiplied by the time difference as shown in Equations 10 and 11, respectively. If the old object position is within the old influence region of the CPL query, that means this object may be included in the corresponding CPL query. Then, we further check the actual query results of this query and remove the object if found.

Moreover, the speed of the influence regions of the affected CPL queries and the MBRs in their ancestor nodes may need to be recalculated if the deleted object contributes to the minimum or maximum shrinking speed.

$$r_{old\_inner} = RoadDist(speed_{min} \cdot (t_q - t_{old})) \tag{10}$$

$$r_{old\_outer} = RoadDist(speed_{max} \cdot (t_q - t_{old})) \tag{11}$$

An example of computing Q_{old} is illustrated in Figure 6.9 which shows an object's old position (the black circular point), its old destination (denoted as a star) and the influence regions of five CPL queries (a, b, c, d, e) at t_{old}. Queries a, b and c are pruned using the object's old travel destination since they are not in the traveling direction of the object. Then, the influence regions at t_{old} of queries d and e are computed. Since the object is located in both queries' influence regions, the result lists of the two queries will both be checked.

The process for identifying the query set Q_{new} is very similar to that for Q_{old}. The main differences are the computations of the MBRs and the influence regions used during the search. Since t_{new} is after t_u (the latest update time of the MBR), the MBR at t_{new} is computed by shrinking the stored MBR at four directions by $MBR_{speed} \cdot (t_{new} - t_u)$. The influence regions of CPL queries at t_{new} is computed based on the following equations.

$$r_{new\_inner} = RoadDist(speed_{min} \cdot (t_q - t_{new})) \tag{12}$$

$$r_{new\_outer} = RoadDist(speed_{max} \cdot (t_q - t_{new})) \tag{13}$$

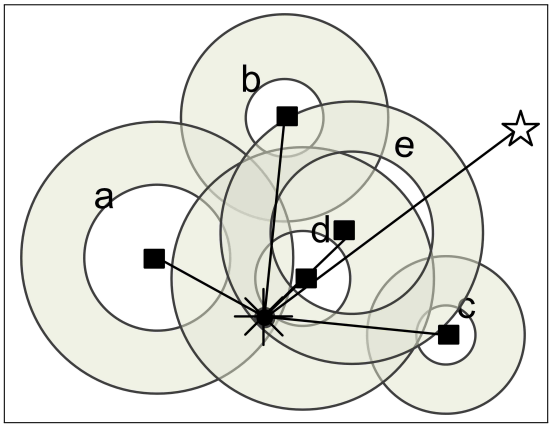


Figure 6.9 Influence Regions at t_{old} in a Leaf Node of the TPR^Q-tree

For each CPL query in the obtained Q_{new}, we add the object's new position to its query result. Also, the speeds of the influence regions of the queries in the Q_{new} and the MBRs of their ancestor nodes may need to be updated based on this object's new speed.

Finally, we record the number of changes for each query result during the object update. If the number exceeds the specified threshold ρ, the server will return the latest query results to the query issuer. An outline of the SU maintenance algorithm is given in Figure 6.10, and Figure 6.11 outlines the algorithm to check the overlap of the point and the node-MBR.

Procedure SU Maintenance
Input : $t_{old}, x_{old}, y_{old}, des_{old}, t_{new}, x_{new}, y_{new}, des_{new}, vId$
Output : Q_{new} and Q_{old}

1. **if** t_{old} is not NULL **then**
2. $\quad Q_{old} \leftarrow isContainPoint(t_{old}, x_{old}, y_{old}, TPR^{Q}.root)$
3. $\quad DeleteOldResult(Q_{old}, vId)$
4. **if** t_{new} is not NULL **then**
5. $\quad Q_{new} \leftarrow isContainPoint(t_{new}, x_{new}, y_{new}, TPR^{Q}.root)$
6. $\quad InsertNewResrult(Q_{new}, vId)$
7. Report updated query results to the user

Figure 6.10 Description of the Solo-Update Maintenance Algorithm

6.3.2.2. Solo-object (SO) maintenance. In general, an object's new and old positions in the same update message are relatively close to one another since they share two consecutive positions on the object's path and are bounded by the maximum moving speed multiplied by the maximum update interval. Therefore, the new and old positions in an object's single update message are very likely to be covered by the influence regions of the same or nearby CPL queries. In other words, these two positions may affect the CPL queries stored in the same or sibling nodes in the TPR^{Q}-tree. Based on this observation, we propose the solo-object (SO) maintenance algorithm that considers the object update message as a whole and computes the two sets of CPL queries affected by the update (i.e., Q_{new} and Q_{old}) simultaneously in one round of the search in the TPR^{Q}-tree.

The SO algorithm is expected to be more efficient than the previous discussed SU algorithm because the SU algorithm carries out two rounds of the search

```
Procedure isContainPoint()
Input : t, x, y, destination, root
Output : Q
1.  node ← root
2.  Q ← empty
3.  nodeList ← {node}
4.  while node not leafnode do
5.      for each entry ent ∈ node do
6.          MBR ← compute ent.MBR at time t
7.          if (x, y) is in MBR then
8.              nodeList ← nodeList ⋃ {ent.child} − {node}
9.      node ← nodeList[0]
10. while nodeList not empty do
11.     for each entry ent ∈ node do
12.         if ent.CPL is on the object's destination then
13.             IR ← compute the influence region of ent.CPL at time t
14.             if (x, y) isin IR then
15.                 if (x,y) is included in the ent.CPL then
16.                     Q ← Q ⋃ ent
17. return Q
```

Figure 6.11 Description of the IsContainPoints() Algorithm

separately (for the new and the old positions) to identify the two sets of CPL queries, which may visit the visit the same tree nodes repeatedly.

Figure 6.12 presents an outline of the SO maintenance strategy. In particular, we start the search from the root of the TPR^{Q}-tree. For each entry of the visited internal node, we compute its MBRs at t_{old} and t_{new}, respectively, similar to the computation discussed in the SU algorithm. If the object's old or new position is covered by the MBRs, the child node of this entry will be added for checking as well. Until the leaf level is reached, the influence regions of the CPL queries stored in the visited entries will be computed at t_{old} and t_{new}, respectively. Then, the old and new positions will be compared against the respective influence regions. If the old position is included in the influence region of a CPL query, the old position will be removed from the query result. If the new position contributes to a CPL query, the new position will be inserted into the query result. Next, the shrinking speeds of influence regions of all the updated CPL queries will be recalculated. The MBRs of the ancestors of the updated entries will be recomputed as well. At the end, if the query results have been changed significantly (exceeding a certain threshold), a query update report will be sent back to the query issuers.

Procedure SO Maintenance
Input : $t_{old}, x_{old}, y_{old}, t_{new}, x_{new}, y_{new}, vId$

1. $Q_{old} \leftarrow empty$
2. $Q_{new} \leftarrow empty$
3. $node \leftarrow root$
4. $nodeList \leftarrow \{node\}$
5. **while** *node* **not** $leafnode$ **do**
6. **for each** entry *ent* of the node **do**
7. $MBR_{old} \leftarrow computee.MBRattimet_{old}$
8. $MBR_{new} \leftarrow computee.MBRattimet_{new}$
9. **if** $(old_x, old_y) isin MBR_{old}$ **then**
10. $nodeList \leftarrow nodeList \bigcup \{ent.child\}$
11. **else if** (new_x, new_y) is in MBR_{new} **then**
12. $nodeList \leftarrow nodeList \bigcup \{ent.child\}$
13. remove node from *nodeList*
14. $node \leftarrow nodeList[0]$ \\get the first node in the nodeList
15. $NodeUpdateList \leftarrow empty$
16. **while** *nodeList* is not empty **do** \\now check the leaf nodes
17. **for** each entry *ent* of the node **do**
18. **if** $ent.CPL$ is on the object's old destination **then**
19. $IR_{old} \leftarrow$ compute the influence region of $ent.CPL$ at time t_{old}
20. **if** (old_x, old_y) is in IR_{old} **then**
21. **if**(old_x, old_y) is included in the $ent.CPL$ **then**
22. remove (old_x, old_y) from $ent.CPL$
23. $NodeUpdateList \leftarrow NodeUpdateList \cup ent$
24. **else if** $ent.CPL$ is on the object's new destination **then**
25. $IR_{new} \leftarrow$ compute the influence region of $ent.CPL$ at time t_{new}
26. **if**(new_x, new_y) is in IR_{new} **then**
27. **if**(new_x, new_y) are the new answer to $ent.CPL$ **then**
28. add (old_x, old_y) to $ent.CPL$
29. $NodeUpdateList \leftarrow NodeUpdateList \cup ent$
30. Recalculate the IR of entries in $NodeUpdateList$
31. Update the MBRs of the ancestor nodes of entries in $NodeUpdateList$
32. Report updated query results to the user

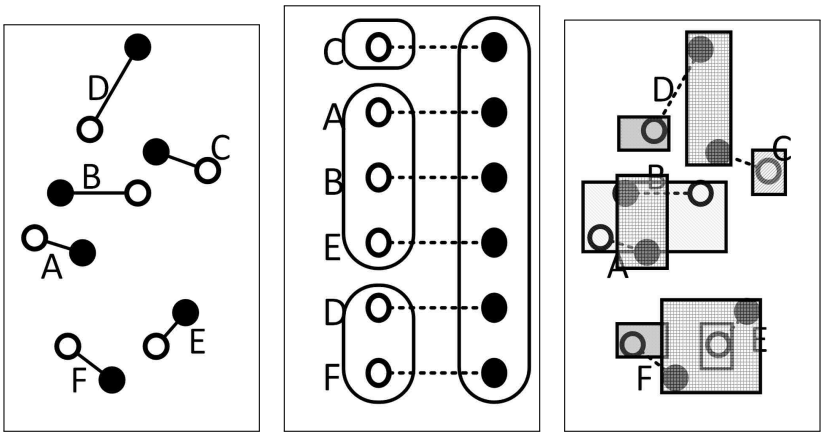
Figure 6.12 Description of the Solo-Object Maintenance Algorithm

6.3.2.3. Batch-object maintenance. With the increase of the number of moving objects, the number of object updates at each timestamp will also grow larger. Among the large amount of updates that are received at the same timestamp, it is likely that some are from nearby objects and, hence, they may influence the same or nearby CPL queries. According to this observation, we take one step further from the previous SO algorithm by considering all updates received at one single timestamp as a whole, and propose a batch-object (BO) maintenance algorithm.

Upon receiving the update messages at a timestamp, the BO algorithm first conducts two rounds of grouping: (i) grouping objects based on their update timestamps; (ii) grouping objects based on their location proximity. In the first round of grouping, the objects' new positions can be easily grouped together as they are all at the same current timestamp. The challenging design issue is the grouping of the objects' old information. This is because objects which issue updates at the same time now may have issued their last updates at totally different timestamps. In other words, the different timestamps associated with the old positions make these old positions incomparable.

It isn't possible to directly group the old positions based on only location proximity while overlooking their update timestamps. To overcome this problem, we group old positions based on their updated timestamps by putting the old positions with the same updated timestamp into the same group. So far, we have obtained one group for the objects' current positions and multiple groups for the objects' old positions. The benefit of the first round of grouping is that it avoided repeated computation of the MBRs and influenced regions in the TPR^{Q}-tree for objects falling into the same timestamp. Next, we divided the obtained groups into sub-groups based on the location proximity. Specifically, we employed a similar technique in the R^{*}-tree by constructing MBRs for the nearby objects. For the sake of clarity in subsequent discussion, we call the MBRs constructed from update messages, the *message-MBRs*.

Figure 6.13 illustrates the group formation for a set of object-update messages. In Figure 6.13(a), the circles denote the old positions and the black points denote the new positions of six objects: A, B, C, D, E, and F. All these update messages were received at time t_{new}. The previous updates of objects A, B, C, D, E and F were made at time t_1, t_1, t_0, t_3, t_1, and t_3 respectively. Figure 6.13(b) shows the update messages grouped according to their timestamps. As shown in the figure, new positions form a single group as they all have the same timestamp.



(a) Object's New and Old Positions (b) Object Grouping based on Timestamps (c) Message-MBRs

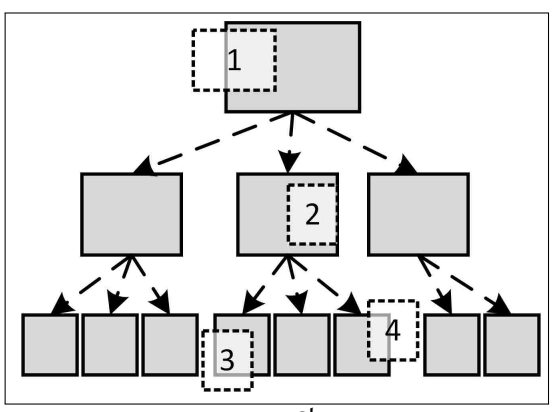
Figure 6.13 Group Formation for a Set of Update Messages

Old information, however, forms three different groups with object C in the first group, objects A, B and E in the second group, and objects D and F in the third group. Then, Figure 6.13(c) shows the message-MBRs of further partitioning of the three groups based on their location proximity.

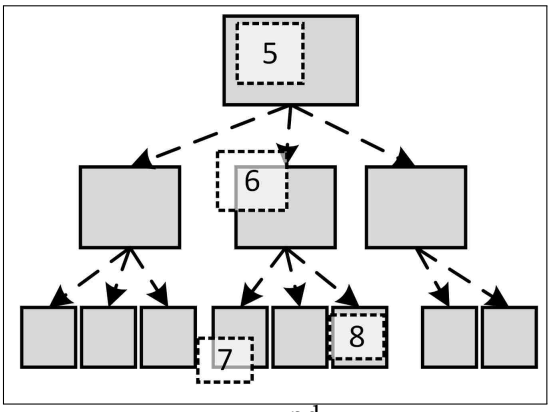
After the grouping, the next step of the SO algorithm is to search the TPR^Q-tree to find the CPL queries that overlap with the message-MBRs, i.e., to find the CPL queries that may be affected by this set of object updates. Here, if we search each message-MBR in the TPR^Q-tree, repeated node accesses will still be exist. For example, suppose that the received update messages form two message-MBRs. Figure 6.14(a) and 6.14(b) show the search of the first and second message-MBRs, respectively, where the dashed rectangles denote the message-MBRs and the number is the count of the page accesses).

As shown, the two searches accessed the same tree nodes consecutively and resulted in a total of eight total node accesses. If the two searches are carried out simultaneously as shown in Figure 6.14(c), the repeated node accesses can be avoided and the cost will be cut in half. Therefore, in our BO algorithm, we consider all message-MBRs against the MBR in the same tree node to ensure that each tree node is not to be accessed more than once for a set of updates received at the same timestamp.

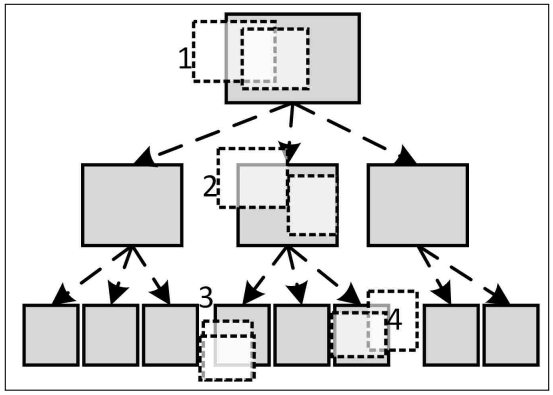
Also noticeable is that not all message-MBRs overlap with the MBR of the examined tree node. Figure 6.15 illustrates this kind of situation, whereby the two message-MBRs M_1 and M_2 overlap with different nodes in the TPR^Q-tree.



(a) Tree Search for 1st message-MBR



(b) Tree Search for 2nd Message-MBR



(c) Tree Search for Two Simultaneous Message-MBRs (Node Accesses = 4)

Figure 6.14 Different Strategies for Searching the Message-MBRs

If a message-MBR does not overlap with the MBR of a node in the TPR^Q-tree, there is no need to further consider this message-MBR under the branches of this tree node. Our BO algorithm leverages this pruning criteria which greatly reduces the amount of comparison as well as the computation of the MBRs and influence regions needed for the comparison.

