View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

INDEXING AND QUERYING MOVING OBJECTS DATABASES

DAN LIN

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

My foremost thank goes to my supervisor Prof. Beng Chin Ooi. Without him, this thesis would not have been possible. I appreciate his vast knowledge in many areas, and his insights, suggestions and guidance that helped to shape my research skills.

I would like to thank Prof. Christian S. Jensen and Prof. Elisa Bertino for their patience and valuable advice during my internship. I would also like to thank Dr. Mong Li Lee, Dr. Zhiyong Huang and Dr. Chee Yong Chan for their help when I started my graduate student life.

I thank all the students in the database lab, whose presences and fun-loving spirits made the otherwise grueling experience tolerable. I enjoyed all the vivid discussions we had on various topics and had lots of fun being a member of this fantastic group. I specifically thank Hua Lu and Linhao Xu for their contributions to the system development as presented in this thesis.

Last but not least, I thank my family for always being there when I needed them most, and for supporting me through all these years.

CONTENTS

Acknowledgement **ii**

Summary **xii**

1 Introduction **1**

1.1 Moving Objects Databases 3

1.1.1 Indexing Moving Objects 4

1.1.2 Querying Moving Objects 5

1.1.3 Privacy Issues 6

1.2 Objectives and Contributions of This Thesis 7

1.2.1 Contributions on Index Structures 8

1.2.2 Contributions on Density Queries 9

1.2.3 Contributions on Protecting Location Privacy in Moving-Object Environments 10

1.2.4 Contributions on Extending a DBMS 10

1.3 Outline of The Thesis 11

2 Literature Review **13**

2.1 Traditional Indexes in Spatial Databases 13

2.2 Moving Objects Indexes 16

2.2.1 Indexing historical movement 17

2.2.2 Indexing current and future movement 18

2.3 Queries on Moving Objects 25

2.3.1 Range Query 25

2.3.2 K-nearest Neighbor Query 26

2.3.3 Density Query 30

2.4 Concurrency in Indexes 31

2.5 Approaches for Location Privacy Protection 33

2.6 Summary 35

3 The B^x-tree: Query and Update Efficient B$^+$-tree Based Indexing of Moving Objects **36**

3.1 Synopsis of Our Proposal 37

3.2 Structure and Algorithms 39

3.2.1 Index Structure 39

3.2.2 Querying 44

3.2.3 Insertion, Deletion, and Migration 54

3.3 Performance Studies 56

3.3.1 Experimental Settings 56

3.3.2 Filter Rate 57

3.3.3 Number of Sub-intervals,n 60

3.3.4 Range Query 62

3.3.5 kNN Query 69

3.3.6 Update 70

3.3.7 Effect of Concurrent Accesses . . . 73
3.3.8 Storage Requirements . . . 74
3.4 Summary . . . 75

4 Effective Density Queries on Moving Objects 77
4.1 Motivation . . . 78
4.2 Problem Statement . . . 80
4.3 The MODQ Framework . . . 83
4.4 Density Computation . . . 84
4.4.1 Overview . . . 84
4.4.2 Density Histogram . . . 85
4.4.3 Query Processing . . . 90
4.5 Performance Studies . . . 95
4.5.1 Experimental Settings . . . 95
4.5.2 DCT Compression Accuracy . . . 96
4.5.3 Density Queries . . . 100
4.5.4 Maintenance Cost . . . 106
4.6 Summary . . . 107

5 Location Privacy in Moving-Object Environments 108
5.1 Synopsis of Our Proposal . . . 109
5.1.1 Comparison to Existing Approaches . . . 112
5.2 The Strategies and the Architecture of the Location Privacy Protection System . . . 113
5.3 Algorithms . . . 116
5.3.1 Data Transformation . . . 116
5.3.2 Updates . . . 123

5.3.3 Queries 126

5.4 System Analysis 130

5.4.1 Privacy 130

5.4.2 Communication Cost 135

5.5 Performance Studies 136

5.5.1 Experimental Settings 136

5.5.2 Range Queries 137

5.5.3 K Nearest Neighbor Query 145

5.5.4 Update 146

5.6 Summary 148

6 Adapting Relational Database Engine to Accommodate Moving Objects 150

6.1 System Overview 151

6.1.1 The SpADE Client 152

6.1.2 The SpADE Server 153

6.1.3 Client/Server Protocols in SpADE 154

6.2 System Implementation 155

6.2.1 Data Modelling and the B^x-tree 155

6.2.2 Implementation Issues 156

6.3 Performance Studies 162

6.4 Summary 163

7 Conclusions and Future Work 164

7.1 Conclusions 164

7.2 Future work 166

LIST OF TABLES

3.1 Parameters and Their Settings 58

4.1 Parameters and Their Settings 96

5.1 Parameters and Their Settings 136

6.1 Moving Object Relation Scheme 157

LIST OF FIGURES

1.1 An Overview of Our Study 2

2.1 An Example of the R-tree Structure 14

2.2 An Example of the Quadtree Structure 16

2.3 An Example of the TPR-tree 22

2.4 An Example of the Constrained Range Query 26

2.5 An Example of Nearest Neighbor Search 27

2.6 An Example of the Constrained kNN Query 29

3.1 Space-Filling Curves 40

3.2 B^x-Tree with $n = 2$ Phases 42

3.3 Query Window Enlargement 45

3.4 Possible Positions of a Query Interval w.r.t. a Label Timestamp . . 47

3.5 Time Length Enlargement 48

3.6 “Jump” in the Index 49

3.7 Range Query Algorithm 50

3.8 Function TimeParameterizedRegion() 51

3.9 kNN Query Algorithm 53

3.10 B^x-Tree Evolution 55

3.11 Filter Rates for Varying Query Time 59

3.12 Filter Rates for Varying Query Window Size 60

3.13 Range Query Performance for Varying n and Query Time 61

3.14 Average Range Query Performance for Varying n 62

3.15 Effect of Varying Buffer Size 63

3.16 Effect of Varying Query Time 64

3.17 Effect of Query Window Size 65

3.18 Effect of Varying Query Interval Length 66

3.19 Effect of Maximum Speed on Range Query Performance 67

3.20 Effect of Data Distribution on Range Query Performance 67

3.21 Effect of Data Sizes on Range Query Performance 68

3.22 Effect of k on kNN Query Performance 69

3.23 Effect of Varying Update Time on the Update Cost 70

3.24 Effect of Varying Maximum Update Interval on Update Performance 71

3.25 Effect of Data Sizes on Update Cost 72

3.26 Effect of Concurrent Operations 74

3.27 Storage Requirement 75

4.1 An Example of Density Query Results 78

4.2 An Example of Answer Loss 79

4.3 Overlapping vs. Non-overlapping Regions in a Density Query 81

4.4 Problem Parameters 82

4.5 An Example of the DCT 86

4.6 DH Maintenance Algorithm 89

4.7 Maintenance in DH 90

4.8 Intersection between the Final Answer and DH Cells 90
4.9 Density Query Algorithm 91
4.10 Conflicting Types of Cells 92
4.11 Refinement Algorithm 94
4.12 DCT Compression Accuracy 96
4.13 False Positives and Negatives for Varying DCT Coefficients 97
4.14 False Positives and Negatives with Elapsed Time 98
4.15 Effect of the Error Factor and DCT Coefficients 99
4.16 Density Query Example 101
4.17 Histogram vs. Non-histogram 102
4.18 The MODQ vs. the DCF 103
4.19 Effect of Density Threshold and Query Size 103
4.20 Effect of Database Size 105
4.21 Effect of Data Distribution 105
4.22 Maintenance Cost 106

5.1 LPP System Overview 113
5.2 An Example of Position Transformation 118
5.3 Multiple Transformation Generation Algorithm 120
5.4 Super Query 121
5.5 Update Algorithm 124
5.6 An Example of Update Operation 125
5.7 An Example of Query Operation 126
5.8 Range Query Algorithm 127
5.9 Original Data vs. Transformed Data 134
5.10 False Positive Rate for Varying λ 137
5.11 False Positive Rate for Varying Query Size 138

5.12 False Positive Rate 139
5.13 Impact of Data Sizes on Range Query Performance 140
5.14 Query Cost of One Agent with Varying the Data Size 141
5.15 Impact of Number of Agents on Range Query Performance 141
5.16 Query Cost of One Agent for varying number of agents 142
5.17 Impact of Number of Agents and Data Sizes on Range Query Performance 143
5.18 Impact of Query Size on Range Query Performance 143
5.19 Query Cost of One Agent for Varying the Query Size 144
5.20 Impact of Skewed Data on Range Query Performance 145
5.21 Impact of k on kNN Query Performance 146
5.22 Impact of Data Sizes on Update Performance 147
5.23 Effect of Number of Agents on Update Performance 148
5.24 Effect of Data Distribution on Update Performance 149

6.1 System Architecture 152
6.2 Execution of a Spatial-temporal Query 158
6.3 Query and Update Performance of SpADE System 162

Summary

With the rapid developments in positioning technologies such as the Global Positioning System (GPS) and wireless communications, tracking of continuously moving objects has become feasible in terms of technology and implementation cost. However, this recent development poses new challenges to traditional database technology. In particular, traditional database systems have not been designed to support high update load due to object agility, predictive and spatio-temporal based query processing, and location privacy protection. In this thesis, we address three important basic issues in moving objects databases: indexing, querying and location privacy protection. The main design criteria of the algorithms and data structures is cost effective integration into an existing DBMS. In this connection, we extend an existing RDBMS, MySQL, to include a new indexing mechanism and query processing strategies with minimal alteration to existing codes.

In moving object applications, large quantities of location samples obtained via sensors are streamed to a database. Disclosure of new positions cause updates on the database, and objects are stored as snapshots taken at different times, and queries against such objects involve interpolation of new positions based on the

current position, velocity, and the valid time of the objects. To facilitate fast location of spatial objects for efficient update and querying, an efficient index must be designed to meet both objectives, fast update and retrieval. Indexes based on minimum bounding regions (MBRs) such as the R-tree exhibit high concurrency overheads during node splitting, and each individual update is known to be quite costly. This motivates us to design a solution that enables the B$^+$-tree to efficiently manage moving objects. We represent moving-object locations as vectors that are timestamped based on their update time. By applying a novel linearization technique to these values, it is possible to index the resulting values using a single B$^+$-tree that partitions values according to their timestamp and otherwise preserves spatial proximity. We develop algorithms for range and k nearest neighbor queries. The proposal can be grafted into existing database systems cost effectively. An extensive experimental study was conducted to evaluate the performance characteristics of the proposal and the results show that it substantially outperforms the R-tree based TPR*-tree for both single and concurrent access scenarios.