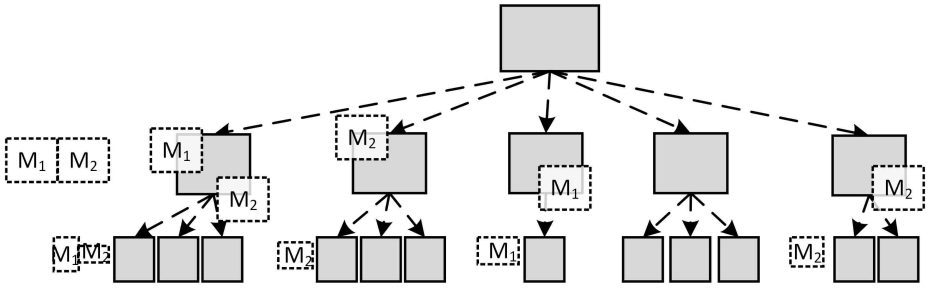


Figure 6.15 Message-MBRs Overlapping with MBRs in the TPR^Q-tree

An overview of the BO algorithm is shown in Figure 6.16. First, the message-MBR are obtained (line 2–3). Then, the search starts from the root of the TPR^Q-tree (line 4). For each visited internal node, we check the flags of all the message-MBR to see if this node's parent overlaps with the message-MBRs. If so, we further compare this message-MBR with the MBRs of each entry in the examined node. Flags are updated for the children node of each entry after the comparison (line 13). As for the leaf (line 16–27) node, we also check the flags first. For the candidate CPLQs obtained from the search, we finally evaluate the query against the actual object position in the update message to adjust the query results similarly to that in the previous two maintenance algorithms.

6.4. QUERY COST ANALYSIS

The CPL query algorithms consist of two phases: the initial phase and the maintenance phase. At the initial phase, a snapshot predictive line query is executed. The cost of this snapshot query has been analyzed in [65]. Moreover, all the proposed three algorithms share the same initial phase, but they differ in the maintenance phase. Therefore, we focus on the analysis of the maintenance cost in this section.

In our cost analysis, we assume that both moving objects and querying road segments are uniformly distributed in the space being considered. Without loss of generality, we also assume that all moving objects are alive during the life time of the queries and all the queries considered are issued at the same timestamp with the same life-time length. We estimate the average maintenance cost in terms of the number node accesses (or disk page accesses assuming one node per disk page). Specifically, the average maintenance cost per query per timestamp is computed as the total number of disk page accesses ($Cost_{total}$) divided by the product of the total number of CPL queries (N_q) and the total timestamps (T) during the query life time, which can be expressed as:

$$Cost = \frac{Cost_{total}}{N_q \cdot T} \tag{14}$$

6.4.1. Cost of Solo-Update (SU) Maintenance. To obtain the average maintenance cost according to Equation 14, we only need to estimate the unknown value, i.e., $Cost_{total}$. The total number of page accesses ($Cost_{total}$) during the query

Procedure BO Maintenance Algorithm
Inputs : $updates$: a set of update messages received at the same time stamp
Outputs : Q_{Result}

```
1.  G← groups of object updates at the same timestamp
2.  for each group G do
3.      message-MBR ← group objects in G according to location proximity
4.  node ← root of the TPR^Q-tree
5.  nodelist ← {node}
6.  while node is not the leafnode do
7.      for each message-MBR do
8.          if flag(node, message-MBR) is true then
            \\this node's parent overlaps with the message MBR
9.              for each entry in node do
10.                 compute the MBR at the message-MBR's timestamp
11.                 if MBR overlaps with the message-MBR then
12.                     add this entry's child node to the NodeList
13.                 set the flag(entry.child, message-MBR) to true
14.     remove node from nodeList
15.     node ← nodeList[0] \\get the first node in the nodeList
16. while nodeList is not empty do
17.     for each entry in the node do
18.         for each message-MBR do
19.             if flag(node, message-MBR) is true then
20.                 compute the IR at the message-MBR's timestamp
21.                 for each position contributes in message-MBR do
22.                     if the position is in IR and position is included in the ent.CPL then
23.                         if message-MBR's timestamp is the new timestamp then
24.                             add position to ent.CPL
25.                         else
26.                             remove position from ent.CPL
27.                         NodeUpdateList ← NodeUpdateList ∪ ent
28. Recalculate the IR of entries in NodeUpdateList
29. Update the MBRs of the ancestor nodes of entries in NodeUpdateList
30. Report updated query results to the user
```

Figure 6.16 Description of the Batch-Object Maintenance Algorithm

life time using the SU algorithm is the multiplication of two factors: the number of times that the TPR^Q-tree is accessed and the number of page accesses per tree access.

The number of times that the TPR^Q-tree is accessed is twice that of the total number of update messages in the system. This is because the SU maintenance approach treats one update message as a deletion followed by an insertion. Let

m_i denote the total number of update messages from an object i during the query life time T. The total number of update messages in the system is computed as $\Sigma_{i=1}^{N}(m_i)$, where N is the total number of objects. Then, the total number of tree accesses (denoted as $Total_{ta}$) by the SU algorithm is $2 \times \Sigma_{i=1}^{N}(m_i)$.

The second step is to estimate the cost of searching the TPR^{Q}-tree for a single operation (either a deletion or an insertion). Given an object's old or new position, the average number of CPL queries whose influence regions may contain this position is determined by the area of the influence region at the update timestamp and the density of the queries in the space being considered. The area covered by the outer circle of an influence region at the query update timestamp t_u is estimated as $\Pi \cdot [(T - t_u) \cdot SpeedMax]^2$, where $(T - t_u) \cdot SpeedMax$ is the outer radius r_{out} of the influence region at t_u. Since t_u evolves from time 0 to T, the average area of the influence region (denoted as $Area_{IR}$) during T is the integration $\frac{\int_0^T \Pi \cdot r_{out} \mathrm{d}t}{T}$ which is equal to the following:

$$Area_{IR} = \frac{\Pi \cdot (SpeedMax)^2 \cdot T^2}{3}$$

Take the object position as the center and draw a circle of the size of $Area_{IR}$ as illustrated in Figure 6.17, where the dark point in the center represents the object O's position and the circles drawn in solid lines represent the outer circle of the CPL queries' influence regions. If the CPL query road segment intersects with the object's circle, this query's influence region may contain the object. In other words, the CPL query whose querying road segments are in the shaded area should be considered.

Next, we estimate the number of queries that may fall into the object's influence circle. Assuming the road segments being queried are uniformly distributed, the number of CPL queries per unit area is $Density_{query} = N_q / Area_{total}$. Thus, the number of queries in the object's influence circle is the multiplication of the area of the influence circle and the density, which is $n_q = Area_{IR} \cdot Density_{query}$. Since these n_q queries are close to one another, they are likely to be stored close to one another in the TPR^{Q}-tree as well. Therefore, the average number of leaf nodes containing the n_q queries is estimated as n_q / f, where f is the fanout (i.e., average number of entries per node) of the TPR^{Q}-tree. The number of the parent nodes of the leaf nodes containing these n_q queries is estimated as n_q / f^2. In general, the number of nodes accessed at the level l of the tree is e (n_q / f^l), where the level of the leaf node is 1. After summing up the node accesses at each level, we obtained the total

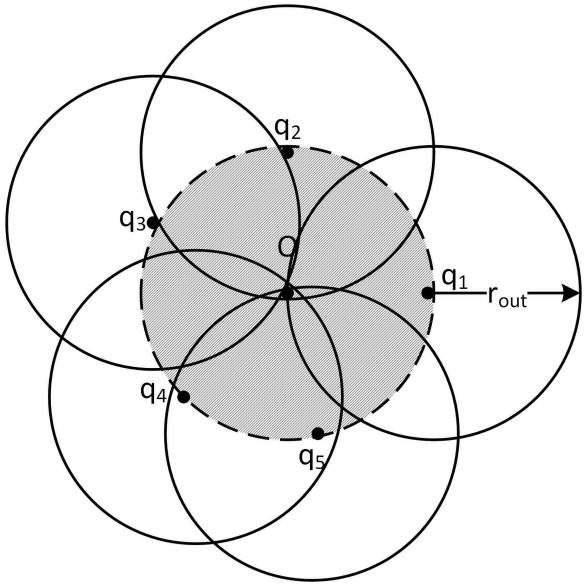


Figure 6.17 Example of the CPL Queries that may Contain the Object

number of node accesses during one round of the tree search: $C_{tree} = \sum_{l=1}^{h} (n_q/(f^l))$, where h is the height of the tree.

Finally, we can compute the average query cost of the SU maintenance as follows:

$$Cost_{su} = \frac{Total_{ta} \cdot C_{tree}}{N_q} \tag{15}$$

$$= \frac{2 \times \Sigma_{i=1}^{N}(m_i) \cdot \Pi \cdot (SpeedMax) \cdot T)^2 \cdot (1 - (1/f^h))}{3 \cdot (f-1) \cdot Area_{Total}} \tag{16}$$

6.4.2. Cost of Solo-Object (SO) Maintenance. The SO cost analysis follows the same procedure as that of the SU. The SO algorithm also utilizes Equation 14. To find the total cost for the entire query life time, the number of tree accesses during the query life time and the average page accesses for a tree access must be estimated.

The SO algorithm accesses the TPR^Q-tree each time it receives an update message (Figure 6.12). Furthermore, it compares the entire update message against the TPR^Q-tree. Thus, the number of tree accesses in SO is simply the total number of update messages: $\Sigma_{i=1}^{N}(m_i)$.

The search cost for one tree access (i.e., for an update message) for the SO algorithm is also estimated using the same calculation method used for the SU

algorithm. Thus, we first find the area where the object can influence query results as shown in Figure 6.18. The small circles at the center of the bigger circles are the object's new and old positions. The bigger circles represent the maximum query overlap area of each position. The distance between two positions is d. The total query-affected area is the area covered by the boundaries of the two circles: $(2\Pi \cdot r_{out}^2) - (r_{out}^2 \cdot \arccos(\frac{d}{2r_{out}}))$ and the average query-affected area over the T time period is

$$Area_{IR,SO} = \frac{\int_0^T (2\Pi \cdot r_{out}^2) - (r_{out}^2 \cdot \arccos(\frac{d}{2 \cdot r_{out}}))dt}{T},$$

which is equivalent to:

$$[2 \cdot \Pi \cdot s^2 \cdot T^2] - [\frac{s^2 \cdot T^2 \cdot \arccos(\frac{d}{2 \cdot s \cdot T})}{2}] + [\frac{d}{6s^2T} \cdot (\frac{(u_T^{1.5} - u_0^{1.5})}{3} + \frac{d_2 \cdot (u_T^{0.5} - u_0^{0.5})}{4})]$$

where s is the $SpeedMax$, and $u_i = (SpeedMax \cdot t)^2 - (\frac{d^2}{4}); i \in \{0, T\}$.

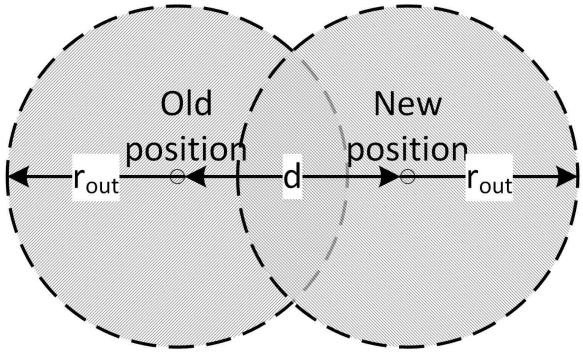


Figure 6.18 Maximum Query Overlapping Area Corresponding to One Update Message

The number of queries in the influence region $n_{q,SO}$ is $Area_{IR,SO} \cdot Density_{query}$. Following the same procedure as in SU cost analysis, the total number of node accesses during one round of tree search $C_{tree,SO}$ is obtained as: $C_{tree,SO} = \sum_{l=1}^{h} (n_q/(f^l))$ (h is the height of the tree). Then the average query cost of SU maintenance becomes:

$$Cost_{so} = \frac{\Sigma_{i=1}^{N}(m_i) \cdot Area_{IR,SO} \cdot (1 - (1/f^h))}{(f-1) \cdot Area_{Total}} \quad (17)$$

Theorem 1. *The maintenance cost of the SO algorithm is always no greater than the cost of the SU algorithm.*

Proof. The worst case of the SO algorithm is obtained when each point accesses entirely different tree nodes. This means that no overlap between the circles showed in Figure 6.18 exists. When there is no overlap between circles, d becomes zero and the area covered by two circles (i.e, $Area_{IR,SO}$) becomes $(2\Pi \cdot r_{out}^2)$. When the value of $Area_{IR,SO}$ is plugged on Equation 17, it is simplified to Equation 16, which is the cost for SU algorithm. □

6.4.3. Cost of Batch-Object (BO) Maintenance. The BO algorithm also needs to find the number of tree accesses and the number of page accesses per each tree accessed to estimate the $Cost_{total}$ in Equation 14.

The number of tree accesses in BO algorithm depends on the number of distinct timestamps at which update messages are initiated, because, the BO algorithm groups update messages received at the same timestamp and access the tree only once per all messages in the same timestamp. Thus, assuming the number of distinct update message timestamps are N_{ts}, the TPR^{Q}-tree accesses is also N_{ts}.

The average page accesses per each tree access depends on the number of subgroups and their MBR extent. Figure 6.19(a) shows a subgroup whose MBR dimensions are d_1 x d_2 with a maximum query overlap area. The MBR of the subgroup is represented by the filled rectangle and the maximum distance to a query road segment from the MBR boundary is the r_{out}. The area covered by the dashed-line shape is the influence region of the MBR. Its area is calculated as follows:

$$a_i = (d_1 \cdot d_2) + 2(d_{i1} \cdot r_{out}) + 2(d_{i2} \cdot r_{out}) + \Pi r_{out}^2. \tag{18}$$

Since all MBRs are compared simultaneously against each tree node, repetitive node accesses are not counted. The page accesses per one tree-search is the total distinct node accesses on the TPR^{Q}-tree. This means that the common areas in different query influence regions should be counted only once. Figure 6.19(b) shows an example of overlapped query-influence areas. This area is given in Equation 19.

$$\sum_{i=0}^{n-1} a_i - \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} Overlap_{i,j} \tag{19}$$

The answer to this calculation is approximated to the area of the MBR covered by all round-cornered rectangles $Area_{IR,BO}$ (Figure 6.19(b)). Then the average number of queries that can overlap with the area is:

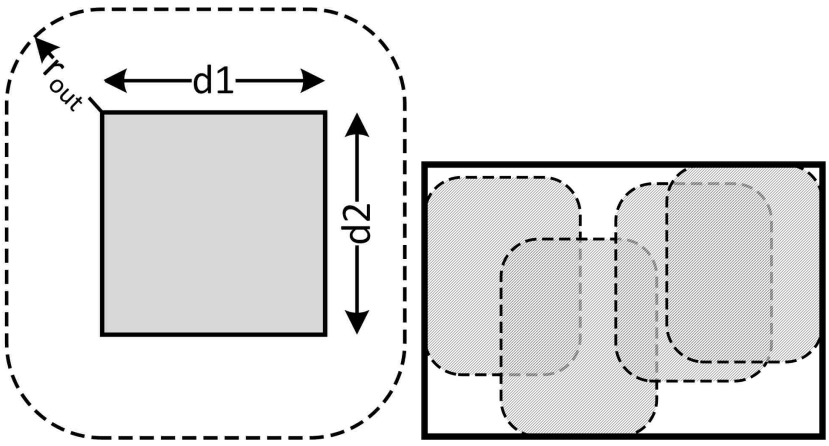
$$n_{q,BO} = Area_{IR,BO} \cdot Density_{query} \tag{20}$$

Following the same cost estimation steps as in the SU and SO cost analysis, the average BO maintenance cost becomes:

$$Cost_{BO} = \frac{N_{ts} \cdot Area_{updateMBR} \cdot (1 - (1/f^h))}{(f-1) \cdot Area_{Total}} \tag{21}$$

Theorem 2. *The maintenance cost of the BO algorithm is always less than the cost of the SO algorithm.*

Proof. The worst case of MO algorithm is obtained, when each MBR accesses distinct tree nodes. To have distinct node accesses, no overlap should exist among MBRs. This can be explained with Equation 19. According to 19, when $Overlap_{i,j}$



(a) for a sub-group of same time stamped positions (b) for a group of sub-groups

Figure 6.19 Maximum Query-overlap Area

$\forall i, j$ is zero, the maximum effective area is obtained and it is simply the summation of all MBRs areas.