With the aid of the advanced indexing techniques, more complex queries can be supported in the location-based services. In this thesis, we study an emerging query, *density query*, which is designed to identify dense regions such as regions with high possibilities of a traffic jam. Specifically, we define a particular type of density query which reports all evidence of dense regions, and then we proceed to propose an algorithm for the efficient computation of density queries. While we use the B^x-tree as the underlying structure, the algorithm is independent of any structure. We conduct an extensive experimental study to evaluate the performance of the algorithm, and the results confirm the efficiency of the proposed algorithm.

The expanding use of location-based services has profound implications on the privacy of personal information. If no adequate protection is adopted, information

about movements of specific individuals could be disclosed to unauthorized subjects or organizations, thus resulting in privacy breaches. Therefore, we propose a framework for preserving location privacy in moving-object environments. Our approach is based on the idea of sending to the service provider suitably modified location information. Modifications such as transformations by scaling are performed by agents interposed between users and service providers. Agents execute data transformation and the service provider directly processes the transformed dataset. Our technique not only prevents the service provider from knowing the exact locations of users, but also protects information about user movements and locations from being disclosed to other users who are not authorized to access this information. A key characteristic of our approach is that it achieves privacy without degrading service quality. We also define metrics to quantify the privacy properties for our framework, and examine our approach experimentally.

Based on our proposal, we extend an open source database system MySQL to provide the required functionalities for managing moving objects. The most important feature of our system is that we do not infiltrate into the MySQL core. That is, the proposed indexing structure and algorithms could be crafted into most existing DBMS backend cost effectively.

To sum up, we have made contributions in addressing three core problems in moving objects databases and extending an existing DBMS to provide necessary and efficient support for location based services.

CHAPTER 1

Introduction

Spatial databases have been extensively studied in the last two decades resulting in numerous conceptual models, multi-dimensional indexes and query processing techniques. In these traditional spatial databases, spatial data objects are usually assumed to be fairly static, which impedes the direct migration of these techniques to an emerging area – the moving objects database (MOD).

With the advances in positioning technologies such as GPS and rapid developments of wireless communication devices, it is now possible to track continuously moving objects such as vehicles, users of wireless devices and goods. A wide range of applications related to moving objects have been developed. For instance, in an intelligent traffic control system, if we store information about locations of vehicles, congestion may be alleviated by diverting some vehicles to alternate routes, and taxis may be dispatched quickly to passengers. Another interesting example is location-based digital game where the positions of the mobile users play a central role. In such kind of games, players need to locate their nearest neighbors to fulfill

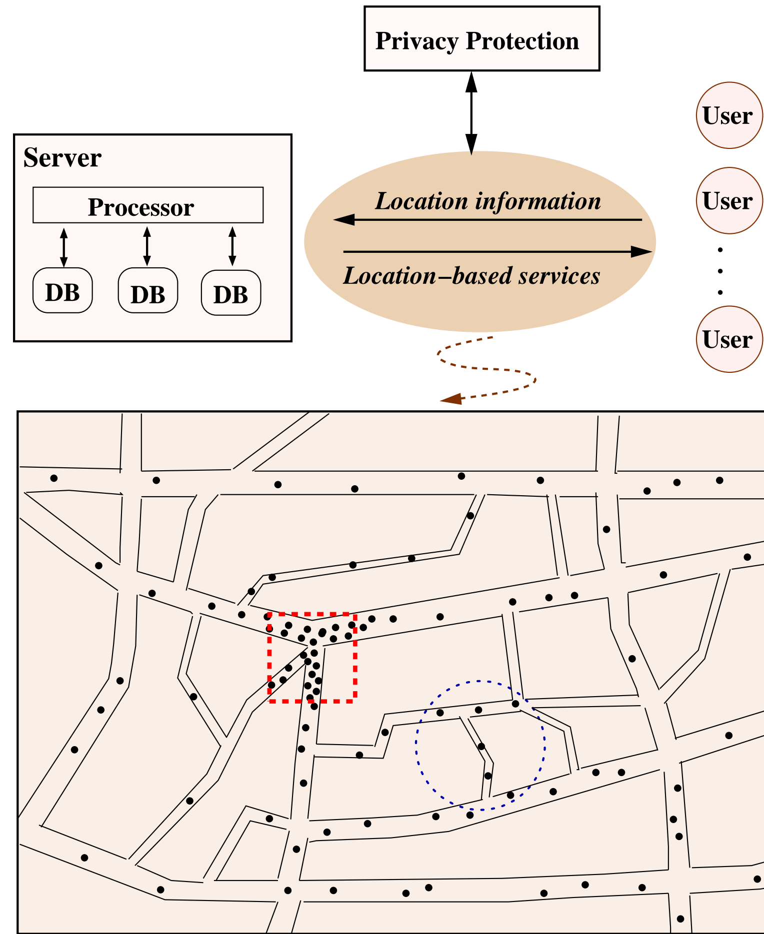


Figure 1.1: An Overview of Our Study

"tasks" such as "shooting" other close players via their mobile devices. MOD technique is also very important in the military. With the help of the MOD techniques, helicopters and tanks in the battlefield may be better positioned and mobilized to the maximum advantage.

New MOD applications engenders new technical challenges [95] which cannot be met by existing DBMS. Research issues such as data uncertainty, data imprecision, data modelling, representation by query language, simulation test bed, indexing techniques, querying techniques and location privacy need to be examined. Among them, indexing and querying techniques are the most crucial parts in the moving objects database systems, and privacy protection is an important and sensitive issue

that needs to be addressed in order for MOD applications to gain wide acceptance. These three issues form the focus of this thesis and their relationship is captured in Figure 1.1. Like any other applications, users and specialized devices are position providers and query issuers. For example, they could be vehicles or mobile device holders which are shown as black points in the map. The server manages the MOD and provides location-based services to users. The server has the functionality and capability like finding dense regions as shown by the rectangle in Figure 1.1, and finding k nearest neighbors for a moving object as shown by the circle. When subscribing such location-based services, users may worry about the leak of their private information. There are various models to protect privacy within the server. However, in this thesis, we propose an alternative approach to the privacy protection problem by introducing an anonymization and mapping layer between the server and the users.

The rest of the chapter is organized as follows. We first discuss the problems on indexing, querying moving objects and location privacy protection by examining existing techniques in Section 1.1. Then, in Section 1.2, we present an overview of our proposed method and state the contributions we made. Finally, in Section 1.3, we present the outline of the thesis.

1.1 Moving Objects Databases

In this section, we describe the background on moving objects databases, their characteristics, and peculiarities, and research problems.

1.1.1 Indexing Moving Objects

In the traditional spatial databases, indexes are mainly designed to speed up retrievals since objects are usually assumed to be constant unless explicitly updated. Thus, in order to capture continuously moving objects, traditional indexes have to update locations of moving objects continuously (e.g., once at each timestamp). When facing such a large amount of sampling states streaming to the database, the dominant indexing technique for static spatial data with low dimensionality – the R-tree [28] (and its descendants such as R*-tree [6])– exhibits poor update performance.

To reduce the number of updates on the indexes, strategies such as expressing the objects' positions as functions of time, and delaying of updates have been employed. As reported in [16], the use of moving functions reduces the need for updates by a factor of three for some vehicle data. However, simply applying these strategies to static databases still can not effectively reflect the dynamic nature of the moving objects. Thus, many other researchers work on developing new indexes specifically for moving objects. One representative index is the Time-Parameterized R-tree (TPR-tree) [76]. In the TPR-tree, both moving objects and their bounding rectangles are modelled as linear functions of time. The TPR-tree can then support queries on the current and anticipated near-future positions of moving objects. Similar to that in the R-tree, bounding rectangles in the TPR-tree also overlap and the overlap may become serious as time elapses. As a result, a search operation needs to travel multiple paths from the root of the index tree to leaf nodes. This problem is inherent in many multidimensional indexes. And it is exacerbated by the concurrency control algorithms, because concurrent and frequent tree ascents may lead to costly lock conflicts. Another problem with existing solutions to moving object indexing is that they cannot be easily integrated

into existing database systems due to the complexity of the algorithms.

Therefore, one objective of our study is to design a more efficient index of moving objects which can be grafted into existing database management systems cost-effectively.

1.1.2 Querying Moving Objects

Moving objects databases need to accommodate frequent updates while simultaneously allowing for efficient query processing. The developments of moving objects indexing techniques offer a foundation for the various types of query services. The most common types of queries are point queries, range queries and k-nearest neighbor queries.

- Point queries: "find the location of an object O at a given time t." For example, where is the car0001 now? The answer should return the location of the car0001.

- Range queries: "find all objects whose locations fall within a given range R from time t_1 to t_2." For example, the query could be how many cars are in the area01.

- K-nearest neighbor queries: "find the top k nearest objects of a given object O at a given time t." For example, find the k nearest taxis for a traveller.

Proposals for efficient computation of the above queries can be found in [7, 9, 41, 43, 75, 84, 94]. In this thesis, we will present that our proposed index structure can answer these common queries efficiently.

There are several more complex queries that have been studied, e.g. reverse nearest neighbor queries [7], continuous range (k-nearest neighbor queries) [54, 55, 56, 57, 89, 97] and etc.

More recently, a new type of query, density query, has been gaining interest from both industry and research communities. The objective of the density query is to find dense regions with a high concentration of moving objects. It may have applications in a range of areas. For example, in traffic management systems, density queries may be used for identifying regions with potential for congestion and traffic jams. The concept of density queries on moving objects was first introduced by Hadjieleftheriou et al. [29]. However, the definition given by them is not very practical. And they only solved a simplified version of their proposed density query. In this thesis, we will examine the density queries and present better definitions and solutions.

1.1.3 Privacy Issues

The expanding use of spatial, mobile and context-aware technologies, the deployment of integrated spatial data infrastructures and sensor-networks, and the use of location data as the foundation for many current and future information systems have profound implications on the privacy of personal information. Today people are increasingly aware of privacy issues and do not want to expose their personal information to unauthorized subjects or organizations. An important problem is represented by the possibility that a piece of personal information released by an individual to a party be combined by this party, or other parties, with other information, leading to the disclosure of sensitive personal information. In other cases, even if an individual does not directly release personal information to another party, this party may still become aware of this information if it has to provide a service to such an individual. This is in particular the case of location-based service providers that, because of the very nature of the services they provide, need to track user movements and locations. It is then easy, based on this information, to discover

user habits and other personal information. There is therefore an important concern for *location privacy* in location-based services, that is: "how can we prevent other parties from learning one's current or past location? [11]". By looking more closely at the privacy problem in such a context, we can see that there are at least two important requirements, that is, keeping movement and location information private from service providers and from other users. For example, GPS users who do not want to disclose their locations to the system may still require service such as "is there any of my friends close to me now?" There are two privacy requirements for this query. First, service providers are not allowed to know the real locations of users. Second, users can only query an authorized dataset (e.g. a list of their friends).

Some early works on location privacy protection suggest the use of policies which serve as a contractual agreement about how user's location information can be used by service providers [30, 82]. More recent works focus on the development of anonymization techniques specific to location-based service environments. A common technique is based on the notion of spatial-temporal cloaking [26]. The main drawback of these approaches is that they cannot guarantee the accuracy of the query answers. Motivated by this, we develop a novel scheme that can provide privacy protection without sacrificing the service quality.

1.2 Objectives and Contributions of This Thesis

As discussed in the previous sections, existing indexing techniques for moving objects databases still suffer from either update or query problems, and may not be able to support a new type of query, the density query, in a straightforward way. Moreover, few work has been done for the location privacy problems in the moving-

object environments. Therefore, the aim of this thesis is mainly threefold: (i) to design a new and efficient index structure; (ii) to explore the proper definition of density queries and develop a theoretical framework as well as detailed algorithms; (iii) to establish a framework for preserving location privacy in moving-object environments. Besides the theoretical studies, we also aim to build a real system based on our proposed index structure.

1.2.1 Contributions on Index Structures

We proposed a new index structure, called the B^x-tree, which was based on the most popular B^+-tree. The B^x-tree indexes the positions of moving objects as linear functions of time. By applying a novel linearization technique, the B^x-tree maps the two-dimensional positions and time attributes to one-dimensional values, which makes it possible to adopt the B^+-tree technique to manage moving objects and preserve their spatial proximity. The details of the algorithms will be covered in Chapter 3. Here, we summarize our contributions as follows.

- Our proposed B^x-tree does not exhibit the update performance problems associated with bounding-rectangle-based multidimensional indexes.
- Our proposed B^x-tree can support various types of queries and does not compromise on query and storage efficiency.
- Our proposed B^x-tree is built on the B^+-tree, which is widely used in the commercial databases. Thus, the B^x-tree can be grafted into existing database management systems more cost effectively than techniques relying on unsupported indexing techniques. Further, the B^x-tree can directly make use of the well tested and efficient Blink-tree concurrency control mechanism.

- A thorough experimental study have been carried out, which show that the B^x-tree is capable of outperforming the recent TPR*-tree [90] for both single and concurrent access scenarios.

1.2.2 Contributions on Density Queries

For the density query, we examined its earlier definition and found that the definition was impractical to some extent. Therefore, we introduced a more meaningful definition of the density query. Given the new definition, we developed a two-phase framework which built a filter on top of indexes. The details of this framework will be presented in Chapter 4. Here, we summarize our contributions as follows.

- We provide a definition of the density query for moving objects that can avoid the answer loss problem. Based on this definition, we propose a specialization of the density query that may return useful answers and is amenable to efficient computation.

- We propose algorithms to process the resulting density query efficiently. The algorithm utilizes temporal histograms of counters for each partition in a partitioning of the data space. We propose to use the Discrete Cosine transform (DCT) to compress the histograms. This compression incurs very few errors in the answer set, but offers space savings of up to 90%, which also reduces I/Os.

- We conduct extensive experiments. The results suggest that our proposed algorithm offers an improvement of a factor of 4 in terms of I/O, compared to a naive algorithm. The results also indicate that although we reduce the storage usage greatly by using the DCT, the answers are still highly accurate.

1.2.3 Contributions on Protecting Location Privacy in Moving-Object Environments

We investigate location privacy issues in moving-object environments, and propose a framework for location privacy assurance. Details of the algorithms will be presented in Chapter 5. Specifically, our contributions are the following.

- We propose a framework that can not only prevent service providers from inferring the exact locations of users, but also keep information about the location of an individual private from other individuals not authorized to access such information.
- We propose algorithms in the framework that can support continuous updates and various types of queries without degrading the service quality.
- We develop metrics to measure the level of privacy achieved by our framework. In particular, we will investigate the threats posted by the agents and the query server from discovering the users' true locations and movement pattern. We then propose intuitive methods to quantify the level of protection against these threats in our system.

1.2.4 Contributions on Extending a DBMS

It is a common knowledge that the major database market is cornered by a few vendors, and it is not an easy task to introduce a specialized DBMS supporting MOD applications. From the vendors' view points, it is too risky to touch the kernel such as implementation of a new index for every new applications as the new component is not a stand alone software, and it will affect other components such as query processors, cost model and buffer manager. In this thesis, one of our

main goals is to extend an existing DBMS such as MySQL for MOD applications. Apart from studying individual issues analytically and empirically, we incorporate our proposals into MySQL. We make the following contributions:

- We propose a client/server architecture for geo-enabled mobile service applications. The coupling between client and server is minimized to support system independence.
- We implement a moving object database system utilizing MySQL as the underlying relational engine. The boundary between our implementation and MySQL is clearly defined, which ensures the integrity of MySQL and the easy deployment or even re-porting of our proposal.
- We implement the B^x-tree into MySQL.
- We implement spatial-temporal query processing strategies, by taking full advantage of the popular database connectivity technology – JDBC.

In summary, we design and implement an extended DBMS architecture for supporting MOD applications.

1.3 Outline of The Thesis

The rest of the thesis is organized as follows:

- Chapter 2 reviews indexing and querying techniques in static spatial databases and moving object databases, and surveys state-of-the-art privacy preserving strategies.
- Chapter 3 presents our proposed index structure for moving objects, called the B^x-tree. This novel index structure enables the B^+-tree to manage moving objects as well as various types of queries.

- Chapter 4 presents a new definition of the density query and corresponding solutions. We propose a general framework based on which we solve the density query in an efficient way.

- Chapter 5 presents an approach to ensure location privacy in moving-object environments. We interpose agents between users and servers and use multiple successive transformations on data to keep the server from inferring the real positional information.

- Chapter 6 presents an operable database system which is built on top of a popular relational database system MySQL. In this system, we implement the B^x-tree non-intrusively into the MySQL core.

- Chapter 7 concludes our work and discusses directions for future work.

Two papers have been published from the work reported in this thesis. The main idea of indexing moving objects, presented in Chapter 3, has been published in [36]. The work on querying moving objects, presented in Chapter 4, has been published in [37].

CHAPTER 2

Literature Review

In this chapter, we first briefly review the traditional indexes in spatial databases. Then we investigate existing indexing and querying techniques for moving objects databases. Finally, we discuss some related work in location privacy issues.

2.1 Traditional Indexes in Spatial Databases

Most indexes of moving objects are based on some famous traditional indexes [10, 23, 79, 80, 100], especially the R-tree (and its variants), thus, we will first make a brief review of these indexes to obtain a better understanding of later works.

The R-tree [28] (see Figure 2.1) is a hierarchical, height-balanced index structure. Objects are represented by minimum bounding rectangles (MBRs). Each leaf node of the R-tree points to the MBRs of objects and each internal node points to other internal nodes or leaf nodes. Due to possible overlaps of the MBRs, the search to find out rectangles intersecting a given range has to descend all subtrees

that intersect or fully contain the range specification. To insert an object, they traverse a single path from the root to the leaf. At each level they choose the child node whose corresponding MBR needs the least enlargement to enclose the MBR of the new object. If there is not enough space left in the leaf node, the node should be split and its ancestor nodes should be adjusted accordingly. As for deletion, they first perform an exact match query for the object in question. If it is found in the tree, it will be deleted. If the deletion does not cause an underflow, they check whether the MBR could be reduced and propagate this adjustment upwards. If an underflow occurs, they remove all entries in this leaf node and then reinsert them.

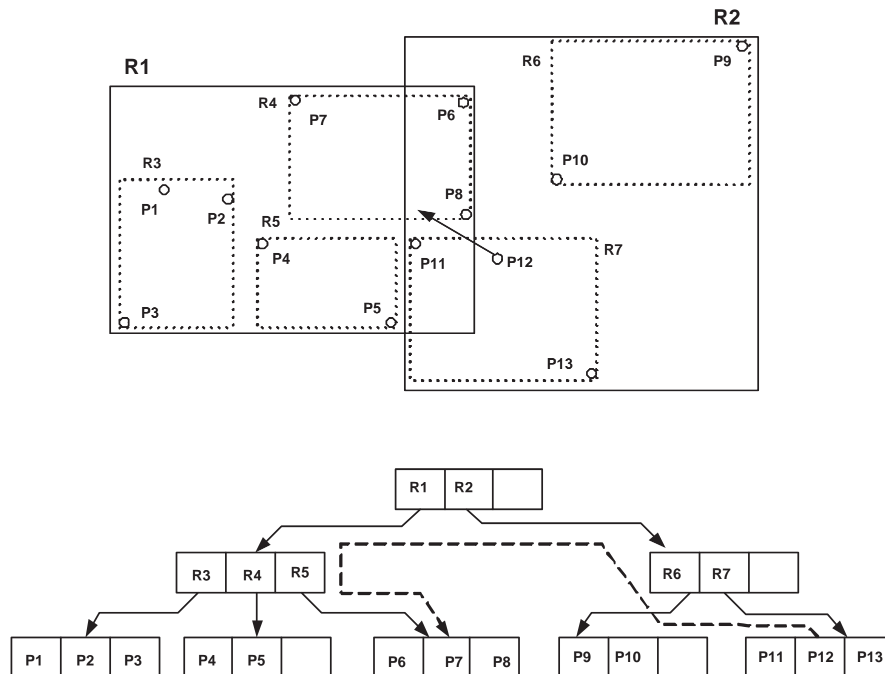


Figure 2.1: An Example of the R-tree Structure

Based on a careful study of the R-tree, Beckmann et al. [6] identify several weaknesses of the node splitting algorithms used by the R-tree, and then propose the R*-tree. Beckmann et al. also confirm the observation of Roussopoulos et al. that the insertion phase is critical for good search performance. The design of the R*-tree therefore introduces a policy called forced reinsert: if a node overflows, it

is not split at once. Rather, part of the entries of the node are removed and then reinserted into the tree. The R*-tree has been proved to be the most successful variant of the R-tree. Beckmann et al. report performance improvements of up to 50% compared to the R-tree.