Then, let us consider the number of elements in a group is $elements_g$. Hence, N_{ts} in Equation 21 can be re-written as $Total_{messages}/elements_g$. The number of subgroups also can be obtained in terms of $elements_g$. For that, assume the average number of elements in a subgroup to be $elements_{sg}$. Then, the average number of subgroups $n = 1 + (elements_g/elements_{sg})$. The maximum aggregated area of MBRs is obtained when n is large. The largest n is obtained when $elements_g$ is largest and $elements_{sg}$ is the smallest. The smallest possible $elements_{sg}$ is one. When, $elements_{sg}$ is one, each a_i in Equation 19 becomes Πr_{out}^2, according to Equation 18. With the deduced parameter values Equation 21 can be simplified as follows:

$$Cost_{BO} = \frac{\frac{Total_{messages}}{elements_g} \cdot \sum_{i=0}^{elements_g+1} \Pi r_{out}^2 \cdot (1 - (1/f^h))}{(f-1) \cdot Area_{Total}}$$

In this equation, when $elements_g$ = 1, cost for SO algorithm is obtained. To sum up, the BO algorithm obtains the SO maintenance cost, when: (i) no overlaps between subgroups exists, (ii) the number of objects in one group is one, and (iii) the number of elements in sub groups is one (i.e., two subgroups in the group). □

6.5. PERFORMANCE STUDY

The proposed algorithms were evaluated on moving object data sets generated by the Brinkhoff generator [64]. The moving object datasets were generated using four real road maps selected from different states in United States. The road maps contain a similar number of road segments, but different topologies. The number of moving objects in each dataset ranges from 10 k to 100 k.

For each dataset, sets of queries were randomly generated by randomly selecting a query issuer and its query issuing position. Then the querying road segment was selected from its path which will be reached by the end of the predictive query length. The predictive query lengths were ranged from 10 to 60 minutes. The chosen parameters and their values are presented in Table 6.1. The bold values represents the default value for each parameter.

We compare our proposed approaches with a naive approach that executes snapshot predictive line queries [65] for every update message. Since the initial phase of the four approaches are the same in the following, we only report the

Table 6.1 Simulation Parameters and Their Values for Continuous PLQ Algorithm

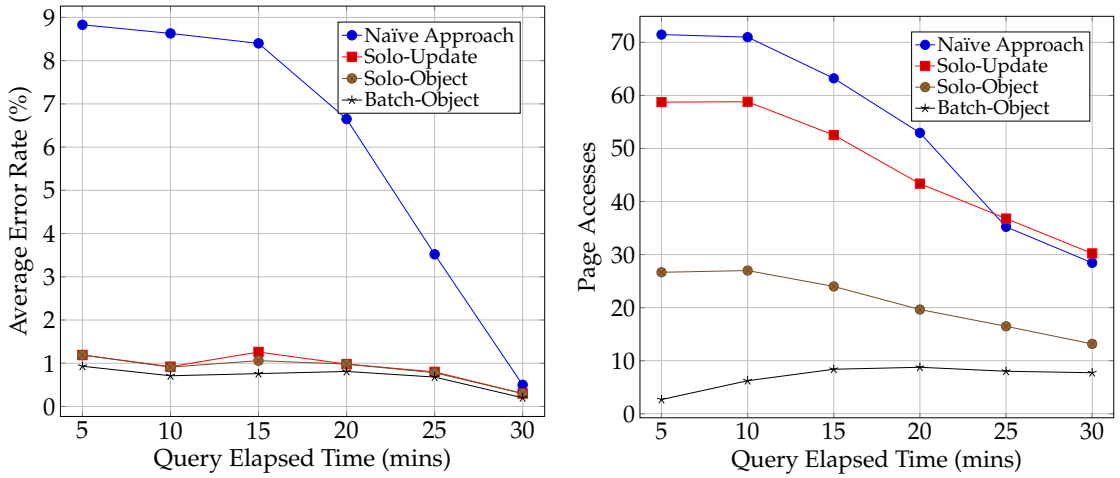
Parameters	Values
Buffer	YES, **NO**
Query Percentage	0.5%, 2%, 5%, **20**%, 40%, 60%, 80%, 100%
Number of moving objects	10K, 20K, ..., **50K**, 60K, ..., 100K
Predictive time length	10, 20, **30**, 40, 50, 60 (mins)
Road maps	Alpine (CA), Charles (MD), Salem (NJ), **Worth (MO)**

comparison of their maintenance cost. Their performance is measured in terms of the prediction error rate and the I/O cost. The error rate was computed by comparing the number of objects in the predictive query results with the actual number of objects on the query road segment at the query time. The I/O cost is the number of disk page accesses. The reported I/O cost is the average page accesses per query per timestamp. It first calculates the average page accesses (averaged per query and per timestamp) during each 5 minute time interval throughout the query life time ($AvgPg(5min)$). The average page accesses for the entire query life time is, then, calculated by taking the average of all $AvgPg(5min)$'s in the query life time.

6.5.1. Maintenance Phase. In the following, we evaluated various factors that may affect the query performance including time, number of queries, number of moving objects, predictive length, road topology, and buffer size.

6.5.1.1. Query performance over the query lifetime. First, we evaluated the performance of the query result maintenance as time passes. We compute the average maintenance cost and prediction error rate per timestamp within each 5-minute interval for 30 minutes. Figure 6.20(a) and 6.20(b) report the performance of the naive approach and the three proposed approaches: Solo-Update (SU), Solo-Object (SO), and Batch-Object (BO).

From Figure 6.20(a), we can observe that our proposed three algorithms consistently yield a much lower prediction error than the naive approach. This is because the naive approach defines the query ring based on the Euclidean distance to the query road segment [65], whereas the influence regions employed by SU, SO and BO consider the road distance, which is a more accurate method of estimating vehicles that may enter the query road segment. In addition, we can also see that the prediction accuracy of our three algorithms is similar, which is not affected by the various maintenance algorithms adopted.



(a) Error Rate (b) Query Cost

Figure 6.20 Query Performance Over the Query Life Time

Figure 6.20(b) shows the average maintenance cost. As expected, our proposed three algorithms all perform better than the naive approach, and the BO approach performs best. This is because the naive approach needs to execute each query at every timestamp, which may involve duplicate efforts when there is no change in results.

The TPR^{Q}-tree, which is utilized by the proposed algorithms, takes object update messages and checks all affected queries simultaneously, which helps reduce unnecessary efforts on the query processing significantly. The reason that the BO approach achieves the least maintenance cost is that the BO approach is the most aggressive compared to the other two approaches and considers the most possibilities for simultaneous executions for cost saving.

In addition, the experimental results also demonstrate the evolvement of the maintenance cost with time. As shown in Figure 6.20(b), the closer to the end of the query life time (i.e., the time when the query issuer will enter the querying road segment), the less maintenance cost is needed in general. The possible reason for such behavior is that the influence regions are shrinking as time passes and hence the number of objects to be checked become fewer. Note that the BO approach shows a slight increase of the maintenance cost at the beginning. The reason is that the BO approach considers all the updates issued at one timestamp and the

number of updates are fewer when the system is just starting because the objects take some time to speed up.

6.5.1.2. Effect of the number of queries. In this round of experiments, we evaluated the effect of the number of queries on the query performance by varying the total number of queries from 0.5% of the total number of moving objects to 100%. As shown in Figure 6.21, the naive approach exhibits a relatively stable performance regardless of the number of queries. This means that the average cost per query is independent from the total number of queries being executing. Each query is applied in the same process and on the same tree (R^D-tree). Hence, the cost depends only on the size of the R^D-tree, but not the number of queries.

Our proposed SU, SO, and BO approaches, however, access the TPR^Q-tree; furthermore, the number of queries stored in the tree changes the tree structure. In fact, the number of queries decides the tree fanout (f) and the height of the tree (h). These two factors directly impact the query maintenance cost. The query cost, in all three proposed algorithms, is proportional to the expression $\frac{1-\frac{1}{f^h}}{f-1}$. The impact of h and f is contravened in both the this expression and the average query cost.

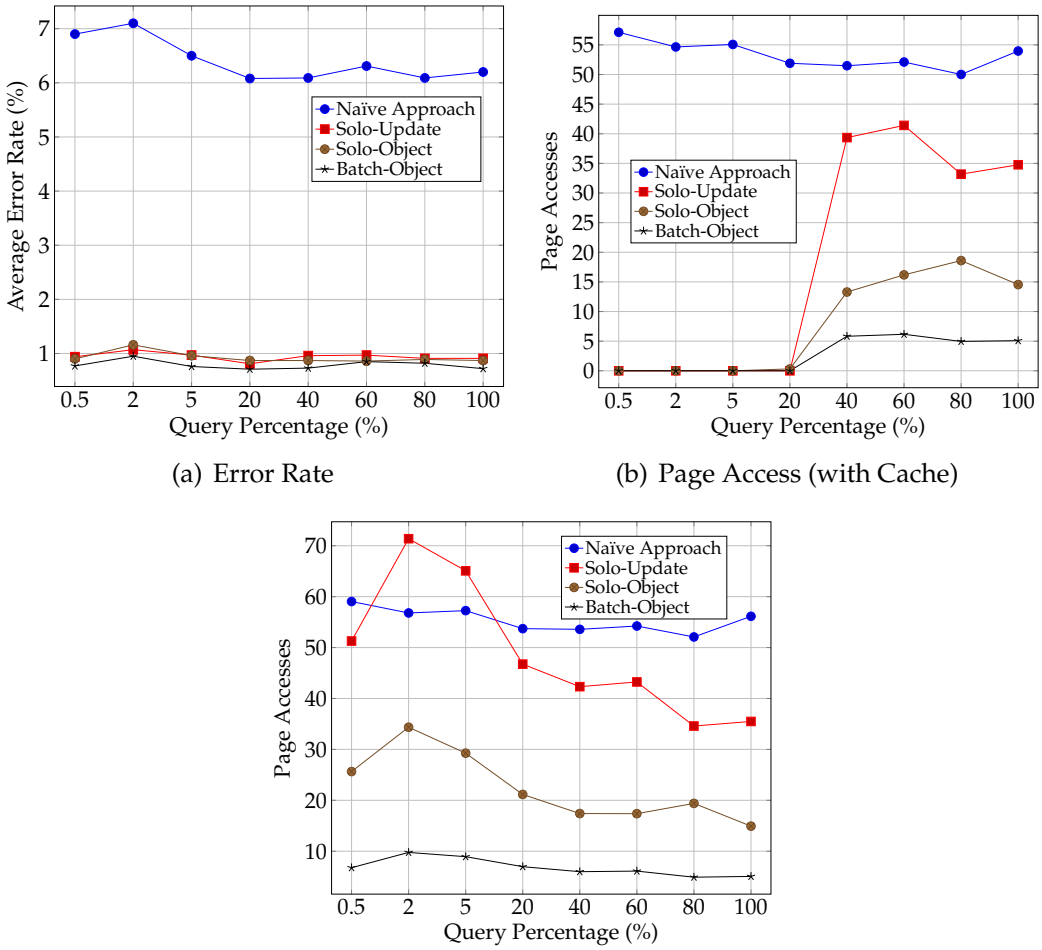
For smaller h values, the impact of both f and h is significant. For example, for 0.5% (250 in count) of queries, all queries can be accommodated in the root; which means h is one and f is greater (Table 6.2). When the number of queries is increased up to 2%, the number of tree levels increases and, at the same time, fanout decreases. Both these changes result to increase the value of the expression. When h gets bigger, the expression becomes nearly independent of h as $\frac{1}{f^L}$ becomes insignificant. The expression is, then, left only to f. Hence, as the number of levels is increased (i.e, higher number of queries), a smooth query cost decrement is demonstrated.

Table 6.2 TPR^Q-tree Structure's Information

Query Percentage	0.5%	2%	5%	20%	40%	60%	80%	100%
Number of Queries	250	1000	2500	10000	20000	30000	40000	50000
Number of tree levels	1	2	2	2	2	2	2	3
fanout	231	182	159	180	180	174	180	179

6.5.1.3. Effect of buffer utilization. We repeat the set of experiments conducted in the previous section to see the effect of the buffer utilization.

Specifically, we employ a buffer with 50 k capacity and LRU (Least Recently Used) replacement policy. Figure 6.21(b) reports the query cost for deferent query percentages with the buffer.[5] As the figure shows, the query maintenance cost up to 20% is essentially a zero and the rest of the query sets shows an increased query cost. The increased costs are comparable to that in Figure 6.21(c).



(a) Error Rate

(b) Page Access (with Cache)

(c) Page Access (Without Cache)

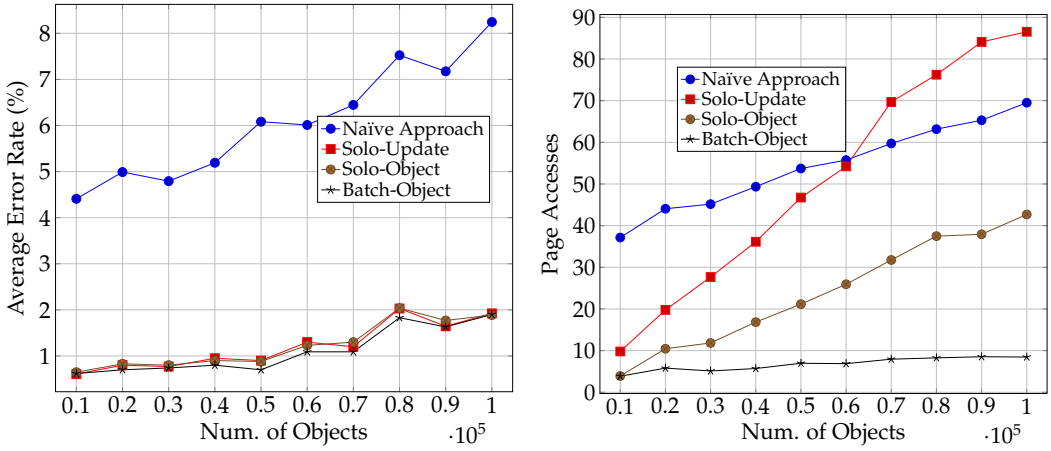
Figure 6.21 Effect of Number of Queries

Comparing Figure 6.21(c) with Figure 6.21(b), it is clear that a query performance improvement of up to 20% has occurred due to buffer usage. This is because, up to 20%, the number of tree nodes in the entire tree structure is less than 50. This means that the entire tree can be accommodated by the buffer. Thus, at most, one disk access is made per one tree node. Once the node is stored in the

[5]Since the accuracy is not affected by the buffer, it is omitted in the discussion

buffer, no buffer replacement is required. When the number of nodes in the tree exceeds the buffer size, the buffer cannot accommodate all necessary tree nodes simultaneously. Thus, buffer-miss rate increases and hence page access count increases.

6.5.1.4. Effect of number of moving objects. In this round of experiments, we evaluated performance based on the number of moving objects increasing from 10 K to 100 K. Figure 6.22(a) shows the average error rate of proposed algorithms together with the naive approach. As the figure shows, similar to the other cases reported in early sections, all three algorithms show competitive accuracy. The error rates, in all approaches, increase slightly with the number of objects. This is because the more moving objects, the more uncertainty in prediction. However, our approach always achieves a lower error rate for the same reason discussed in the previous section.



(a) Error Rate (b) Page Access

Figure 6.22 Effect of Number of Objects

Figure 6.22(b) shows the query cost of all four algorithms. According to the graph, one common feature of all the algorithms in this study is that they all consumed more page accesses when the object count was increased. In the naive approach, this happens because the R^D-tree expands with the higher number of objects and, hence, the number of node accesses is increased. In the proposed algorithms, the tree structure remains unchanged, but the number of update messages compared against the tree is increased.

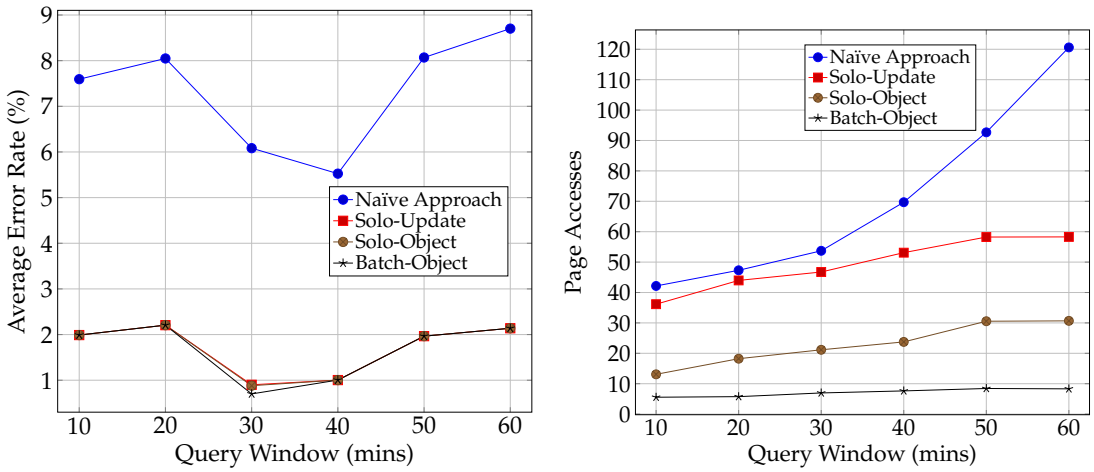
Another vital observation is the naive approach gives the worst query cost for a lesser number of objects, and it defeats the performance of SU when the number of objects are increased (approximately at 60k). The reason can be explained as follows. The page access count in the naive approach depends on two factors: the size of the R^D-tree and the number of update messages received. The expansion of the R^D-tree is slower for higher object counts than the smaller object counts. This same expansion speed will be applied on the page access count as well. Additionally, the number of update messages is directly proportional to the access count, because for each update message, the R^D-tree is searched.

However, the SU algorithm also accesses the TPR^Q-tree per each update message. In fact, SU algorithm accesses the TPR^Q-tree twice per each message. So, the SU algorithms' page access count increases at a faster rate compared to the naive approach. Similarly, the naive approach and the SO algorithm performance curves are more likely parallel to each other (i.e., the same rate). This is because, both naive and SO algorithms access their trees once per each message. The gap between two plots explains the advantage of the TPR^Q-tree over the R^D-tree.

The BO algorithm, on the other hand, behaves totally different to the other approaches and shows extremely better performance. As the figure shows, the BO algorithm has not been affected by the number of objects as it was in the other three algorithms, especially when the number of objects was higher. As a matter of fact, the BO algorithm's performance depends only on the number of different time stamps and it is countably finite, within the 30-minute time period. Thus, the BO shows a bounded query cost independent of the number of objects.

6.5.1.5. Effect of predictive time length. In this set of experiments, the predictive time length is varied from 10 to 60 minutes. As shown in Figure 6.23(a), the error rate stayed in a similar range regardless of the predictive time length for both approaches. The behavior can be explained as follows. For the naive approach, it executes the query at every timestamp and, hence, any change of object travel plan will be captured. Similarly, in proposed approaches, the effect of the object update on the query results at every timestamp is considered.

On the other hand, the predictive time length does affect the query cost as shown in Figure 6.23(b)). The query cost of the naive approach increases when the predictive time length is longer. This is because in the naive approach, a bigger ring query is generated for a longer predictive time length. In the proposed approaches



(a) Error Rate (b) Page Access

Figure 6.23 Effect of Predictive Time Length

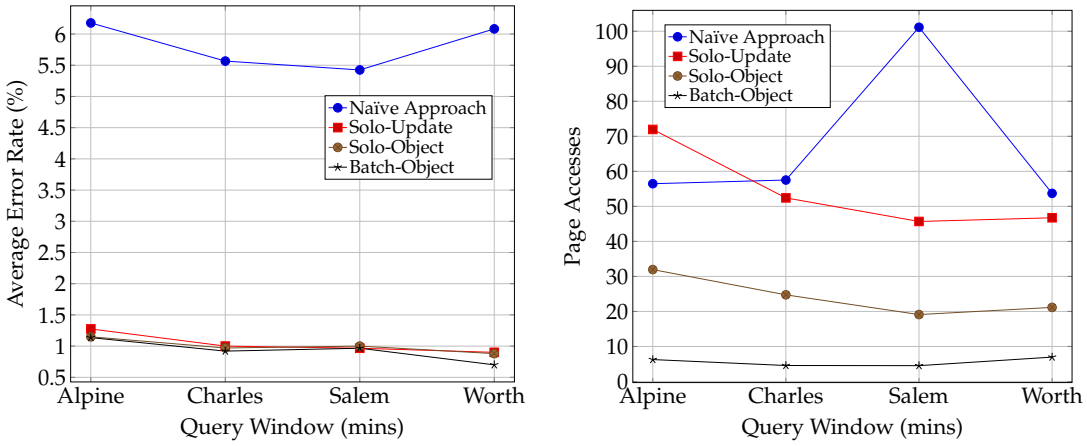
the query cost also increases with the length of the query window; but, at a slower rate. This is again due to the advantage of the TPR^{Q}-tree utilization.

As explained in Section 6.4, the proposed algorithms' total query cost depended on either the number of update messages (for SU and SO) or the number of different time stamps within the query life time (for BO). The average query cost for a 5-min time interval depended on the message counts within the 5 minutes. Thus, no matter how long the predictive query window was, the average query cost depended on the average number of messages within the query window.

Given a fixed number of objects and (assuming the same mobile patterns for any query window size) the average number of messages independent on the query window. The other factor that affected the query cost of the proposed algorithms was the query influence area: The higher the query window, the higher the query effective area. Thus, all three proposed approaches experienced slightly higher query costs with the wider query window.

6.5.1.6. Effect of road topology. This section evaluates the effect of the road topology by testing different maps: Alpine (CA), Charles (MD), Salem (NJ), and Worth (MO). The number of edges in each map was 1576, 1766, 1789, and 1573 , respectively, and the average road segment length was 232 m, 370 m, 515 m, and 551 m, respectively. By observing the average error rate of individual topology in Figure 6.24(a), the overall conclusion tended to confirm that the larger the number of edges, the lower the error rate. Regarding the page accesses as

shown in Figure 6.24(b), our approach was relatively independent of the number of edges. However, all three algorithms show better performance when the average road segment length was bigger. This is because, when the road segments are lengthier, the update messages time interval is more spaced out (further apart). Thus, algorithms handle less update messages.

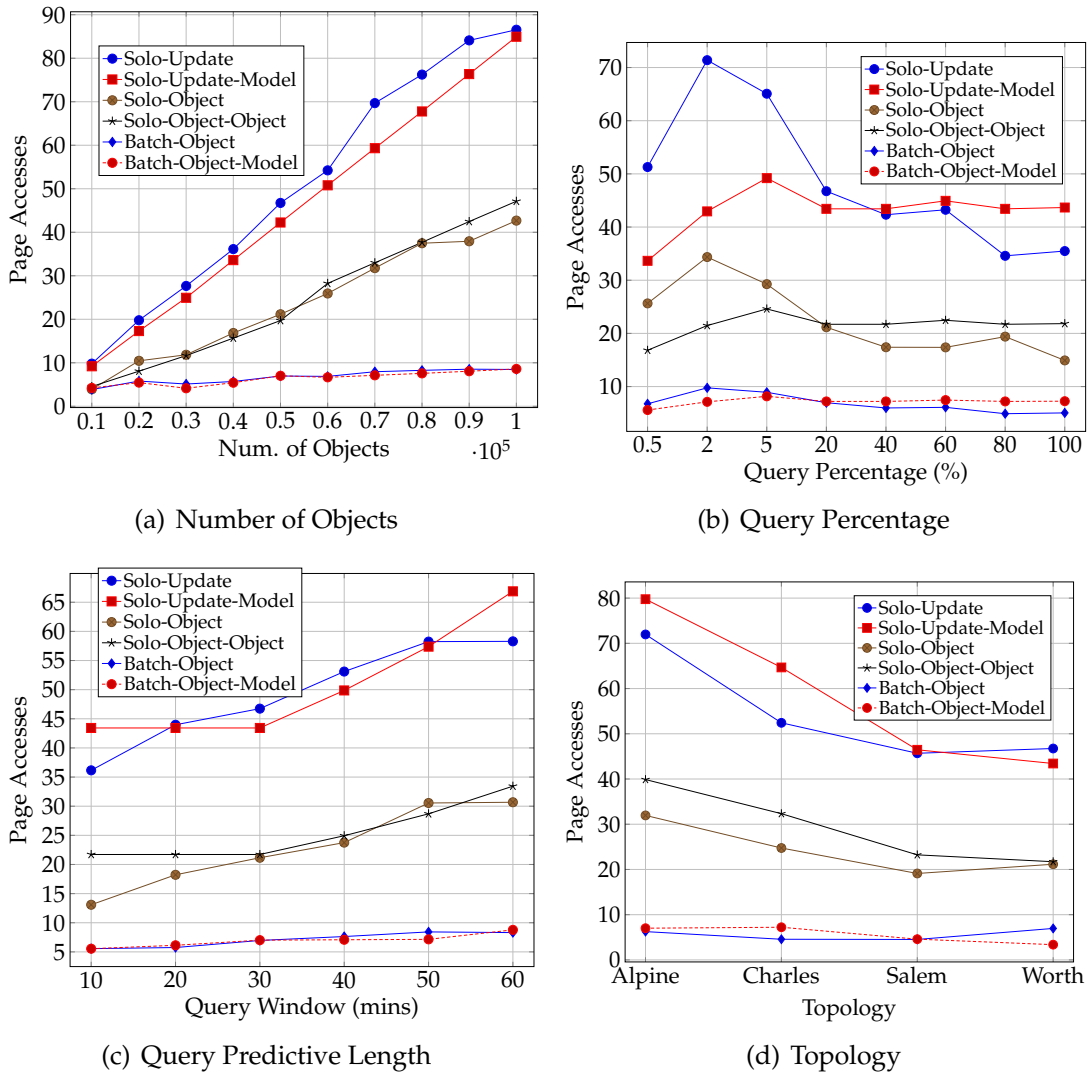


(a) Error Rate (b) Page Access

Figure 6.24 Effect of Road Topology

6.5.2. Cost Model Evaluation. This section validates the cost model discussed in Section 6.4 for maintenance cost of the proposed three algorithms. The evaluation was performed based on Equations 16, 17, and 21. Figure 6.25 compares the estimated cost computed from the cost model with the experimental results obtained from the proposed three maintenance algorithms. Figure 6.25(a) shows the effect of the number of objects. In this case, the cost model's error rate is below 10%. Figure 6.25(b) shows the effect of the number queries, whereby the estimation is getting close to the actual cost with the increase of the number of queries. The reason is straightforward. The cost model is developed based on uniform distribution of queries and when there are more queries, their distribution will be closer to uniform distribution.

Figure 6.25(c) shows the comparison of the estimated cost and the actual cost in the case when the predictive query length is varied. Again, we can see that the cost model yields an error of around 10%. Finally, Figure 6.25(d) reports the comparison results when testing different map topologies which also shows



(a) Number of Objects

(b) Query Percentage

(c) Query Predictive Length

(d) Topology

Figure 6.25 Cost Model Validation

comparatively good accuracy of the cost model. To sum up, our cost model achieved around 90% accuracy in most cases.

6.6. SUMMARY

This chapter addressed the effect of temporal domain on SPLQs result. Specifically, this chapter presented an efficient and effective method to track changes over time in SPLQ result which will be reported to the issuer if any significant change of the prediction is identified. The proposed method utilizes a novel indexing structure, namely TPR^Q-tree, to index query information. Three algorithms were also proposed to efficiently manage the update messages with the aid of the information in the TPR^Q-tree. The TPR^Q-tree maintains each query's

Influence Region, which covers the set of objects that will enter into the query segment at the query time. Three algorithms composed of two phases: initial phase and maintenance phase. The initial phase finds the SPLQ result and the maintenance phase maintains it. Three algorithms (Solo-Update, Solo-Object, and Batch-Object) employ different techniques to handle update messages and see its impact on the initial result. Solo-Update maintenance algorithm handles the deletion and the insertion of new object information separately while the Solo-Object maintenance algorithm considers both deletion and insertion together. The Batch-Object maintenance algorithm, on the other hand, handles update messages group-wise.

All three algorithms perform better than the naive approach, which repeatedly executes the initial phase. Among three proposed algorithms, the Batch-Object maintenance algorithm shows the best performance.

7. PREDICTIVE DENSITY QUERIES

This chapter explores solutions to the traffic congestion problem by providing information to users on possible future traffic congestions. Predicting traffic information based on the current and future traffic contributors' behavior, using *Continuous Predictive Line Query (CPLQ)*, presents travelers with influential information to evade possible traffic en-route. However, CPLQs are demand-driven queries, i.e., the user specifies a road of interest to him/her and the query returns predicted traffic condition of the demanded road segment at the estimated time that the user specifies, and may not produce the most relevant query information due to their inherent discrete nature. Thus, the key contributions of this chapter are:

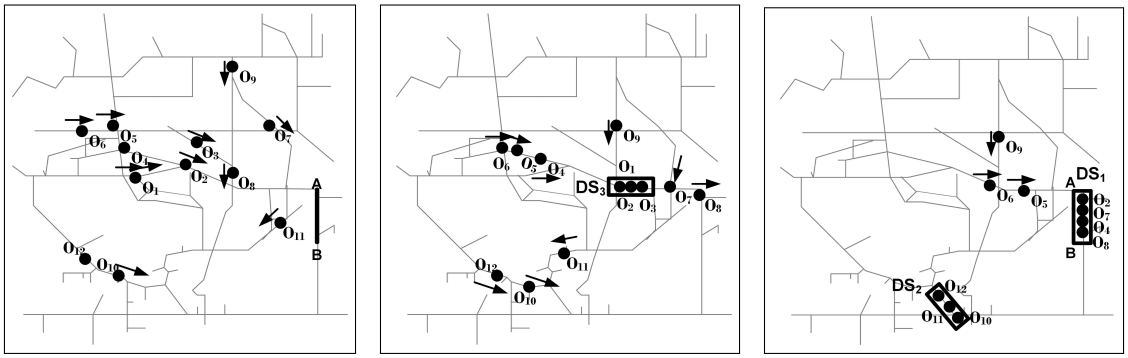
- to redefine the density query and introduce *Predictive Density Query* (PDQ) to provide more relevant, realistic, and reliable query results that takes *road network* into account to provide *predictive* traffic information;
- to design and implement an efficient query processing algorithm for predictive traffic information; and
- to perform extensive experimental study and analysis to evaluate the proposed algorithm.

Drawbacks of on-demand type queries are best exemplified using an example as follows. Consider Figure 7.1 that shows a PLQ request for road segment $\overline{AB}$. Here, the filled circles represent mobile objects, the rectangles surrounding the circles represent high density road segments, and arrows beside mobile objects represent its moving direction. A snapshot of vehicles at t_0 is illustrated in Figure 7.1(a), and the predicted vehicle information at time t_1 and at time t_2 are shown in Figures 7.1(b) and 7.1(c) respectively.

As shown, the predicted mobility information at t_2 acts in favor of two potential dense areas: DS_1 (on $\overline{AB}$); and DS_2. This, however, is not a realistic traffic congestion since a traffic congestion at some other road segment a priori, DS_3 at t_1 for example, could influence the vehicles to deviate from their previously reported (e.g., the destination) or inferred (e.g., the shortest path) information. As a result, the traffic congestion predicted for $\overline{AB}$ on t_2 would not take place, resulting

in either a congestion on a different road segment or no congestion on the network at time t_2 due to the detours.

A more practical scenario is explained in Figure 7.2. Due to the dense road segment DS_3, some vehicles (which are moving on the road segments covered by the shaded area) would not be able to continue on their previously planned path as expected. As a result, road DS_1 would not be dense. DS_2, however, might remain unchanged. This scenario explains that predicting dense areas on a given timestamp could be inaccurate unless the influence of former possible dense areas are taken into consideration. Consequently, evaluating traffic on arbitrary timestamps (e.g., user defined query timestamps), makes the prediction ineffective.