If we simply apply the R-tree like technique to index the locations of moving objects, readjusting the entire index is inevitable. For example, the short movement of p_{12} in Figure 2.1 will cause the nodes and MBRs along two paths to be adjusted. Such adjustments are expensive when large numbers of updates are continuously issued. Hence, the original R-tree technique is not directly suitable for moving objects. However, due to its robustness in handling spatial objects, the R-tree and its variants are still good basis for extension for supporting moving objects.

Another often used index structure is the quadtree. Samet [77] has done a thorough survey of the quadtree and the related hierarchical data structures. The basic idea of the quadtree is to recursively decompose the space. Variants of quadtrees can be differentiated on the following two bases: (i) the type of data that they are used to represent; (ii) the principle guiding the decomposition process; and (iii) the resolution (variable or not). Currently, the quadtrees are used for point data [8, 21], regions [42], curves [4, 31, 86] and volumes [35, 50]. The decomposition may be into equal parts on each level, or be governed by the input. The resolution of the decomposition (i.e. the number of times that the decomposition process is applied) may be fixed beforehand, or be governed by properties of the input data. Figure 2.2 shows an example of common quadtree structure, where the shaded regions denote the places containing data. The root node corresponds to the entire space, and each son of a node represents a quadrant of the region of this node. We can see that the quadtree is not a balanced tree.

Directly adopting the quadtree technique in the moving object database en-

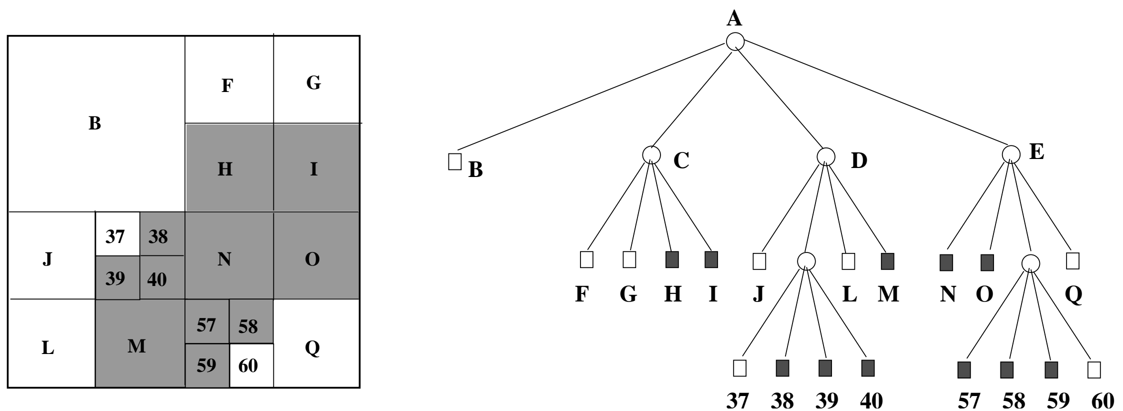


Figure 2.2: An Example of the Quadtree Structure

counters similar problems as existing in the R-tree like structures. In the following section, we will see what kind of modifications are needed to make the quadtree suitable for moving object indexing.

2.2 Moving Objects Indexes

Traditional indexes for multidimensional databases, such as the R-tree and its variants were, implicitly or explicitly, designed with the main objective of supporting efficient query processing as opposed to enabling efficient updates. This works well in applications where queries are relatively much more frequent than updates. However, applications involving the indexing of moving objects exhibit workloads characterized by heavy loads of updates in addition to frequent queries.

Several new index structures have been proposed for the moving-object indexing, and recent surveys exist that cover different aspects of these [2, 53, 61]. One may distinguish between indexing of the past positions and indexing of the current and near-future positions of spatial objects. Our work belongs to the latter one.

2.2.1 Indexing historical movement

Historical data of moving objects is very useful in applications such as the road planning and resource management. However, in such a database, the volume would be very large since objects move all the time and the database has to capture a great deal of the location information. Hence, the critical problem is to decide what is good historical data and how to store them efficiently.

One of the earliest work is the Historical R-tree (HR-tree) [59] which constructs an R-tree for each timestamp in history. Consecutive R-trees can make use of common paths if objects do not change their positions, and new branches are created only for objects that have moved. HR-trees are very efficient for timestamp queries, as search degenerates into a static query for which R-trees are very efficient. Their disadvantage is the extensive duplication of objects that leads to huge space consumption. As a side effect of this fact, their performance on interval queries is very poor. Aimed at achieving good performance on both timestamp and interval queries, Tao and Papadias propose the Multi-version 3D R-tree (MV3R-tree) [88] which combines Multi-version B-trees [5] and 3D R-trees [93]. The MV3R-tree involves numerous improvements that result in large space savings without compromising timestamp query performance compared to the HR-tree. Furthermore, the MV3R-tree includes a small auxiliary 3D R-tree on the leaf nodes (not on the actual objects). As reported by the authors, the MV3R-tree usually outperforms the traditional 3D R-tree on interval queries and its performance does not deteriorate significantly when time evolves.

Another direction of indexing historical information of moving objects is to represent the historical movement of objects by their trajectories, i.e., a set of line segments. Intuitively, an R-tree can be used to index the trajectories of objects by bounding the line segments with MBRs. Based on this idea, Pfoser et al. propose

the Spatio-Temporal R-tree (STR-tree) and Trajectory-Bundle tree (TB-tree) [69]. The STR-tree organizes line segments not only according to spatial properties, but also by attempting to group the segments according to the trajectories they belong to. The TB-tree aims only for trajectory preservation and leaves other spatial properties aside, while it performs better than the STR-tree. The main problem of such index structures is the dead space in each MBR which may degrade both update and query efficiency.

Recently, Frentzos [22] proposes the Fixed Network R-tree (FNR-tree) by taking into account the constraint of road networks. The general idea that describes the FNR-tree is a forest of 1-dimensional (1D) R-trees on top of a 2-dimensional (2D) R-tree. The 2D R-tree is used to index the spatial data of the network (e.g. roads consisting of line segments), while the 1D R-trees are used to index the time interval of each object's movement inside a given link of the network. However, the dead space problem still exists in the 2D R-trees used by the FNR-tree, and the 1D R-tree may not be efficient to index objects when the road is long and objects density is high.

2.2.2 Indexing current and future movement

More recent works focus on indexing current and future movement. This category can be further divided into two sub-categories: indexing locations of moving objects and using functions to approximate movement.

Indexing locations of moving objects

One of the differences between moving objects and static objects is that the locations of moving objects vary over time. In order to represent moving objects in the database, it is inevitable to employ a large volume of updates.

To overcome this problem, Song et al. [83] introduce a hashing technique which uses buckets to hold moving objects. They save the bucket information for each object instead of the object's exact location so that update is triggered only when the object leaves its original position very far (i.e. moves out of its current bucket). Although this method reduces update frequency and speeds up the update process, it suffers from the accuracy problem when answering queries. For example, when the query range intersects with the bucket, the system can not distinguish which objects in the bucket are in the range and which are not.

Similar to Song et al.'s idea, Kwon et al. [45] propose the Lazy Update R-tree (LUR-tree), in which they suggest to ignore deletions of objects that do not move out from the current MBR, or enlarge MBR slightly if objects do not move far away from it. However, this algorithm also downgrades query performance since the ignorance of deletions makes the index loose chances of obtaining a better structure, and the enlarged MBRs may overlap more severely, both of which cause subtrees to be traversed unnecessarily.

Later, Xia et al. [96] propose the Q+R-tree. The Q+R-tree makes use of the topography and the patterns of object movement. It distinguishes fast-moving objects from quasi-static objects, and stores these two types of objects in a Quad-tree and an R*-tree respectively. Objects may switch between two trees when they change their moving status. The Q+R-tree performs well only when there are very few fast-moving objects.

Another work that aims to speed up the update processing is proposed by Lee et al. [46]. They observe that the traditional R-tree update strategy is a top-down search which is inherently inefficient because the objects are stored in the leaf nodes, whereas the starting point for an update is the root. Therefore, they propose a bottom-up update strategy. The main idea is to execute the update

from the bottom of the tree. They first consider enlarging the leaf MBR or placing the new object location in another sibling leaf node if the object moves outside its current leaf MBR. If this strategy does not work, they then ascend the index with an auxiliary data structure, Direct Access Table (DAT), which is a summary of the R-tree and provides direct access to the index nodes. This algorithm encounters the similar problem of the LUR-tree, where the overlaps among enlarged MBRs result in more unnecessary traversals in the tree.

Indexing based on time functions

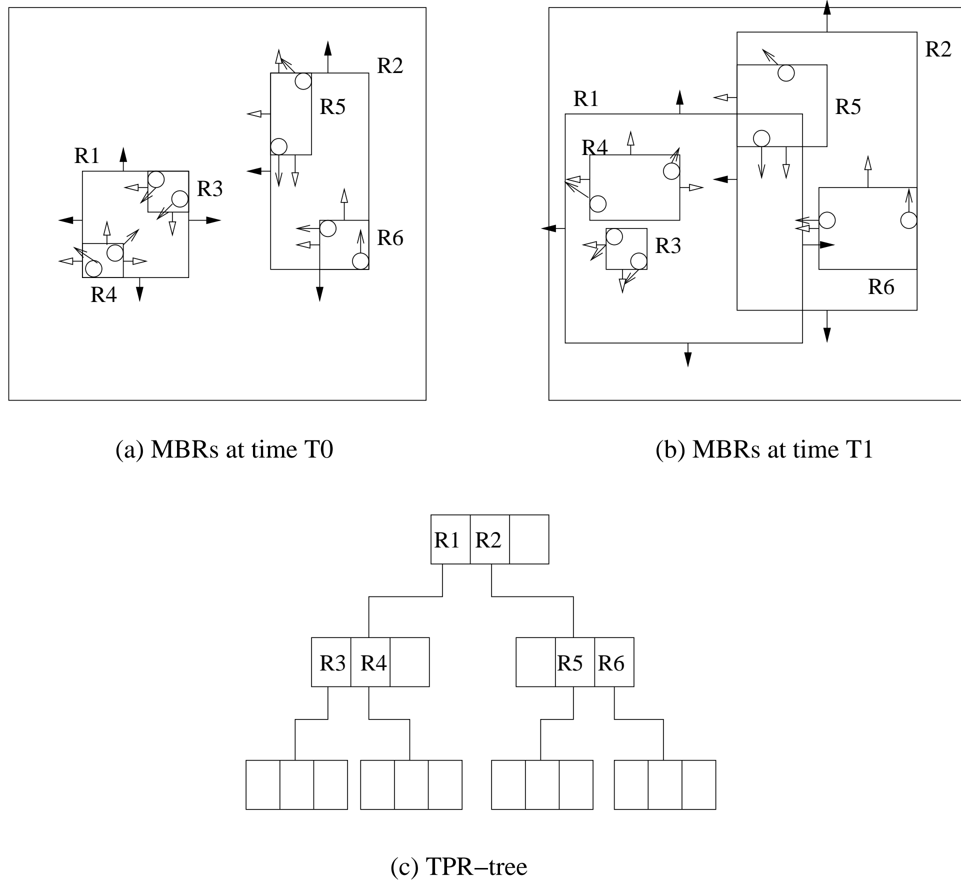
A crucial issue in the approach of indexing locations of moving objects is to maintain up-to-date information about the locations of moving objects. For a large amount of objects, too many database update operations may be triggered after each state sampling. For example, there could be thousands of cars on a small segment of a highway at any given time of a day. And of course they are moving continuously unless there is an accident or a heavy traffic congestion on their paths. Updating their current locations once a second cause thousands of transactions per second, not to mention the query transactions which could cripple the server at the control center. Consequently, keeping track of each and every car's current location in real time is very hard to achieve and even impossible. Thus, instead of updating continuously, Sistla et al. [81] present a new model that uses a linear function with the parameters of the position and velocity vector (so-called dynamic attributes) to represent the current and near-future locations of moving objects. These parameters need to be updated only when the moving objects change their speeds or directions significantly. As reported in [16], the use of such moving functions reduces the need for updates by a factor of three for some vehicle data.