(a) A snapshot of vehicles at t_0 (b) Predicted Traffic Information at t_1 (c) Predicted Traffic Information at t_2

Figure 7.1 Predicted Traffic Information on Different timestamps

The closest query type that provides proactive-based predicted traffic information is the *density query*; the density query presents information on regions with more vehicles than a certain threshold. However, density query solutions simply consider the object's mobility on the Euclidean space thus not based on real world settings.

This chapter augments both demand-based queries and density queries by proposing a novel proactive query type named the Predictive Density Query (PDQ). In short, the term *mutually independent* refers to the dense road segments where one road segment's density does not influence another road segment's density (the formal definition can be found in Section 7.1). PDQ improves upon density queries by considering road networks under constraint rather than assuming an Euclidean space.

(a) A snapshot of vehicles at t_0 (b) Proposed Approach with Quarantine Area at t_1 (c) Proposed Approach with Quarantine Area at Time $t_2(> t_1)$

Figure 7.2 Influence of Prior Dense Areas on Later Dense Areas

The PDQ identifies the *earliest* dense road segments that are mutually independent. The term *earliest* means that the query process excludes two types of follow-up dense areas: (i) possible upcoming dense road segments that were already identified as dense before; and (ii) newly formed dense road segment occurrences due to the influence of an previous dense area. The first case is considered because the dense road segments are already at full road capacity hence, cannot be assumed to carry any through-traffic until the situation is resolved. The second case is considered due to the fact that the vehicles' reactions for a future condition has not been yet reported and hence unknown.

The rest of the chapter is organized as follows. Section 7.1 formally defines the density query problem. Section 7.2 and 7.3 present the utilized index structure and query algorithms, respectively. Section 7.4 reports the experimental results. The chapter is summarized in Section 7.5.

7.1. DEFINITIONS

Without loss of generality, the definition of density query assumes uni-directional or bi-directional roads with separate lanes for each direction. In other words, it is assumed that the high traffic density of one direction does not affect the traffic on the other direction. Under this assumption, in what follows presents definitions for *Density*, *Dense Road Segment*, *Mutually Independent Dense Road Segments*, and *Density Query*.

Definition 6. *[Density] The density of a road segment r is represented as $density(r) = N/m * len(r)$, where N is the number of objects on r, $len(r)$ is the length of road segment r, and m is the number of lanes.*

Definition 7. *[Dense Road Segment (DRS)] Given a density threshold ρ and a road segment s, the road segment s is dense, if and only if the density is greater than the threshold ρ.*

Definition 8. *[Mutually Independent Dense Road Segments (MIDRS)] Given any two dense road segments R_a and R_b with occurrence times t_a and t_b, respectively, R_a and R_b are mutually independent dense road segments, if*

1. *$O_a \cap Q_b = 0$, where O_i is the engaged object set of density on $R_i, i \in (a, b)$ and*
2. *Network distance between the closer end-points of R_a and R_b are greater than a threshold σ.*

Here, the first condition is used to ensure that only concurrent dense road segments are considered; dense road segments caused by the vehicles that have already been accounted for in antecedent dense road segment are excluded from considered mutually independent. For example, only the earliest dense road segment of a chain of dense segments is considered. The second condition ensures to avoid considering any subsequent dense road segment developed due to a previous density propagation. For example, vehicles moving on a road segment, say R_a, may change their route and avoid traveling through R_a due a predicted congestion and consequently contribute to a new congetion on R_b. In such a case, the second condition discards new road segment R_b.

Definition 9. *[Predictive Density Query (PDQ)] Given a road map G and a time window t_{max}, a Predictive Density Query (PDQ) gives a list of predicted mutually independent dense road segments $\{DS_1, DS_2, DS_3, ..., DS_n\}$, where the occurrence times $t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_n (t_n \leq t_{max})$; t_i is the occurrence time of DS_i.*

7.2. DATA STRUCTURE

Predictive Density Query (PDQ) utilizes an indexing structure that maintains both the network and mobile object information, and a two dimensional histogram which maintains the vehicle count of the cell for discrete timestamps. The PDQ can be supported by any indexing structure that supports predictive queries on objects moving under road network constrains, such as IMORS (Indexing Moving

Objects on road sectors) [15], the ANR-tree (Adaptive Network R-tree) [17], the R-TPR$^{\pm}$-tree [36], and the TPR^{uv} [38]. The proposed algorithm utilizes a recently proposed, efficient mobile object indexing structure, the R^D-tree [65], to index the road networks and moving objects.

The R^D-tree[6] indexes the road segments in its R*-tree where the object information is indexed with respect to the road segment. The two-dimensional histogram comprises of squared shaped cells that covers the road network. The cells maintain the counts of moving objects that might cross the cell within the time period [t_{now}; t_{now} + H] for equally calibrated timestamps; here H is the horizon – the time window in which the prediction is valid. The histogram is initialized according to the moving object's estimated traveling path.

7.3. QUERY ALGORITHM

The query algorithm consists of three phases – filtering, refinement, and refreshing – that are explained below.

7.3.1. The Filtering Phase. The filtering phase utilizes the histogram to extract out the most potential grid cells that may contain dense road segments (see Figure 7.3). The extracted cells are enqueued into a priority queue. The priority of the queue is decided according to, first, the timestamp and, next, the adjusted density. The adjusted density for time t_i ($density_i$) is calculated as:

$$\frac{number\ of\ objects\ in\ the\ cell\ at\ t_i}{total\ length\ of\ road\ segments\ in\ the\ cell}$$

The cell en-queuing process first accesses the histogram values of every cell for the same timestamp (line 2-3) and calculates its *adjusted density* (line 6). Each calculated adjusted density is compared with the system parameter cell density threshold (line 7-8). The ones that surpasses the threshold comparison are enqueued to a priority queue. The same process is performed for the all of the timestamps from the smallest (i.e., the earliest) to the highest.

Figure 7.4 illustrates an example of en-queuing process of the cells. Figure 7.4(a) shows each cell's histogram followed by the total length of road segments in the cell. For example, `[8, 12, 7, 3]` in cell A is the histogram and [4] is the total length of road segments covered by cell A. Figure 7.4(b) shows the placement of the candidate cells in the priority queue selected based on the histogram information

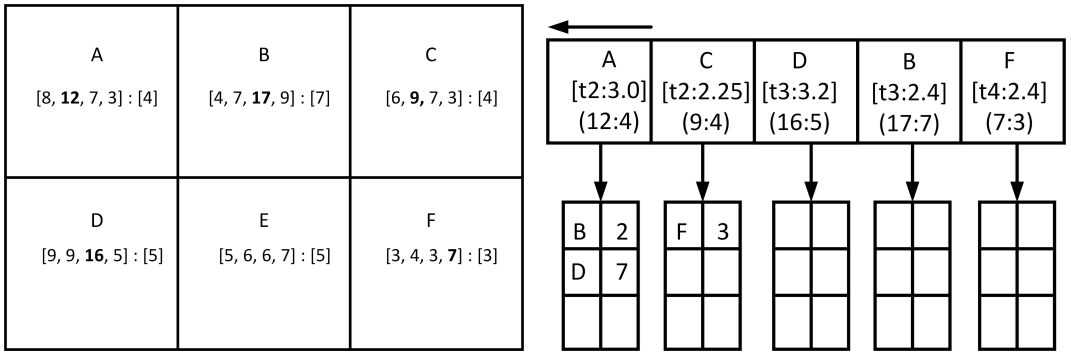
[6] R^D-tree was discussed in details in Chapter 4

Filtering Phase
Input : *Cell Histograms, quary time* t_q
Output : *a priority Queue* : Q

1. $t \leftarrow t_q$
2. **while** $t < t_q + H$ **do**
3. **for each cell** c **do**
4. $N \leftarrow$ c.number of vehicles at t
5. $l \leftarrow$ c.total road segment length
6. $adjusted\ density\ (d) \leftarrow \frac{N}{l}$
7. **if** $(d > threshold)$ **then**
8. $Q.add(c, d, t)$
9. $t = t + \Delta T$
10. **for each cell** c **in** Q **do**
11. findInfluencedCells(c)
12. **return Q**

Figure 7.3 Filtering Phase of the Predictive Line Query Algorithm

with an assumed *cell density threshold* of 2.1. Based on the histogram information, cell A has the highest priority (12/ 4 = 3) as it is the cell whose histogram has the earliest (at t_2) highest adjusted density. Both B's and D's histograms show similar potential at t_3. Among them, D gets the next priority in the queue since D has the next highest density (D's 16/5 = 3.2 $>$ B's 17/7 = 2.4).



(a) Vehicles Histogram and Road Segments per Cell for $t_1, t_2, t_3,$ and t_4

(b) Candidate Dense Cells in the Priority Queue with Density Threshold ≥ 10

Figure 7.4 Histogram and Candidate Dense cells

Each cell c in the queue maintains a list of influenced cells whose priority is lesser than its own, along with the number of vehicles coming from the cell c (line

11). The list is maintained to prevent repetitive access to the histogram to get the latest adjusted densities (This will be discussed in more detail later in this chapter).

7.3.2. The Refinement Phase. The refinement phase first dequeues the highest prioritized cell from the priority queue. The dequeued cell is then passed to the first of the three stages: coarse-grained, mid-grained, and the fine-grained. This procedure is repeated for the next highest prioritized cell in the same timestamp. If no cell is available for the same timestamp, the *Refreshing phase* is activated in which the entires in the queue and their priority would be changed. Then the highest prioritized cell from the updated queue is selected and passed through the refinement phase.

Refinement Phase
Input : *a priory queue with potential dense cells* : Q, *density threshold* ρ
Output : *a set of road dense segments*

1. **while** $(Q <> null)$ **do**
2. $E \leftarrow Q.poll$
3. $t_q \leftarrow E.t$
4. $c \leftarrow E.c$
5. $CenterX \leftarrow c.CentreX$
6. $CenterY \leftarrow c.CentreY$
7. $innerL \leftarrow \frac{c.length}{2} + (v_{min} \cdot t)$
8. $outerL \leftarrow \frac{c.length}{2} + (v_{max} \cdot t)$
9. $Edges_{ring} = predictive\ Squared\ Ring\ Query(CenterX, CenterY, t, innerL, outerL)$
10. **if** $(Edges_{ring} <> null)$
11. **for each** $e_i \in Edges_{ring}$
12. $Directions = getDirections(e_i, c)$
13. $Vehicles = Vehicles \bigcup getVehicles(e_i, Directions)$
14. **for** $eachvehicle v_i \in Vehicles$
15. $edge \leftarrow v_i.findRoadSegmentAt(t_q)$
16. **if** $edge_i \in c.edges$
17. $N_i \leftarrow (edge_i.N) + 1$
18. **if** $\frac{N_i}{l_i} > \rho$
19. $Densed\ Segments \leftarrow \bigcup edge_i$
20. return *Densed Segments*

Figure 7.5 Refinement Phase of the Predictive Line Query Algorithm

The *coarse-grained* stage finds the road segments that carry the vehicles that might travel through the cell at the querying time. This is performed in a similar way to the ring query first introduced in 5 with the exception of the ring shape

which is now a square shaped ring, called *square-shaped-ring* (instead of a circular shaped ring) as in line 9 in Figure 7.5.

The squared shaped ring query is graphically explained in Figure 7.6. The cell is shown in dashed lines. The square shaped ring is represented by the shaded area between solid-lined squares. The dimension of the *square-shaped-ring* is determined according to the road network information (line 7-8). The lengths $innerL$ and $outerL$ are the distances to the closest and the farthest vehicle that "*might*" be able to cross the cell according to the road speed limits.

The *mid-grained* stage employs the road segments obtained from the coarse-grained stage and retrieves vehicles in the relevant hash bucket of each road segment found in the coarse-grained stage (line 10-13). The relevancy is determined according to the geometric area formed by the two lines that begin at the mid point of the road segment and go through the outer most corners of the square (see Figure 7.7).

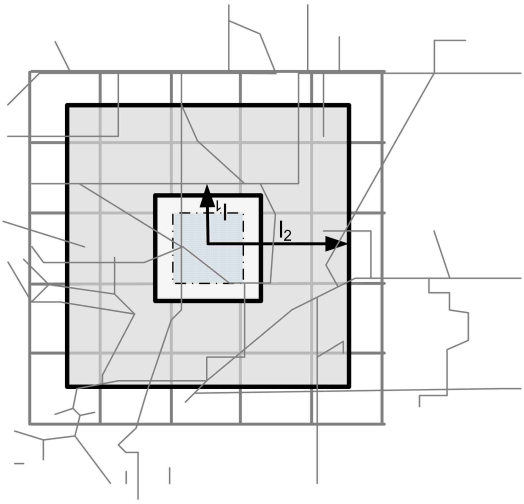


Figure 7.6 Squared Shaped Ring Query in the Coarse-Grained Filtering Phase

The corners of the square that the lines go through and the number of buckets correspondence with the geometric area depend on the time that the density is being looked for (in another term, distance to the cell from the mid point of the edge) and the total number of hash buckets set up. The Figure 7.7 illustrates two cases where the number of hash buckets are 8 and different distances to the querying cell (due to the difference in times the density is looking for: t_a and $t_b > t_a$). As the figure shows the number of hash buckets selected to examine

cell A's density is 3 (hash bucket 1, 2, and 3) where that for cell B is only bucket 0 and 1.

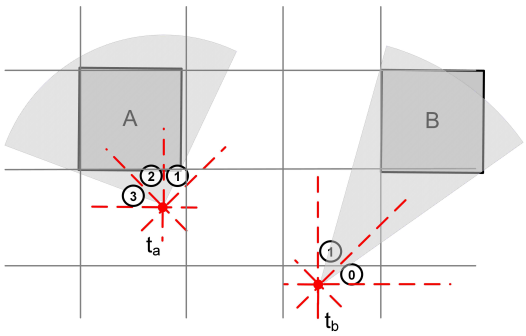


Figure 7.7 Two Examples of Modified Hash Bucket Selection

The bucket selection with respect to the cell provides two advantages: (i) it avoids multiple access to the same set of vehicles, which results in higher performance in terms of CPU time as well as page accesses. This is because the vehicles for all of the road segments in the cell is considered together, instead of considering vehicles per each edge where multiple considerations for the same bucket is possible (ii) it alleviates the mismatch between the geographical direction of vehicle's destination and the querying road segments, with respect to the vehicles current position. The mismatch may introduce false positives in the query result.

An example for bucket selection mismatch which explains its effect on false positives is illustrated in Figure 7.8. The filled dot V is the vehicle's current position, the star represents its destination, and the set of lines connected the dot and the star is the vehicle's tentative path. According to the vehicle's current position and the destination, the vehicle is stored in hash bucket 1 (because the dotted arrow lies on the area corresponding to bucket 1). Thus, the vehicle V would not be satisfied by the bucket selection and would not be taken for further consideration where its tentative path is examined. This omission would contribute to a false negative in the query results.

The *fine-grained* stage uses vehicle's shortest path and road speed limits to determine the vehicles that *will* be traveling on roads in the cell. This step also compares the vehicles in a way that each path needs to consider only once for all of the road segments in the cell. To implement this, each edge is associated with a vehicle count. Each vehicle's tentative path is then examined. The edge on which

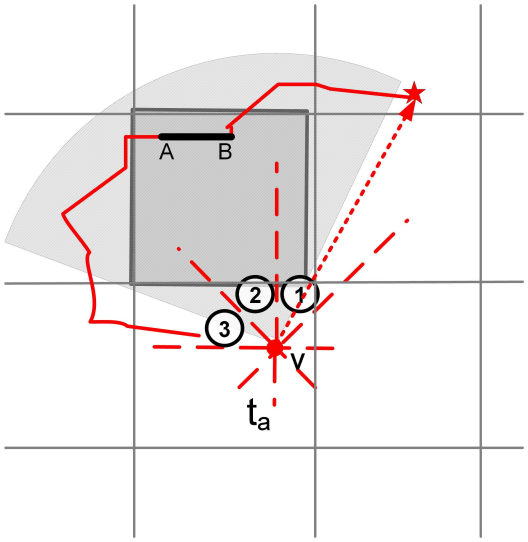


Figure 7.8 Bucket Selection Mismatch between Vehicle's Destination and Querying Road Segment

the vehicle's future position lies increases its vehicle count. After analyzing all of the vehicles, the edge's vehicle count is compared against the density threshold and reports the dense road segments, if any found.