Largely based on the idea introduced by Sistla el al., the quad-tree or the

R-tree based index structures for moving objects have been proposed. Tayeb et al. [92] employ the PMR quadtree to index the future linear trajectories of one-dimensional moving points as line segments in (x, t)-space. The segments span the time interval that starts at the current time and extends some time into the future, after which time, a new tree must be built. Next, Kollis et al. [43] employ dual transformation techniques which represent the position of an object moving in d-dimensional space as a point in $2d$-dimensional space. Agarwal et al. [1] extend the transformation to arbitrary dimensionality, and propose theoretical indexes that achieve good asymptotic performance. These solutions, however, are not efficient in practice due to the large hidden constants in their complexities. By using the similar technique, Patel et al. [67] have developed a practical indexing method, called STRIPES, which maps 2D moving objects to 4D points and then index them by the PR bucket quadtree [78]. STRIPES supports efficient updates and queries but requires large storage space.

Similar to the quadtree-based techniques, Chon et al. [15] model the space-time domain space as a grid and the trajectory of a moving object as a polyline in the grid. The advantage of using the grid is the great speedup of the query processing. However, this algorithm incurs high overhead of updates since it requires duplicating an object across all cells. Furthermore, different to previous models, the trajectories of moving objects in this model are affected by other moving objects. This means one insertion may cause a series of updates since it not only needs to change trajectories of the newly inserted object, but may also need to change the trajectories of other objects who have been influenced by the current object.

Index structures based on the R-tree are the Time-Parameterized R-tree (TPR-tree) and its variants. Saltenis et al. [76] propose the TPR-tree which augments



(a) MBRs at time T0

(b) MBRs at time T1

(c) TPR–tree

Figure 2.3: An Example of the TPR-tree

the R-tree with velocities to index moving objects. Figure 2.3 shows an example. The white circles represent the positions of moving objects, and the arrows indicate their movements. Moving objects are stored as a reference position and a corresponding velocity vector. The coordinates of the bounding rectangles are also functions of time. As shown in the figure, in each dimension, the lower bound of a MBR is set to move with the minimum speed of the enclosed points, while the upper bound is set to move with the maximum speed of the enclosed points. This ensures that the bounding rectangles are indeed bounding for all times considered. As time elapses, the grown MBRs will overlap more severely (see Figure 2.3(b)) and adversely affect query performance. Therefore, frequent updates are needed

to ensure that moving objects that are currently close are assigned to the same bounding rectangles. Further, bounding rectangles never shrink and are generally larger than strictly needed. To counter this phenomenon, the so-called "tightening" is applied to bounding rectangles when they are accessed.

Next, two notable proposals exist that build on the ideas of the TPR-tree. Procopiuc et al. [70] propose the STAR-tree. This index seems to be best suited for workloads with infrequent updates. Tao et al. [90] propose the TPR*-tree. They adopt assumptions about the query workload and improve the construction algorithms by carefully choosing the insertion path for each moving object. Their approach only alleviates the mentioned MBR overlapping problem but still cannot fully solve it. As reported, the TPR*-tree achieves better and stable I/O performance compared with the TPR-tree. However, the insertion and deletion algorithms of the TPR*-tree are much more complicated which may impede its integration to existing database systems. Further, the performance of the TPR*-tree is tested by setting the page size to 1K bytes which is not very appropriate since the typical page size is 4K bytes, and in modern hardware, most OS read in 8K bytes. The smaller page size allows better optimization and may inflate the performance gain somewhat.

More recently, Cui et al. [18] found that, to better manage moving object databases, there is a need to improve the utilization of the main memory. Since main memory is much faster than disk, efficient management of moving-object database can be achieved through aggressive use of main memory. They propose an Integrated Memory Partitioning and Activity Conscious Twin-index (IMPACT) framework where the moving objects database is indexed by a pair of indexes based on the properties of the objects' movement, where a main-memory structure manages active objects while a disk-based index handles inactive objects. This

framework can be applied to most existing indexing techniques and it achieves better performance when the migration of objects between the disk and the memory is not very frequent.

We would also like to mention that besides the linear moving function model, which is used in most work, a recent proposal considers non-linear object movement [87]. The idea is to derive a recursive motion function that predicts the future positions of a moving object based on the positions in the recent past. However, this approach is much more complex than the widely adopted linear model and complicates the analysis of several interesting spatio-temporal problems. Thus, we decide to use the linear model in our work.

Different from previous works, our idea is to enable the B$^+$-tree to index moving objects and we propose the B^x-tree [36]. We map the object locations to certain timestamps and then convert them from 2-dimensional space to 1-dimensional space by employing space-filling curves. Most recently, Yiu et al. [98] suggest that capturing velocity information and using a higher dimensional Hilbert curve may achieve better performance. Correspondingly, they propose the B^{dual}-tree. The B^{dual}-tree is composed of two B$^+$-trees and these two trees swap states every maximum update interval. Objects in the B^{dual}-tree are indexed according to the keys obtained from mapping both location and velocity data to one-dimensional space by a 4-dimensional Hilbert curve. During the query, they decompose the Hilbert interval of each node into squares with continuous Hilbert values. These squares can be treated as MBRs as that in the TPR-tree and hence the query algorithms of the TPR-tree can be applied to the B^{dual}-tree by minor modifications. Their experimental results show that the B^{dual}-tree is as efficient as the B^x-tree regarding the update performance, while it performs similarly to the TPR*-tree and outperforms the B^x-tree and STRIPES (a recent index based on dual transformations)

regarding the query performance. Note that the experiments are carried out by assuming the disk page size to be 1K bytes which is smaller than the standard setting (4K bytes). As we know that, an index is chosen not based on certain page size it is good at, but rather, is chosen based on its performance on the standard page size. Another disadvantage of the B^{dual}-tree is that its query algorithm is more complicated and not based on that of the B^+-tree, which may make it hard to be integrated into existing database systems.

In this thesis, we will present an optimized version of the B^x-tree. We will compare the B^x-tree with both the TPR*-tree and the B^{dual}-tree. Our experimental results will show that the optimized version of the B^x-tree is superior to existing indexes with respect to both update and query performance.

2.3 Queries on Moving Objects

In this section, we first review the common definitions and solutions of range and k-nearest neighbor queries which are supported by our proposed index structure. Then we briefly introduce some variants of range and k-nearest neighbor queries which are proposed under certain constraints, e.g., network constraints. Finally, we present the related work to the newly emerging query – the density query.

2.3.1 Range Query

The range query is one of the most common queries in spatial-temporal databases, which retrieves all objects whose location falls within the rectangular range at a timestamp or during a time interval. According to the query timestamps, range queries can be further divided into predictive range queries or historical range queries. In our work, we focus on predictive queries.

For the range query, the search process in an R-tree like index structure is very different from that in a B-tree due to the lack of ordering and the possible overlap among keys. To find all bounding rectangles intersecting a given range, the search process will descend all subtrees that intersect or fully contain the range specification.

Besides the unconstrained movement that is the scenario mostly asserted in traditional spatiotemporal access methods, some recent works suggest to take into account the infrastructure constraint. Objects that constrain movement are termed infrastructure. For example, pedestrians may be blocked by infrastructures such as buildings, lakes etc; vessels may be blocked by infrastructures such as rocks, islands etc. Under this condition, Pfoser et al. [68] propose to decompose a given query window based on the infrastructure contained in it and then queries the resulting segmentations not occupied by infrastructure to save cost. Figure 2.4 illustrates a query example. The biggest rectangle is a given range query, the black blocks denote infrastructure inside this range query and the white rectangles are the decomposed sub-queries.

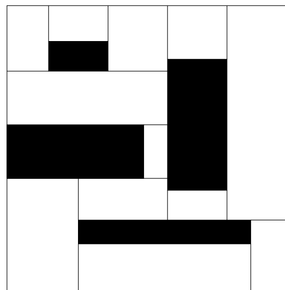


Figure 2.4: An Example of the Constrained Range Query

2.3.2 K-nearest Neighbor Query

The k-nearest neighbor (kNN) query retrieves k objects for which no other objects are nearer to the query object at the query timestamp. The kNN query is a

little more complicated than the range query. A number of methods have been proposed for efficient processing of nearest neighbor queries for stationary points. The majority of the methods use branch-and-bound strategies and index structures. Generally speaking, algorithms of nearest neighbor queries can be easily extended to k-nearest neighbor queries. One outlier example is a method based on Voronoi cells [74] proposed by Berchtold et al. [9].