The Refinement phase will repeat for the next element in the queue if it has the same timestamp as the previous cell. If no cell is available for the same timestamp, then the Refreshing phase is activated.

7.3.3. The Refreshing Phase. If dense areas were found in the filtering phase, the Refreshing phase perform two (independent) actions before the filtering phase is repeated: (i) rejuvenate the data by disengaging extraneous data (lines 1-5 in Figure 7.9) (ii) update the priority queue (lines 6- 16).

In the rejuvenate process, a quarantine area is defined for each identified dense road segment within the same timestamp. The area contains the dense road segments and the segments that the congestion would propagated to. The formal definition of quarantine area is defined in Definition 10. Value of n can be defined per each road segment or for the entire road network, depending on the behavior of its density propagation, based on the past data.

Definition 10. *[Quarantine Area] Given a road network $\mathcal{G}(\mathcal{V},\mathcal{E})$ and a set of dense road segments $\mathcal{S}$; where $\mathcal{S} \subset \mathcal{E}$. The quarantine are of $\mathcal{S}$ is a set of road segments, $\mathcal{Q} = \bigcup_{i \in |\mathcal{S}|}(\mathcal{S}_i \cup \mathcal{S}_i^n)$. Here $\mathcal{S}_i^n$ is the n -hop adjacent edges of $i^t h$ edge in $\mathcal{S}$.*

Refreshing Phase
Input : *a set of dense road segments* : *Densed Segments*, *priory queue* : *Q*
Output :

Rejuvenating data
1. **for each** *dense segment* s_i **in** *Densed Segments*
2. $quarantine_{segments} \leftarrow s_i.getQuarantineRoadSegments()$
3. $quarantine_{vehicles} \leftarrow s_i.getQuarantineVehicles()$
4. $updateRD - tree(quarantine_{segments})$
5. $updateVehicles(quarantine_{vehicles})$
Update Priority Queue
6. **for each** *denseCell* c_{now} **in** *denseCell list* **do**
7. **for each** c_{next} **in** c_{now}.*adjCell list* **do**
8. $t \leftarrow c_{now}.time$
9. $l \leftarrow c_{next}.total\ road\ segment\ length$
10. $N \leftarrow c_{next}.number\ of\ vehicles\ at\ (t + \Delta T)$
11. $N \leftarrow N - c_{next}.vehicles\ from\ c_{now}$
12. $adjusted\ density\ d \leftarrow \frac{N}{l}$
13. **if** $(d < threshold)$ **then**
14. $Q.remove(c_{next})$
15. **else**
16. $Q.update(c_{next})$

Figure 7.9 Refreshing Phase of the Predictive Line Query Algorithm

The quarantine areas for a road segment $\overline{AB}$ is illustrated in Figure 7.10. The road segment $\overline{AB}$ is the dense road segment and the dashed lined-road segments are the once with the propagation traffic effect. The vehicles on these road segments are then disregard from subsequent dense area identification.

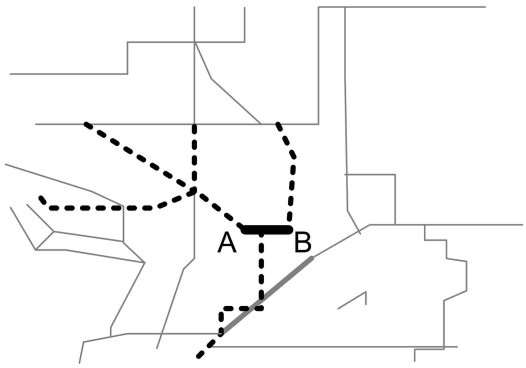
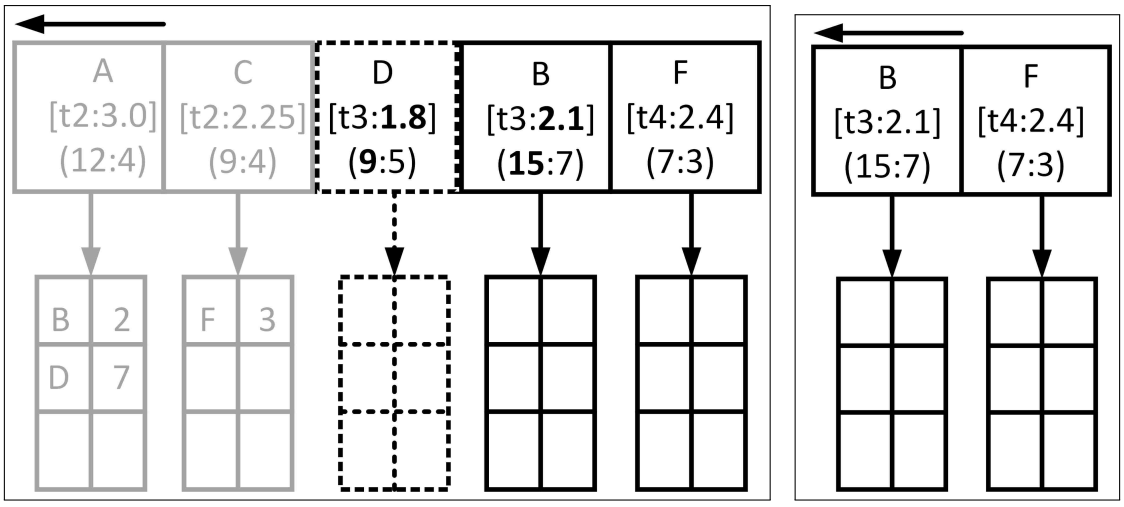


Figure 7.10 Example of the CPLQs That May Contain the Object

Queue update process utilizes the influence cell list associated with the cells in the queue. For each cell that contains a dense road segment, its influential cells' adjusted density is updated. The process is exemplified in Figure 7.11. Figure 7.11(a) shows the queue that was presented in Figure 7.4. The gray colored elements, Cell A and C, represent the cells that have been gone through the refinement phase. Assume that the refinement phase has identified one or more dense road segments and no dense segments in cell A and cell C, respectively. Since cell c has no dense road segments in in, it will not affect the queue. Cell A, however, has dense road segments in it, cells in its adjacent list (i.e, cell B and D) are updated. In fact, the adjacent list shows that 2 vehicles in cell A will next traverse to cell B. Since the vehicles in cell A are stopped due to the high density of cell A, the total number of expected vehicles in cell B would be decreased. The new adjusted density of cell B becomes 15/7 = 2.1. Similarly, cell D's new adjusted density is 1.8. According to the new values, cell D, marked in dashed lines, will be dropped from the queue and cell B will be moved to the front of the queue.



(a) Priority Queue Updating Process at t_3

(b) Updated Priority Queue at t_3

Figure 7.11 Queue Update in Refreshing Phase

7.4. PERFORMANCE STUDY

The system was implemented and tested on a 2.40 GHz Intel(R) Xeon(R) E5620 CPU desktop with 11 Gigabytes of memory. The page size was 4 k. The

implementation of R^D-tree adopted the $R*$-tree implementation of [66]. The internal nodes of a tree being pinned in a LRU buffer of 50 pages.

The proposed algorithm was evaluated on moving object data sets generated by the Brinkhoff generator [64]. The generator was fed with four different US state maps: IA, WA, AZ, and CA. The differences between states come in total land area, number of road segments, and average road segment length, which results in different mobile object distributions. The statistics of the chosen states are given in Table 7.1.

Table 7.1 Statistics of the Data Generator's Input Topologies

State	Land Area	Number of Road Segments	Average Road Segment Length
IA	55,857	3392	356
AZ	66,455	4935	383
WA	113,594	1442	628
CA	155,779	8062	225

The number of moving objects in each dataset ranges from 10K to 100K. Average traveling time of each data set was 60 minutes. The chosen input parameters and their values are presented in Table 7.2 with the default value is in bold.

Table 7.2 Simulation Parameters and Their Values for PDQ Algorithm

Parameters	Values
Number of mobile objects	10,20,30, . . . , **50**, . . . , 100 k
Road network topology	IA, AZ, **WA**, CA
Predictive time window (minutes)	10, 20, **30**, . . . , 60
Cell density threshold (ρ)	**0.05**, 0.1,0.15, . . . , 1
Road density threshold (ϵ)	0.5, **1**, 1.5, . . . , 3
Grid size (d)	**30**, 50, 70, 90
Vehicles equipped with the system	25%, 50%, 75%, **100%**
Step size (minutes)	2, **5**, 10, 15

The performance of the proposed algorithm (PDQ) is compared with that of a naive approach, the simple query (also known as SDQ) that evaluates each object's shortest path to estimate the future dense road segments. In the evaluation process, the SDQ also identifies the quarantine segments as of PDQ and discards from future evaluations.

The performance is measured in terms of number of dense segments found, error, and I/O cost. The *number of dense road segments* is the count of unique dense road segments. The *error* is reported in terms of false positives and false negatives with respect to the SDQ's result. Both PDQ and SDQ assume all objects to be in secondary disk. The *I/O cost* is the average (disk) page accesses per time step. PDQ first calculates the page accesses at each t-minute (t = 5 minutes, in default settings) step throughout the query life time (t_{step}). The average page accesses for the entire query life time is, then, calculated by taking the average of all t_{step}'s in the query life time. SQ accesses all the pages that the vehicles are stored.

Vehicles' shortest path is not assumed to be stored, they are calculated calculated on the fly. Thus, with 50 k objects (the default number of objects), and 20 bytes of space for each vehicle, it takes 250 page accesses to find dense road segment at one time stamp. SQ does not have the advantage of either the cache utilized or the quarantine areas identified. SQ would be constantly replacing pages in Cache since SQ would have to look at all of them. identified quarantine area, on the other hand, does not eliminate any object consideration as the object must be accessed to identify its quarantine status. Thus, SQ will have a constant page access count for a fix number of moving objects. However, its CPU cost (which has not been reported in this chapter, but expected to be done in future) will increase excessively.

The best fit default values of the system parameters, excluding the *vehicles equipped with the system* whose default value was fixed to 100%, were selected to minimize the cost of the system. The cost of the system is defined as the summation of the page accesses, false negatives, and the false positives. However, the pre-experimental results, generated with scaled-down input data set (i.e., with Worth county in Missouri state and other default input parameters) showed that the false positives are mostly independent on the system parameter setting and is relatively small as shown in Figure 7.12.

False negatives and page accesses, on the other hand, have inverse relationship each other as Figure 7.13(a) and 7.13(b) show. Thus, the cost model for the Linear Regression analysis was formed as in 22; where ρ, d , and α are the cell density threshold, grid size, and a parameter in the regression model, respectively. Table 7.3 shows different values for α, the best fit value of ρ, d, and their corresponding simulation results. The rest of the simulations selected α to be 0.05, which considers both page accesses and false negative equally important.

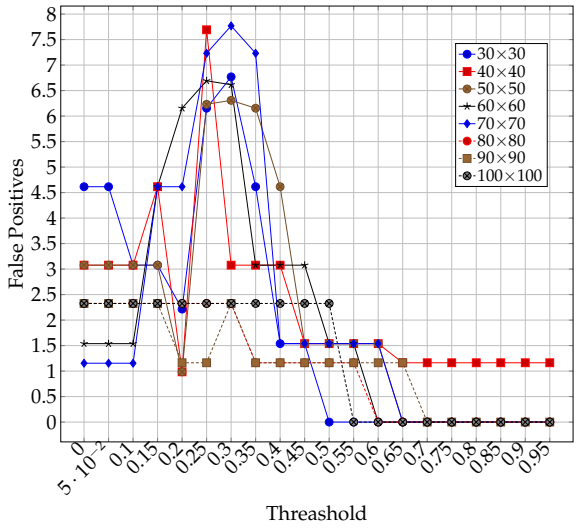


Figure 7.12 False Positives for Varying Grid Sizes, Cell Threshold and Worth County Road Network

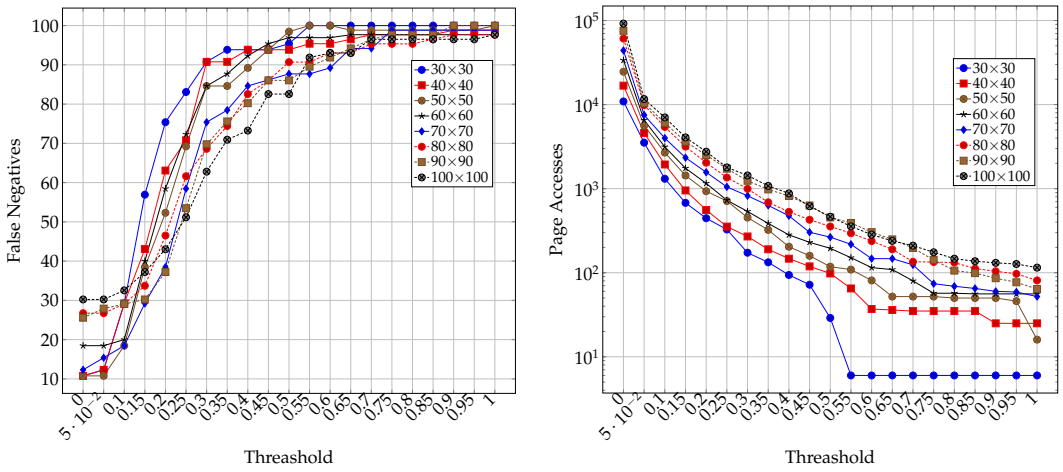
$$cost\ panelty = \alpha \cdot \rho + (1 - \alpha) \cdot d \tag{22}$$

Table 7.3 Results from Regression Analysis for different model parameter

α	ρ	d	False Negatives	False Positives	Page Accesses
0.1	0.1	30	0.292	0.031	262
0.2	0.05	30	0.123	0.046	705
0.3	0.05	30	0.123	0.046	705
0.4	0.05	30	0.123	0.046	705
0.5	0.05	30	0.123	0.046	705
0.6	0.05	50	0.108	0.031	1124
0.7	0.05	50	0.108	0.031	1124
0.8	0.05	50	0.108	0.031	1124
0.9	0.05	50	0.108	0.031	1124

7.4.1. Effect of Cell Density Threshold. Figure 7.14 illustrates the performance of DQ and SQ with respect to cell density threshold ρ. The performance of SQ does not change with the cell density threshold as SQ is independent on the grid. According to Figure 7.14(a), DQ algorithm, however, exhibit better I/O cost for higher cell thresholds. This is because the number of cells passes

from the filtering phase to refinement phase is smaller. The number of identified dense road segments, however, gets smaller producing poor performance. This is further explained in Figure 7.14(c). It shows that the false negatives has reached 100 % for thresholds bigger than 0.75. This is again due to the fewer cells pass to refinement phase. This prevents road segments to be identified dense, which would be otherwise if it was examining individual road segments.