The original and most influential search algorithm is proposed by Roussopoulos et al. [75], which is designed for the R-tree but can also be used for querying moving objects. It is a depth-first method in a branch-and-bound manner. Specifically, starting from the root, all entries are sorted according to their minimum distance (*mindist*) from the query point, and the entry with the smallest value is visited first. The process is repeated recursively until the leaf level where the first potential nearest neighbor is found. During backtracking to the upper levels, the algorithm uses two metrics: *mindist* and *minmaxdist* for ordering and pruning the search tree. In the example of Figure 2.5, the aim is to find the nearest neighbor of the given query point, the algorithm first accesses R_1 since it has the minimum mindist, and

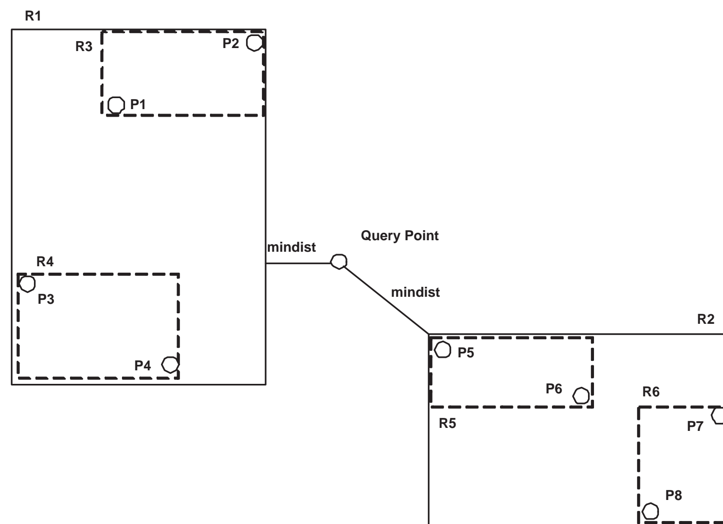


Figure 2.5: An Example of Nearest Neighbor Search

then R_4, where the first candidate object P_4 is obtained. When backtracking to the previous level, R_3 is excluded since its mindist is greater than (or equal to) the distance of P_4. Then R_2 and R_5 are visited, where the actual nearest neighbor P_5 is found. Cheung and Fu [14] prove that, given the *mindist*-based ordering of the tree traversal, the pruning obtained by Roussopoulos et al. can be achieved without use of *minmaxdist*. Some other branch-and-bound methods modify the index structures to better support the kNN query [41, 94], e.g. using bounding circles instead of bounding rectangles.

The most recent work on the kNN queries for moving objects is the work of Benetis et al. [7]. They use the TPR-tree as the underlying index structure and propose a branch-and-bound algorithm similar to that in [75], but use a new metric. This algorithm can return the nearest neighbors for each time point during the query time interval. And the idea behind the metric is to first visit parts of the tree that are on average the closest to the query point. Then the rectangle is pruned if there is no chance that it will contain a point that at some time during the query interval is closer to the query point than the currently known closest point to query point at that time.

Besides what we have discussed above, some interesting variants of kNN queries exist. Song and Roussopoulos [84] introduce a query that is to find k nearest neighbors in a static database for a moving query point. They propose to use prefetch and buffer stategies for such kNN queries. Specifically, they make use of previous result set to bound the current search to a smaller area, or reuse the previous results if the query point only moves a little. They build two buffers. One is for current result and the other one for the predicted next query result which is obtained by prefetching more neighbors of the current query. However, these strategies may not take effect when the query point moves fast.

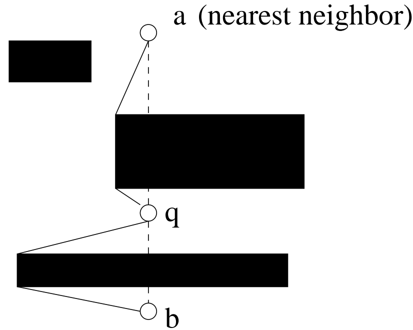


Figure 2.6: An Example of the Constrained kNN Query

Zhang et al. [101] study the problem of kNN queries under the infrastructure constraints as introduced in Section 2.3.1. As shown in Figure 2.6, the nearest neighbor (object b) is not simply the one (object a) with the shortest Euclidean distance to the query point (object q). Thus, the authors propose to maintain partial visibility graphs for infrastructures to facilitate the search.

Papadias et al. [66] propose an architecture that can answer general queries (e.g. range query, kNN query) under the constraints of spatial networks. Two strategies have been developed. One is called *Euclidean restriction*. They first find the object with the shortest Euclidean distance, and then enlarge the search region with the network distance between this object and the query point. The other method is called *network expansion*, which starts the searching from the road segment covering the query point and then incrementally expands the search region according to the node connectivity in the network.

Another variant of the kNN query is the group-nearest neighbor query (GNN) introduced by Papadias et al. [63]. A GNN query retrieves all the objects with the smallest sum of distances to all query objects. They propose several algorithms which are mostly based on the idea of using *mindist*.

2.3.3 Density Query

Density queries are aimed at locating regions with a density higher than a given threshold at the query timestamp.

As will become clear in Chapter 4, any index for moving objects that supports predictive range queries is able to answer density queries by using our framework as presented. In this study, we employed our proposed B^x-tree as the underlying index structure.

Several proposals [62, 64, 65] exist for the computation of spatio-temporal aggregation queries, which are similar to density queries in some sense, since density queries also need to know the numbers of objects inside certain ranges. However, a key difference is that the query ranges are given for aggregation queries, while density queries must locate ranges that satisfy the density threshold.

Existing clustering algorithms [3, 34, 40, 60] can represent the most dense areas by the centers of the clusters. As good examples, Yiu and Mamoulis [99] cluster objects at a certain timestamp; and Li et al. [49] cluster moving objects but at the expense of high maintenance costs. These techniques do not meet the requirements posed by density queries. They are unable to identify the regions which are not given by the cluster centers but have densities higher than the specified threshold. And they are not effective in tracking the continuously changing positions of moving objects.

The most closely related work is by Hadjieleftheriou et al. [29] who first introduce the concept of the density query. They define the period density query as to locate arbitrary dense regions in a time interval. They then found that finding all the arbitrary regions that satisfy the density requirements was considerably difficult, and hence they turned to a simplified density query. Specifically, they partitioned data space into disjoint cells, and reported dense cells, instead of arbitrary

regions, that satisfy the query conditions. There are at least two disadvantages of their approach. First, for the applications we envision, finding dense regions during a time interval appears to be less useful than finding dense regions in a single timestamp. For example, once a traffic jam occurs in some region (i.e. dense region), vehicles (i.e. moving objects) around this region may slow down, and their velocities may change dramatically. Predicting dense regions in the following timestamps according to the original velocities reported for the moving objects may be of less value. Second, their schema suffers from what we termed *answer loss* problem. Consider an example of a square consisting of four cells, where each cell contains one object. Given the density threshold 3, there may be a dense region in the center of the square, but the simplified query algorithm will always report no dense region. Therefore, one purpose of our study was to present a more pragmatic definition of the density query.

2.4 Concurrency in Indexes

Nowadays many database applications run in multiuser environments. Incorrect results may be returned if index structures do not have proper concurrency control. This section introduces concurrency schemas for the two popular indexes, the B^+-tree and the R-tree. Other index structures based on these two indexes can also adopt the similar concurrency algorithms.

To provide efficient concurrent traversal and update of the B^+-tree, Lehman et al. [47] propose the B-link tree. The structure of B^+-tree is slightly modified where every node keeps a right link pointing to the right sibling node in the same level. All nodes in a right-link are ordered by their highest keys. When a search process goes down in the tree, it will learn of any splits racing with it by comparing the

highest keys. If the highest key is lower than the expected key, conclusion can be made that a split must have taken place. This guarantees that insertion can be performed without blocking search processes.

For the R-tree, Kornacker et al. [44] propose the R-link tree which employs a similar modification as that in the B-link tree. The main difference between the R-tree and the B^+-tree is that keys in the R-tree are not ordered. Therefore, a unique logical sequence number (LSN) is introduced for each node and kept in each entry of the internal nodes. Comparison of these LSNs is used to discover node splits. A right link chain is again used to locate newly split nodes.

If a node is split, the new split node is inserted into the right link chain and it holds the old node's LSN. The original node is assigned a new LSN which is higher than the old one. Before the new node is installed, the expected LSN in the corresponding entry of the parent node is not updated. The split of a node can be detected by comparing the expected LSN taken from the entry in the parent node with the actual LSN in this node. If the latter is higher than the former, there is an uninstalled split. Travelling along the right link chain, therefore, is necessary. The traversal is terminated when a node with LSN equal to the expected LSN is encountered. Another difference is that if the bounding rectangle in the leaf node is changed, we must propagate the change to its ancestor nodes. This process employs top-down lock-coupling.

The locking strategy of the B-link tree and the R-link tree is deadlock-free since there is always only one lock in the B-link tree, and the R-link tree employs lock-coupling only in the top-down process.

2.5 Approaches for Location Privacy Protection

Privacy issues in location-aware mobile devices [51] have recently attracted considerable research interest. Some early works on location privacy protection suggest the use of policies, which serve as a contractual agreement about how user's location information can be used by service providers [30, 82]. Typically, users have to trust the service providers. However, such a trusted relationship is hard and costly to establish especially for small or temporary service providers.

Therefore, more recent works focus on the development of anonymization techniques specific to location-based service environments. A common technique is based on the notion of spatial-temporal cloaking. The idea is firstly introduced by Gruteser et al. [26]. They propose the application of the k-anonymity technique to cloak location information in order to support anonymous applications. Specifically, a user's location is represented by a region in which other $k-1$ users are also present. The disadvantage of this model is that it is not suitable for non-uniform distribution. This model has later been improved by Gedik et al. [24]. Their approach supports the assignment of different values for different users to the k parameter in a system. However, they did not consider query execution. In [11], Beresford et al. use the k-anonymity metric in pseudonymous applications. The idea is to rename user's identity when there are at least k users in the same zone. When there are less than k users in the same zone, a user may refuse to disclose his location. Recently, Cheng et al. [13] study the trade-off of location cloaking, privacy and quality of service. They developed queries that evaluate cloaked data and provide probabilistic answers. They also presented quality metrics in order to quantify the effect of cloaking on service quality. Based on the similar idea, Mokbel et al.[52] propose a framework to protect mobile users in location-based services, which adopts the cloaking idea and supports various k parameters. Another recent

approach *hilbASR* is proposed by Kalnis et al. [39], who use Hilbert space filling curve [58] to group users into buckets of k. The idea of *hilbASR* is later applied to a distributed mobile system called PRIVE [25]. The PRIVE relieves the risk of the collapsing of the single centralized agent, by adopting P2P techniques. However, it could be hard to convince users to use such a system because users need to take too many responsibilities.

The above k-anonymity model based approaches have at least one of the following drawbacks. First, some approaches cannot guarantee the accuracy of the query answers. Second, some approaches cannot be applied when there are less than k users in a specific area. Third, they trust agents and allow agents to store information about users, which may make agents the target of attacks by malicious parties. Finally, such a k-anonymity model may not be able to support anonymizations around sensitive areas such as home addresses in non-anonymous applications. For example, if a user's ID is known, the cloaking region around his home address will tell attackers that the user is probably at his home.

For non-anonymous applications, Gruteser et al. [27] propose to partition each region, covered by the system, into sensitive and insensitive zones. Based on such a distinction, they propose disclosure-control algorithms that only release users' positions when they are in insensitive areas and hide users' location information when they enter sensitive areas. However, service quality is not investigated as part of their work. Service quality however may drastically decrease in such an approach due to delays and omission of location information.

A different approach has been proposed by Hore et al. [32], based on encryption. In particular, they suggest encrypting location data and using a privacy-preserving index for executing range queries over encrypted data. However, this technique only works for specific query operators and is unable to provide accurate query

answers.

Unlike existing approaches to the problem of location privacy protection, our approach can be applied to anonymous, pseudonymous and non-anonymous applications, and guarantees 100% correct query answers without information leaking.

2.6 Summary

In this chapter, we first introduced some traditional sptatio-temporal indexes. Then we examined existing indexing techqniues for moving objects. We have seen that most of them are developed from the traditional indexes. Next, we discussed two common types of queries, range queries and kNN queries, as well as some of their variants. We also explored a newly emerging type of query, the density query. After that, we introduce two basic concurrency control algorithms for index structures. Finally, we reviewed some techniques related to privacy protection in the moving-object environments.

CHAPTER 3

The B^x-tree: Query and Update Efficient B^+-tree Based Indexing of Moving Objects

An infrastructure is emerging that enables data management applications that rely on the tracking of the locations of moving objects such as vehicles, users of wireless devices, and goods. Further, a wide range of other applications, beyond those to do with moving objects, rely on the sampling of continuous, multidimensional variables. The provisioning of high performance and scalable data management support for such applications presents new challenges. One key challenge derives from the need to accommodate frequent updates while simultaneously allowing for efficient query processing [38, 61]. In this chapter, we address this challenge and present a new index structure, called the B^x-tree.

The rest of the chapter is organized as follows. Section 3.1 gives a synop-

sis of our proposed solution. Section 3.2 describes the structure of the B^x-tree, and it presents the associated query and update operations. Section 3.3 covers comprehensive performance experiments. Finally, Section 3.4 concludes.

3.1 Synopsis of Our Proposal

We propose a novel way of indexing moving objects using the classical B$^+$-tree without compromising on query and storage efficiency. The motivation for using the B$^+$-tree is threefold. First, the B$^+$-tree is used widely in commercial database systems and has proven to be very efficient with respect to queries as well as updates, robust with respect to varying workloads, and scalable. Second, being a one-dimensional index, it does not exhibit the update performance problems associated with MBR-based multi-dimensional indexes. Third, it is typically appropriate to model moving-object extents as points. This enables linearization and subsequent B$^+$-tree indexing.

To use the B$^+$-tree, we must be able to linearize the representation of the locations of the moving objects. This is done by means of a space-filling curve, which enumerates every point in the discrete, multi-dimensional space. Attractive space-filling curves such as the Peano curve (or Z-curve) and the Hilbert curve, which we use in this study, preserve proximity, meaning that points close in the multidimensional space tend to be close in the one-dimensional space obtained by the curve [58].

A B$^+$-tree with the above space-filling curves works very well for static databases. A naive way to accommodate moving points is to update each object in the database at each timestamp. To avoid an excessive update overhead, we propose a novel indexing method, termed the B^x-tree, where "x" indicates the flexibility of the

proposed method in employing a specific ("x") space-filling curve as part of the linearization function.

First, we model moving objects as linear functions of time. Thus, the data to be indexed in the B^x-tree are not points (constant functions), but linear functions coupled with the times they were updated. Intuitively, an update occurs when the position predicted by an existing function is deemed inaccurate [16]. A recent study of GPS logs obtained from two dozen cars traveling in a semi-urban environment measures the number of updates needed to ensure that the values recorded in the database do not differ by more than some threshold from the real values. For realistic thresholds, the use of linear functions reduces the amount of updates to one third in comparison to constant functions [16]. Second, we effectively "partition" the index, placing entries in partitions based on their update time. More specifically, we first partition the time axis into intervals where the duration of an interval is an approximation of the maximum duration in-between two updates of any object location. We then partition each such interval into n equal-length sub-intervals, termed *phases*. Each phase is assigned the time point it ends as a *label timestamp*, and a label timestamp is mapped to a partition. An update is placed in the partition given by the label timestamp of the phase during which it occurs. For an object, the value indexed by the B^x-tree is the concatenation of its partition number and the result of applying the underlying space-filling method to the position of the object as of the label timestamp of its phase.

This mapping scheme overcomes the limitation of the B$^+$-tree, which is able to only keep the snapshot of all the objects at the same time point. This scheme reduces the update frequency, it preserves spatial proximity within each partition, and it facilitates queries on anticipated near-future positions.

Based on the above, we propose efficient algorithms for range and k nearest

neighbor queries. The algorithms are general and can be applied to indexes that use sampling techniques to model moving objects. Like any new indexing method built on top of the B^+-tree, the B^x-tree can be grafted into existing database systems cost effectively.

We report on an extensive experimental study, which includes a comparison with the TPR*-tree and the B^{dual}-tree. The results show that the B^x-tree outperforms the TPR*-tree and the B^{dual}-tree with respect to storage space, range and k nearest neighbor queries in both single and concurrent access environments. Throughout the chapter, we identify the main optimizations over the basic version of the B^x-tree.

3.2 Structure and Algorithms

We first describe the structure of the B^x-tree. Then we present the algorithms for range queries and kNN queries in detail. Finally, we cover the insertion, deletion, and so-called migration.

3.2.1 Index Structure

The base structure of the B^x-tree is that of the B^+-tree. Thus, internal nodes serve as a directory. In order to support B-link concurrency control [85], each internal node contains a pointer to its right sibling (the pointer is non-null if one exists). Each leaf-node entry represents a moving object and contains the id, velocity, single-dimensional mapping value and the latest update time of the object. Different from the earlier version of the B^x-tree [36], we optimize the fanout of the leaf nodes by not storing also the locations of moving objects, as these can be derived from the mapping values. We proceed to describe how object locations are mapped to

single-dimensional values.

Specifically, we use a space-filling curve for this purpose. Such a curve is a continuous path that visits every point in the discrete, multidimensional space exactly once and never crosses itself.

We consider versions of the B^x-tree that use the Peano curve (or Z-curve) and the Hilbert curve (see Figure 3.1). Although other curves may be used, these two are expected to be particularly good. Analytical and empirical studies [20, 58] show that for the two-dimensional space we consider, these curves are effective in preserving proximity, meaning that points close in the multidimensional space tend to be close in the one-dimensional space after the mapping. The Hilbert curve is expected to be (slightly) better than the Peano curve [20].

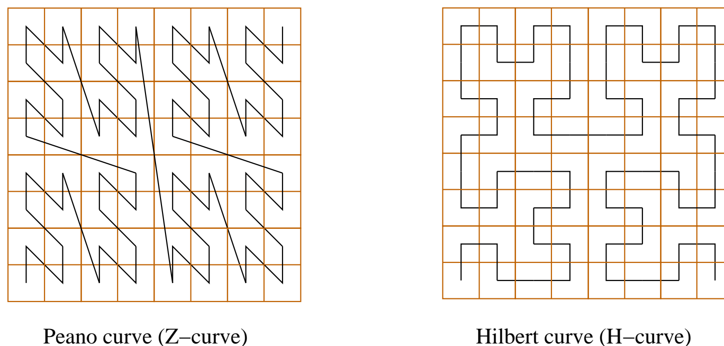


Figure 3.1: Space-Filling Curves

In what follows, we term the value obtained from the space-filling curve the *x_value*; and for brevity, we use simply the Peano curve in most discussions.

To reduce the update workload, we model point values as linear functions of time, rather than simply as static points, i.e., constant functions. An object location is thus given by $O = (\overrightarrow{x}, \overrightarrow{v})$, a position and a velocity, and an update time t_u, where these values are valid.

In a leaf-node entry, an object O updated at t_u is represented by a value

$B^x value(O, t_u)$:

$$B^x value(O, t_u) = [index_partition]_2 \oplus [x_rep]_2 \tag{3.1}$$

where *index_partition* is an index partition determined by the update time, *x_rep* is obtained using a space-filling curve, $[x]_2$ denotes the binary value of x, and $\oplus$ denotes concatenation. We proceed to elaborate this definition.

If we index timestamped object locations without differentiating them based on their timestamps, we not only lose the proximity preserving property of the space-filling curve; the index will also be ineffective in locating an object based on its *x_value*. To overcome such problems, we effectively "partition" the index, placing entries in partitions based on their update time. More specifically, we denote by Δt_{mu} the time duration that is the maximum duration in-between two updates of any object location. We then partition the time axis into intervals of duration Δt_{mu}, and we sub-partition each such interval into n equal-length sub-intervals, termed *phases*.

By mapping update times in the same phase to the same so-called *label timestamp* and by using the label timestamps as prefixes of the representations of the object locations, we obtain index partitions, and the times of updates determine the partitions the updates they go to. In particular, an update with timestamp t_u is assigned a label timestamp $t_{lab} = \lceil t_u + \Delta t_{mu}/n \rceil_l$, where operation $\lceil x \rceil_l$ returns the nearest future label timestamp of x.

For example, Figure 3.2 shows a B^x-tree with $n = 2$. Objects with timestamp $t_u = 0$ obtain label timestamp $t_{lab} = \frac{1}{2}\Delta t_{mu}$; objects with $0 < t_u \leq \frac{1}{2}\Delta t_{mu}$ obtain label timestamp $t_{lab} = \Delta t_{mu}$; and so on.