(a) Vehicles at Time t_2 (b) Vehicles at Time t_2

Figure 7.13 Dynamic Nature of Continuous Traffic Prediction Information

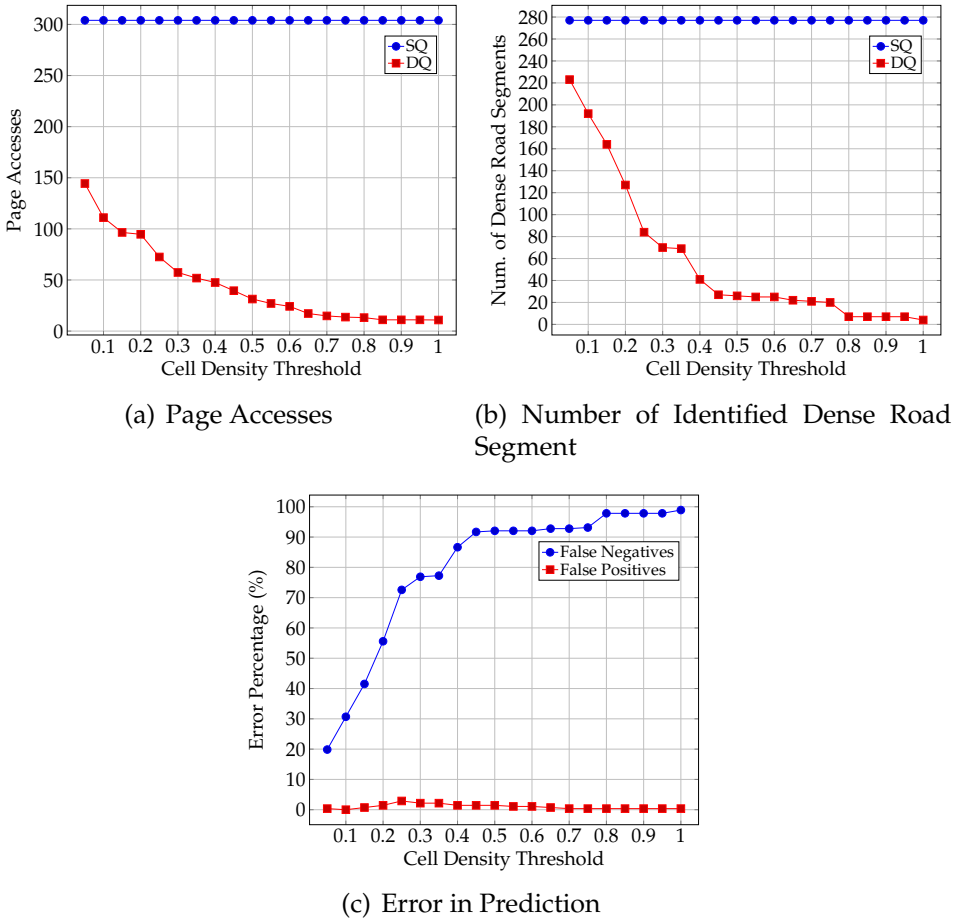
7.4.2. Effect of Road Density Threshold. The page access count increases as the road density threshold ϵ increases (Figure 7.15(a)). That is because, the higher ϵ means that the lesser number of road segments are identified as dense. This results in reduced number of quarantine road segments, which leaves more road segments for later dense road segment identification. Thus, the average page access per one time stamp becomes higher.

Due to the same reason mentioned before (i.e., the higher ϵ means that the lesser number of road segments are identified as dense), both DQ and SQ identifies fewer number of dense road segments as Figure 7.15(b). However, this decrement does not implies a higher false negatives. In fact, false negatives and positives remain almost stable with the road density threshold. This is exhibit in Figure 7.15(c).

7.4.3. Effect of Grid Size . The grid size also does not effect on the performance of SQ as shown in Figure 7.16. DQ has lower PA on smaller grids and

it get stable as the grid size is larger (i.e., smaller the cell size). The reason is that, when the grid size is small, lesser cell will be counted to have potential dense road segments. When the area of a cell is larger, adjusted density of that cell is lower than that of individual adjusted densities, if the large cell is divided into small cells. So, the lesser the number of cells passed to the refinement phase, the smaller the page accesses. Additionally, having smaller cells might have introduced multiple access to the same object lists where this effect might not exist with bigger cells. Both the number of dense road segments and the error rate is better when the grid has smaller cells.

7.4.4. Effect of Number of Mobile Objects. First note that the reported page accesses in Figure 7.17(a) is in the logarithmic scale. As the figure shows both SQ's and DQ's page accesses increase as the number of moving objects increase. But the rate of the increment in SQ is significant compared to that of DQ. SQ's

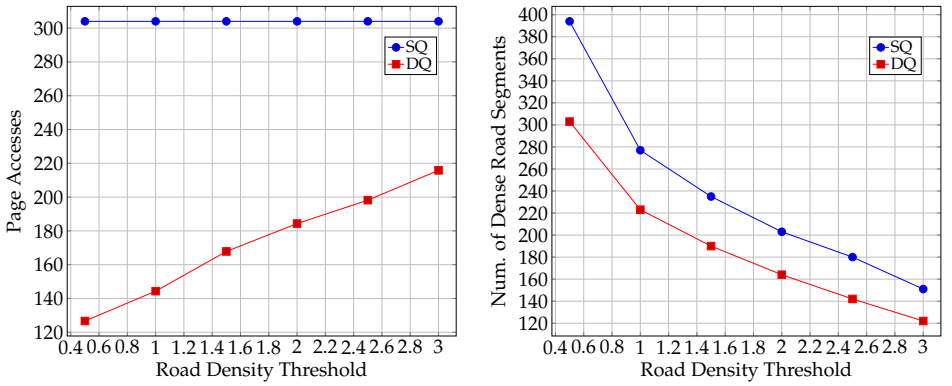


(a) Page Accesses

(b) Number of Identified Dense Road Segment

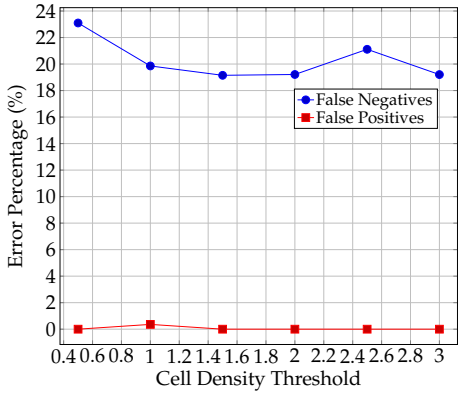
(c) Error in Prediction

Figure 7.14 Query Performance of PDQ with Varying Cell Density Threshold



(a) Page Accesses

(b) Number of Identified Dense Road Segment

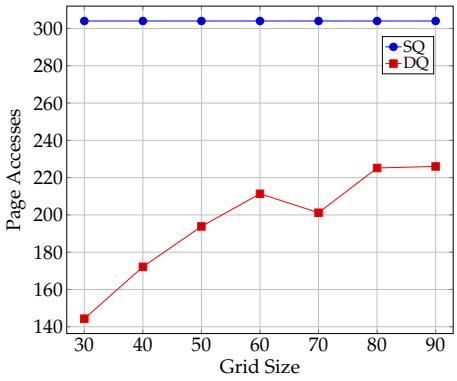


(c) Error in Prediction

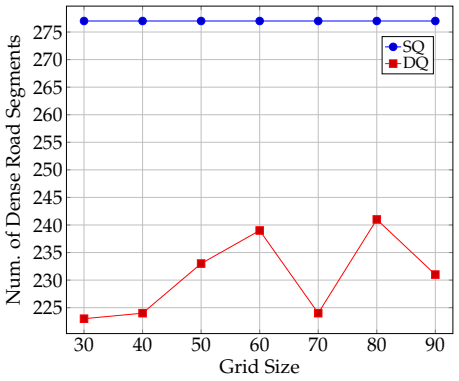
Figure 7.15 Query Performance of PDQ with Varying Road Density Threshold

increment is because of the increased number of shortest paths. DQ, on the other hand, will pass more cells to refinement phase. Since the filtering phase selects only a limited number of cells, DQ needs to consider objects in highest possible cells in the refinement phase compared to that of in SQ.

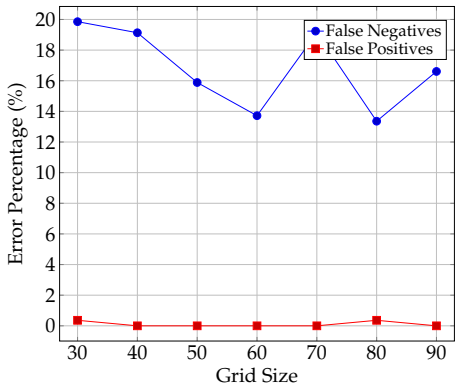
The number of identified dense road segments are illustrated in Figure 7.17(b). It shows that both SQ and DQ shows the same trend for the number of identified dense road segments with the number of moving objects. In fact, as the number of moving objects increases both algorithms find more dense segments. The fewer number of unidentified dense road segments in DQ, compared to SQ, would be eliminated with a reduced cell threshold which allows the filtering phase to pass more cells to the refinement phase. Recall that the default cell threshold and the grid size were chosen to equally weigh the importance of page accesses and the accuracy.



(a) Page Accesses



(b) Number of Identified Dense Road Segment



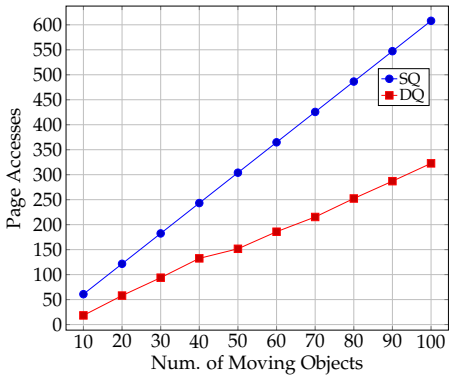
(c) Error in Prediction

Figure 7.16 Query Performance of PDQ with Varying Grid Size

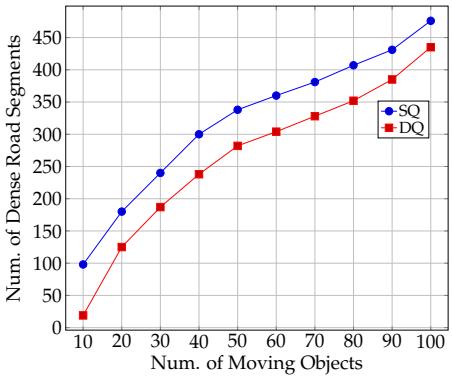
Confirming the assumption made by observing the pre-experimental results, DQ's false positives is negligibly small as shown in Figure 7.17(c). The number of false negatives, however, are big, especially for smaller number of objects. They get better, when the number of objects in the system is higher. This is because when the total number of objects is small, one miss could highly impact the precision negatively.

7.4.5. Effect of Road Network Topology. Regardless of the topology, the number of page accesses does not much vary on the topology that is being used. This is shown in Figure 7.18(a). However, the model formed shows that the page accesses depend on the area of the topology, number of road segments, and the road segment length.

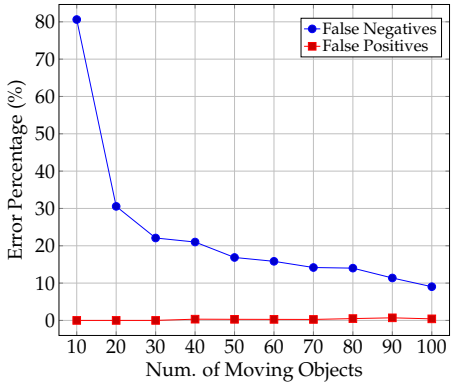
Both DQ and SQ shows the same behavior for finding the number of dense road segments for different topologies (Figure 7.18(b)). Here again, the system has



(a) Page Accesses



(b) Number of Identified Dense Road Segment



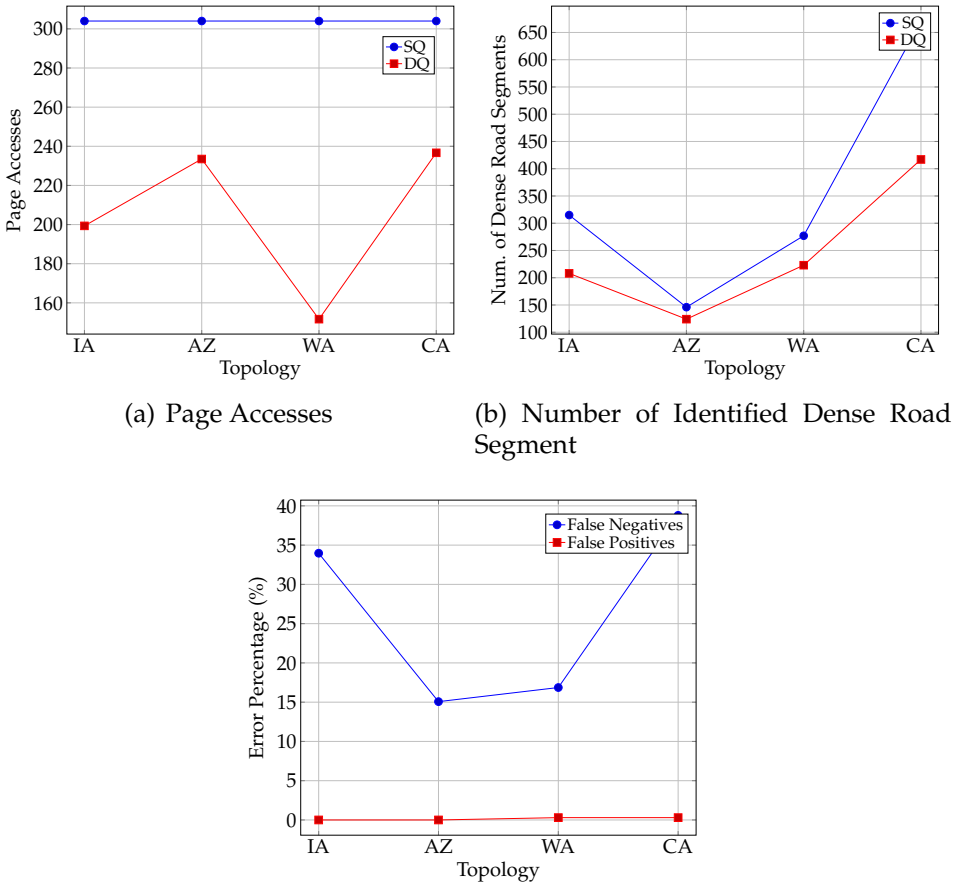
(c) Error in Prediction

Figure 7.17 Query Performance of PDQ with Varying Number of Mobile Objects

missed some dense road segments, which can be easily read with false negatives in Figure 7.18(c).

7.4.6. Effect of Percentage of Vehicles Equipped. The percentage of vehicles equipped with the system certainly will reduce the accuracy as it does not have information about some mobile objects to predict dense road segments more accurately. However, the system adjusts its parameters to cooperate the missed information. In fact, the cell and the road density thresholds are adjusted according to the missing vehicle percentage. For example, if the vehicles equipped percentage is 75 %, the default values, i.e., $\rho = 0.05$ and $\epsilon = 1$, are adjusted to $\rho = 0.0375$ and $\epsilon = 0.75$. The experiments were expanded to see this effect as well. The plot labeled *DQ w/o Adjust* is corresponds to the DQ with this adjustment.

As Figure 7.19(a) shows, the page access difference between two DQ algorithms gets smaller as the percentage of vehicles equipped the system and,



(a) Page Accesses

(b) Number of Identified Dense Road Segment

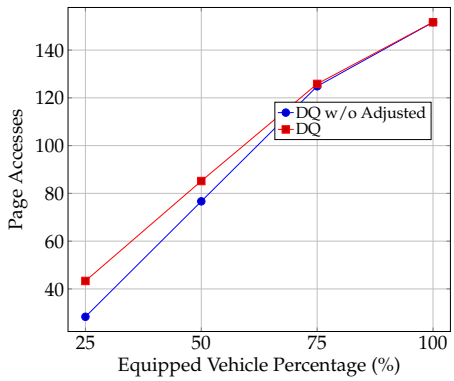
(c) Error in Prediction

Figure 7.18 Query Performance of PDQ with Varying Topology

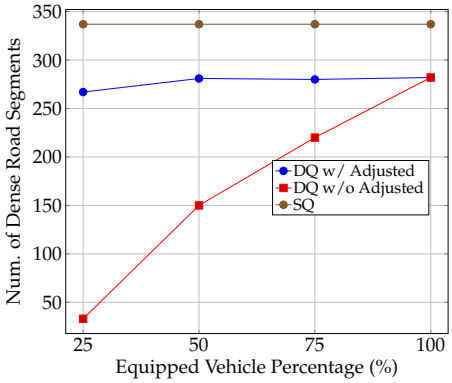
obviously, they become the same at 100 %. The difference between page access count had been introduced due to the threshold differences. Specifically, when the cell threshold is reduced, filtering phase passes more cells to the refinement phase increasing the page accesses. This penalty paid with page accesses can be surpassed by the improvement in the additional number of dense road segments found (shown in Figure 7.19(b)). This is affirmed by the false negative difference illustrated in Figure 7.19(c)

7.5. SUMMARY

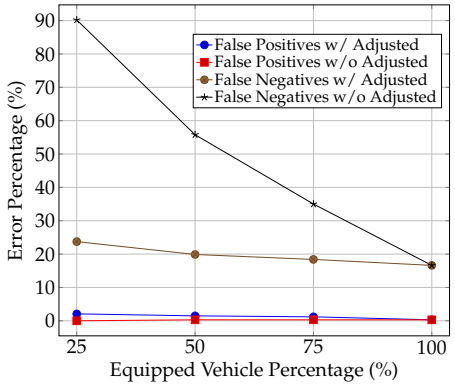
This chapter presented a proactive based approach to provide user with predicted traffic information. The information is provided by the query type named Predictive Density Query (PDQ). Contrast to the traditional density queries, PDQ has three key features. First, it identifies and reports dense areas in terms of



(a) Page Accesses



(b) Number of Identified Dense Road Segment



(c) Error in Prediction

Figure 7.19 Query Performance of PDQ with Varying the Percentage of Vehicles Equipped with the System

road segments considering the underlying road network. Secondly, it provides predictive density information in which users will find more practical than the current density information. Thirdly, it finds the mutually independent dense road segments, which are less, if not zero, impacted from previously dense road segments; hence, it provides a reliable result.

PDQ processing is supported by the R^D-tree and a 2-dimensional histogram. The R^D-tree maintains road network and mobile object information. The histogram maintains a summary of object information per cell of which the network is divided into. Query processing algorithm identifies dense road segments in three phases: filtering, refinement, and refreshing. The filtering phase extracts cells that are possible to have dense road segments by analyzing the histogram information. It en-queues those cells in a priority queue. The priority is being the timestamp and the adjusted density. The refinement phase de-queue each cell and extracts

necessary mobile object information to identify dense road segments within the cell. The refreshing phase, reorder the priority queue, if previously found dense road segments influence the cells in the queue.

Extensive list of parameters were experimentally tested to study their effect on the algorithm's performance. The study shows that the proposed algorithm gives better I/O cost than the naive approach. The PDQ algorithm, however, exhibit a small percentage of false negatives where false positives are nearly zero.

8. CONCLUSION

This research was conducted to develop a new indexing scheme targeting support for predictive as well as current queries on objects moving under the road network constraints and define and design predictive query algorithms.

8.1. CONTRIBUTIONS

- A mobile data indexing structure that index mobile object's information, know as R^D-*tree*, was developed in support of both current and future LDQs queries.
- An on-demand-based predictive query was proposed which provides commuters future traffic information. The query was termed a *Predictive Line Query*. Two versions of the query were considered: snapshot query and continuous query.
  - The snapshot query estimates future traffic condition of a user specified road segment based on the mobile object's current information and currently known future information, such as the destination. Three algorithms were developed to provide the query result.
    - Basic Algorithm
    - Enhanced Algorithm
    - Comprehensive Algorithm
  - The continuous query addresses the issues arose in snapshot query due to the volatility nature of individual mobile object behavior. Three algorithms were developed to monitor the changes of the snapshot query result that could occur as time evolves. The algorithms were:
    - Solo-Update Maintenance
    - Solo-Object Maintenance
    - Batch-Object Maintenance
- A proactive-based predictive query type namely *Predictive Density Query* was developed. The query considers the object's possible influence on other objects which were not considered in the Predictive Line Query. A query algorithm was developed to answer the Predictive Density Query.

- The extensive experiments performed on aforementioned contributions show the performance vise benefits of the proposed indexing structure as well as the supporting query algorithms.

8.2. FUTURE WORK

- Developing indexing structures for road networks that preserves the network connectivity would be a great problem to address. R-tree based indexing structures are common indexing structures that have been utilized to index road networks. These indexing structures were designed to support Euclidean based spatial queries and does not preserve the network connectivity. Hence, tree is not capable to return relevant network information effectively.
- The modern, advanced technologies embedded into the mobile objects have introduced high computational capabilities to them. Thus, studying the the proposed query types and their solutions in the distributed domain would bring more interesting research issues.

BIBLIOGRAPHY

[1] Brian McKenzie, "Out-of-State and Long Commutes: 2011 Report," https://www.census.gov/library/publications/2013/acs/acs-20.html, 2013.

[2] INRIX, "AMERICANS WILL WASTE $2.8 TRILLION ON TRAFFIC BY 2030 IF GRIDLOCK PERSISTS INRIX," http://www.inrix.com/press/americans-will-waste-2-8-trillion-on-traffic-by-2030-if-gridlock-persists/, 2014.

[3] the Statistics Portal, "the Statistics Portal," http://www.statista.com/statistics/218112/forecast-of-global-pnd-market-size-since-2005/, 2014.

[4] Y. N. Silva, X. Xiong, and W. G. Aref, "The RUM-Tree: Supporting Frequent Updates in R-Trees Using Memos," *The VLDB Journal*, vol. 18, no. 3, pp. 719–738, Jun. 2009.

[5] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," *SIGMOD Record*, vol. 29, no. 2, pp. 331–342, May 2000.

[6] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: an Optimized Spatio-Temporal Access Method for Predictive Queries," in *Proceedings of the 29th International Conference on Very Large Data Bases*, ser. VLDB '03, vol. 29. VLDB Endowment, 2003, pp. 790–801.

[7] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. Tsotras, "Indexing Mobile Objects on the Plane," in *Proceedings of 13th International Workshop on Database and Expert Systems Applications*, 2002, pp. 693–697.

[8] S. Saltenis and C. S. Jensen, "Indexing of moving objects for location-based services," in *Proceedings. 18th International Conference on Data Engineering*, ser. ICDE '02, 2002, pp. 463–472.

[9] J. M. Patel, Y. Chen, and V. P. Chakka, "STRIPES: An Efficient Index for Predicted Trajectories," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. ACM, 2004, pp. 635–646.

[10] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and Update Efficient B$^+$-tree based Indexing of Moving Objects," in *Proceedings of the 30th International Conference on Very Large Data Bases*, ser. VLDB '04, vol. 30. VLDB Endowment, 2004, pp. 768–779.

[11] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento, "ST2B-tree: a self-tunable spatio-temporal b+-tree index for moving objects," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management Of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 29–42.

[12] M. L. Yiu, Y. Tao, and N. Mamoulis, "The Bdual-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space," *The VLDB Journal*, 2008.

[13] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh, "A road network embedding technique for k-nearest neighbor search in moving object databases," in *Proceedings of ACM international symposium on Advances in geographic information systems*, 2002.

[14] H.-J. Cho and C.-W. Chung, "An Efficient and Scalable Approach to CNN Queries in a Road Network," in *Proceedings of the 31st international conference on Very large data bases*, 2005.

[15] K.-S. Kim, S.-W. Kim, T.-W. Kim, and K.-J. Li, "Fast Indexing and Updating Method for Moving Objects on Road Networks," in *Web Information Systems Engineering Workshops, 2003. Proceedings. Fourth International Conference on*, 2003.

[16] J. Chen, X. Meng, Y. Guo, S. Grumbach, and H. Sun, "Modeling and Predicting Future Trajectories of Moving Objects in a Constrained Network," in *Proceedings of the 7th International Conference on Mobile Data Management*, ser. MDM '06, 2006.

[17] J.-D. Chen and X.-F. Meng, "Indexing Future Trajectories of Moving Objects in a Constrained Network," *Journal of Computer Science and Technology*, 2007.

[18] K. S. Bok, H. W. Yoon, D. M. Seo, M. H. Kim, and J. S. Yoo, "Indexing of Continuously Moving Objects on Road Networks," *IEICE - Trans. Inf. Syst.*, 2008.

[19] J. Feng, J. Lu, Y. Zhu, and T. Watanabe, "Index Method for Tracking Network-Constrained Moving Objects," in *Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part II*, 2008.

[20] J. Chen and X. Meng, "Update-Efficient Indexing of Moving Objects in Road Networks," *Geoinformatica*, December 2009.

[21] L. Guohui, L. Yanhong, L. Jianjun, L. Shu, and Y. Fumin, "Continuous Reverse K Nearest Neighbor Monitoring on Moving Objects in Road Networks," *Inf. Syst.*, 2010.

[22] "Broadcasting a Means to Disseminate Public Data in a Wireless Environment Issues and Solutions," ser. Advances in Computers, M. Zelkowitz, Ed. Elsevier, 2006, vol. 67, pp. 1–84.

[23] J. Zhang and L. Gruenwald, "Spatial and Temporal Aware, Trajectory Mobility Profile based Location Management for Mobile Computing," in *Proceedings of the 13^{t}h International Workshop on Database and Expert Systems Applications*, 2002, pp. 716–720.

[24] H. Wang and R. Zimmermann, "Snapshot Location-based Query Processing on Moving Objects in Road Networks," in *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, ser. GIS '08, 2008.

[25] D. Kwon, S. Lee, and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree," in *Proceedings of the 3rd International Conference on Mobile Data Management*, 2002, pp. 113–120.

[26] Y. Tao, D. Papadias, and X. Lian, "Reverse KNN Search in Arbitrary Dimensionality," in *VLDB*, 2004, pp. 744–755.

[27] L. Qin, J. X. Yu, B. Ding, and Y. Ishikawa, "Monitoring Aggregate K-NN Objects in Road Networks," in *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, ser. SSDBM '08, 2008.

[28] Y. Xia and S. Prabhakar, "Q+Rtree: Efficient Indexing for Moving Object Databases," in *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, 2003.

[29] C. Jensen, D. Lin, B. C. Ooi, and R. Zhang, "Effective density queries on continuouslymoving objects," in *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, 2006.

[30] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica*, 1974.

[31] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, June 1984.

[32] S. Chen, C. S. Jensen, and D. Lin, "A Benchmark for Evaluating Moving Object Indexes," *Proc. VLDB Endow.*, 2008.

[33] D. Pfoser and C. S. Jensen, "Indexing of network constrained moving objects," in *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, 2003.

[34] E. Frentzos, "Indexing Objects Moving on Fixed Networks," in *Advances in Spatial and Temporal Databases*, ser. Lecture Notes in Computer Science, T. Hadzilacos, Y. Manolopoulos, J. Roddick, and Y. Theodoridis, Eds. Springer Berlin Heidelberg, 2003.

[35] V. T. De Almeida and R. H. Güting, "Indexing the Trajectories of Moving Objects in Networks*," *Geoinformatica*, 2005.

[36] J. Feng, J. Lu, Y. Zhu, N. Mukai, and T. Watanabe, "Indexing of Moving Objects on Road Network Using Composite Structure," in *Knowledge-Based Intelligent Information and Engineering Systems*, 2007.

[37] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, "Continuous Nearest Neighbor Monitoring in Road Networks," in *Proceedings of the 32nd international conference on Very large data bases*, ser. VLDB '06, 2006.

[38] P. Fan, G. Li, L. Yuan, and Y. Li, "Vague continuous K-nearest neighbor queries over moving objects with uncertain velocity in road networks," *Information Systems*, 2012.

[39] J. Le, L. Liu, Y. Guo, and M. Ying, "Supported High-Update Method on Road Network," in *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on*, 2008.

[40] H. Kejia and L. Liangxu, "Efficiently Indexing Moving Objects on Road Network," in *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 2009.

[41] D. Stojanovic, A. N. Papadopoulos, B. Predic, S. Djordjevic-Kajan, and A. Nanopoulos, "Continuous range monitoring of mobile objects in road networks," *Data Knowl. Eng.*, 2008.

[42] H. Jeung, M. L. Yiu, X. Zhou, and C. S. Jensen, "Path Prediction and Predictive Range Querying in Road Network Databases," *The VLDB Journal*, 2010.

[43] J. Dai and C.-T. Lu, "DIME: Disposable Index for Moving Objects," in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, 2011.

[44] H. Wang and R. Zimmermann, "Processing of Continuous Location-Based Range Queries on Moving Objects in Road Networks," *Knowledge and Data Engineering, IEEE Transactions on*, 2011.

[45] Y. Cai and K. A. Hua, "An Adaptive Query Management Technique for Real-Time Monitoring of Spatial Regions in Mobile Database Systems," in *Proceedings of the Performance, Computing, and Communications Conference, 2002. on 21st IEEE International*, 2002.

[46] Y. Cai, K. Hua, and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," in *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, 2004.

[47] H. Hu, J. Xu, and D. L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries Over Moving Objects," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05, 2005.

[48] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch, "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects," *IEEE Trans. Comput.*, 2002.

[49] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05, 2005.

[50] T. Xia and D. Zhang, "Continuous Reverse Nearest Neighbor Monitoring," in *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, 2006.

[51] S. Nutanong, E. Tanin, J. Shao, R. Zhang, and R. Kotagiri, "Continuous detour queries in spatial networks," *Knowledge and Data Engineering, IEEE Transactions on*, 2012.

[52] D. Stojanovic, S. Djordjevic-Kajan, A. N. P. B. Predic, and A. Nanopoulos, "Continuous Range Query Processing for Network Constrained Mobile Objects," in *8th International Conference on Enterprise Information Systems (ICEIS)*, 2006.

[53] B. Gedik and L. Liu, "MobiEyes: A Distributed Location Monitoring Service Using Moving Location Queries," *Mobile Computing, IEEE Transactions on*, 2006.

[54] X. Xiong, M. F. Mokbel, and W. G. Aref, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05, 2005.

[55] R. Nehme and E. Rundensteiner, "SCUBA: Scalable Cluster-Based Algorithm for Evaluating Continuous Spatio-temporal Queries on Moving Objects," in *Advances in Database Technology - EDBT 2006*, 2006.

[56] F. Liu and K. A. Hua, "Moving Query Monitoring in Spatial Network Environments," *Mob. Netw. Appl.*, 2012.

[57] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao, "A Threshold-Based Algorithm for Continuous Monitoring of K Nearest Neighbors," *IEEE Trans. on Knowl. and Data Eng.*, 2005.

[58] D. G. Marios Hadjieleftheriou, George Kollios and V. J. Tsotras, "On-Line Discovery of Dense Areas in Spatio-temporal Databases," in *International Symposium on Advances in Spatial and Temporal Databases, SSTDn*, 2003.

[59] J. Ni and C. Ravishankar, "Pointwise-Dense Region Queries in Spatio-temporal Databases," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, 2007.

[60] X. Hao, X. Meng, and J. Xu, "Continuous Density Queries for Moving Objects," in *Proceedings of the Seventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, ser. MobiDE '08, 2008.

[61] J. Wen, X. Meng, X. Hao, and J. Xu, "An Efficient Approach for Continuous Density Queries," *Frontiers of Computer Science*, vol. 6, no. 5, 2012.

[62] C. Lai, L. Wang, J. Chen, X. Meng, and K. Zeitouni, "Effective Density Queries for Moving Objects in Road Networks," in *Proceedings of the joint 9th Asia-Pacific web and 8^{th} international conference on web-age information management conference on Advances in data and web management*, ser. APWeb/WAIM'07, 2007.

[63] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[64] T. Brinkhoff. (2004) A Framework for Generating Network-based Moving Objects. http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator.

[65] L. Heendaliya, D. Lin, and A. Hurson, "Predictive Line Queries for Traffic Forecasting," *Database and Expert Systems Applications*, 2012.

[66] E. Achtert, H. Kriegel, E. Schubert, and A. Zimek, "Interactive data mining with 3d-parallel-coordinate-trees," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

VITA

Lasanthi Nilmini Heendaliya was born in the city of Rathnapura, Sri Lanka. She attended the Convent of the Child Jesus, Ratnapura for her primary education and later attended the Ferguson Girls High School for her secondary education specialized in Mathematics. Lasanthi completed the island-wide Sri Lanka GCE Advanced Level (A/L) exam in 2000 that earned her the opportunity to attend the Engineering School at the University of Peradeniya, Sri Lanka starting October 2001. Lasanthi completed her undergraduate degree in Computer Engineering in January 2006.

In Fall 2007, she enrolled in the MS program in the department of Computer Science at the St. Cloud State University in St. Cloud, Minnesota. Lasanthi graduated with her MS degree in Fall 2009 and joined the Ph.D. program in Computer Science at the Missouri University of Science and Technology (Missouri S&T) in Rolla, Missouri. Her graduate research advisor is Dr. Ali Hurson. She received her Ph.D. in Computer Science from Missouri S&T in May 2015.

Another conference paper has been submitted for review. Lasanthi's primary research was in the mobile data indexing and querying. She has published two conference papers and two journal papers within the work conducted at Missouri S&T. Her other research interests are in the area of Mobile databases, Big Data, Distributed Computing, and Parallel Computing. Her experience at Missouri S&T expands to the teaching career as well. She served the Computer Science Department as a teaching assistance for more than 5 consecutive semesters. Her teaching interests includes, Computer Programming, Database Management Systems, Data Structures and Algorithms, Operating Systems, Calculus, Linear Algebra, and Numerical Methods.