Next, for an object with label timestamp t_{lab}, we compute its position at t_{lab} according to its position and velocity at t_u. We then apply the space-filling curve

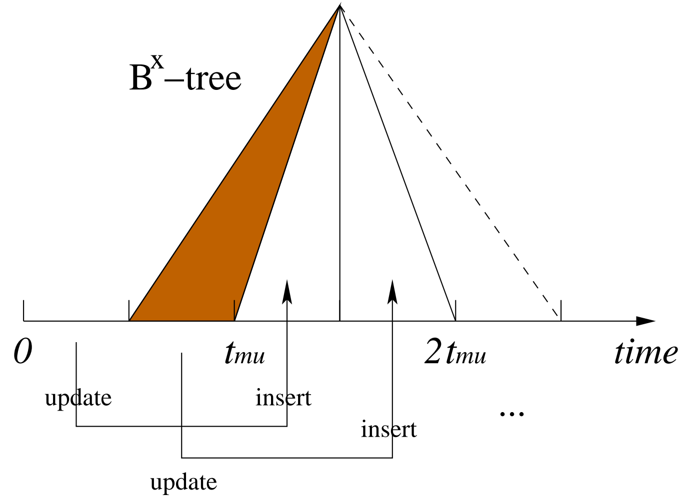


Figure 3.2: B^x-Tree with $n = 2$ Phases

to this (future) position to obtain the second component of Equation 3.1.

This mapping has two main advantages. First, it enables the tree to index object positions valid at different times, overcoming the limitation of the B^+-tree, which is only able to index a snapshot of all positions at the same time. Second, it reduces the update frequency: there is no need to update the positions of all objects each time only some of them are being updated. The two components of the mapping function in Equation 3.1 are consequently defined as follows:

$$index_partition = (t_{lab}/(\Delta t_{mu}/n) - 1) \bmod (n+1)$$

$$x_rep = x_value(\overrightarrow{x} + \overrightarrow{v} \cdot (t_{lab} - t_u))$$

With the transformation, the B^x-tree contains data belonging to $n+1$ phases, each given by a label timestamp and corresponding to a time interval. Within each of these, we apply a space-filling curve to an object position.

The choice of the value of n affects query performance and storage space. A large n results in smaller enlargements of query windows (detailed query algorithms will be presented in the following section), but also results in more partitions and

therefore a looser relationship among object locations. In addition, a large n incurs higher space overhead arising from more internal nodes. We will explore the optimal value of n in experimental studies.

To sum up, we formally define the B^x-tree as follows.

Definition 1 *The B^x-tree is a B^+-tree with the following properties:*

1. *Each entry of the leaf node is in the form of $\langle B^x value,\ v_1,\ v_2,\ t_u\rangle$.*
2. *Each internal node is in the form of $\langle pnt_0,\ B^x value_0,\ pnt_1,\ \ldots,\ B^x value_k,\ pnt_k\rangle$ ($k \leq m$, m is the node capacity).*

To exemplify, let $n = 2$, $\Delta t_{mu} = 120$, and assume a Peano curve of order 3 (i.e. the space domain is 8×8). Object positions given by $O_1 = ((7, 2), (-0.1, 0.05))$, $O_2 = ((0, 6), (0.2, -0.3))$, and $O_3 = ((1, 2), (0.1, 0.1))$ are inserted at times 0, 10 and 100, respectively. We calculate the $B^x value$ for each as follows:

Step 1: Calculate label timestamps and index partitions.

$t^1_{lab} = \lceil(0 + 120/2)\rceil_l = 60$, $\quad index_partition^1 = 0 = (00)_2$

$t^2_{lab} = \lceil(10 + 120/2)\rceil_l = 120$, $\quad index_partition^2 = 1 = (01)_2$

$t^3_{lab} = \lceil(100 + 120/2)\rceil_l = 180$, $\quad index_partition^3 = 2 = (10)_2$

Step 2: Calculate positions x_1, x_2, and x_3 at t^1_{lab}, t^2_{lab}, and t^3_{lab}, respectively.

$x'_1 = (1, 5)$

$x'_2 = (2, 3)$

$x'_3 = (4, 1)$

Step 3: Calculate Z-values.

$[Z_value(x'_1)]_2 = (010011)_2$

$[Z_value(x'_2)]_2 = (001101)_2$

$[Z_value(x'_3)]_2 = (100001)_2$

Step 4: Calculate $B^x value$s.

$$B^x value(O_1, 0) = (00)_2 \oplus (010011)_2 = (00010011)_2 = 19$$

$$B^x value(O_2, 10) = (01)_2 \oplus (001101)_2 = (01001101)_2 = 77$$

$$B^x value(O_3, 100) = (10)_2 \oplus (100001)_2 = (10100001)_2 = 161$$

Note that *at most n+1* ranges exist at a single point in time. As time passes, repeatedly the first range expires (shaded area in Figure 3.2), and a new range is appended (dashed line in Figure 3.2). This use of rolling ranges enables the B^x-tree to handle time effectively.

3.2.2 Querying

In this section, we outline the search strategies of the range query and the k nearest neighbor query in the B^x-tree.

Range Query

We consider the time interval range query instead of the more restricted timeslice range query provided for the basic version of the B^x-tree.

An interval range query retrieves all objects whose locations fall inside the rectangular range $q = ([qx_1^l, qx_1^u], [qx_2^l, qx_2^u])$ within a time interval $[t_s, t_e]$ where t_s is not prior to the current time ("l" denotes lower bound, and "u" denotes upper bound). The timeslice range query is the special case of the interval range query where t_s is equal to t_e.

A key challenge is to support predictive queries, i.e. queries that concern future times. Traditionally, indexes that use linear functions handle predictive queries by means of bounding rectangle (BR) enlargements (e.g. the TPR-tree family); to the best of our knowledge, no algorithm for predictive queries has been proposed for indexes that use snapshots of moving objects (e.g. the Lazy Update R-tree).

We present a generic approach to processing such queries that is not constrained by the base structure. Figure 3.7 outlines the range query algorithm, which we proceed to explain. To support queries on the anticipated, near future positions of objects, the B^x-tree uses query window enlargements instead of BR enlargements. This is done through the TimeParameterizedRegion function call in the algorithm (see Figure 3.8). Because the B^x-tree stores an object's location as of some time after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time.

We first consider the enlargement in the timeslice range query, and then generalize it to the time interval range query. Consider the example in Figure 3.3, where t_{ref} denotes the time when the locations of four moving objects are updated to their current values, and where predictive queries q_1 and q_2 (solid rectangles) have time parameters t_{q_1} and t_{q_2}, respectively. The figure shows the stored positions as solid dots, and the positions of the two first objects at t_{q_1} and the positions of the two last at t_{q_2} as circles. The two positions for each object are connected by an arrow.

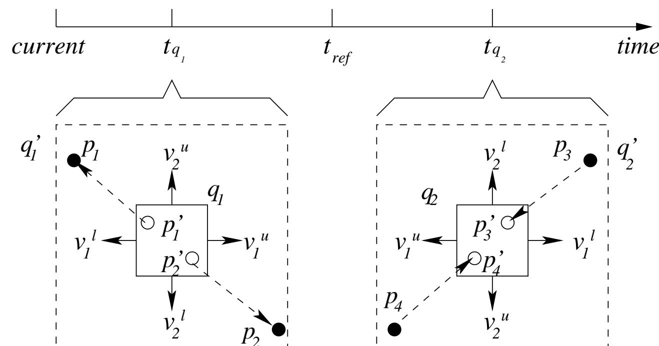


Figure 3.3: Query Window Enlargement

The relationship between the two positions for each object is $p'_i = p_i + \overrightarrow{v} \cdot (t_q - t_{ref})$. The first two of the four objects are thus in the result of the first query, and the last two objects are in the result of the second query. To obtain

this result, query rectangles need to be enlarged to the dashed rectangles as shown in Figure 3.3. We first compute the minimum and maximum velocities of objects inside the query rectangle in each dimension, denoted as $\overrightarrow{v_1^l}$, $\overrightarrow{v_2^l}$, $\overrightarrow{v_1^u}$, and $\overrightarrow{v_2^u}$. For q_1, the upper bound of the query rectangle is attached with the maximum velocity and the lower bound is attached with the minimum velocity. In contrast, the upper bound of the query rectangle of q_2 is attached with the minimum velocity, and the lower bound is attached with the maximum velocity.

In summary, the enlarged query window $q' = ([eqx_1^l, eqx_1^u], [eqx_2^l, eqx_2^u])$ is given as follows:

$$eqx_i^l = \begin{cases} qx_i^l + \overrightarrow{v_i^l} \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx_i^l + (-\overrightarrow{v_i^u}) \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \tag{3.2}$$

$$eqx_i^u = \begin{cases} qx_i^u + \overrightarrow{v_i^u} \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx_i^u + (-\overrightarrow{v_i^l}) \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \tag{3.3}$$

The computation of the enlarged query window q' proceeds in two steps. In the first step, we use the maximum speeds across all objects for obtaining a preliminary q'. In the second step, we try to reduce the query window, by using a two-dimensional speed histogram (e.g. a grid). In particular, for each partition in the B^x-tree, we create a speed histogram that for each cell captures the maximum and minimum projections of velocities onto the axes of the objects in the cell. We intersect the preliminary q' with the histogram, thus obtaining a set of cells which cover the q'. We then use the maximum speeds across these cells to enlarge the original q and obtain a new q' which may be smaller than before. This step can be repeated until q' no longer shrinks. The histogram of speeds can easily be maintained in main memory. Each time an object is updated, we adjust the speeds in the cell that this object belongs to. As time passes, repeatedly the oldest histogram expires with the associated partition, and a new histogram will be constructed.

For the interval range query, the situation of the enlargement becomes a little more complicated. The naive approach of computing the enlarged query window at each timestamp during the query interval is obviously inefficient. Because the enlarged query windows at different timestamps share the same center (the original query window), they usually overlap with one another, which leads to duplicated search in the overlapped region. Therefore, a better solution is to only retrieve objects in the largest query window after the enlargement, and then distinguish them by the individual timestamps.

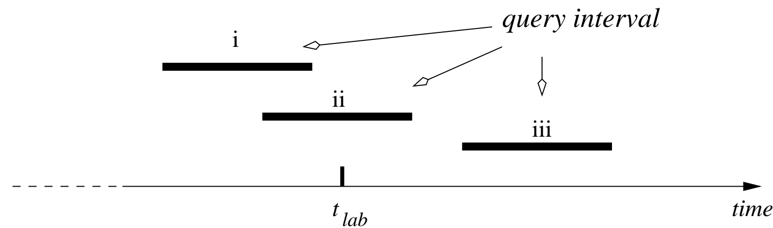


Figure 3.4: Possible Positions of a Query Interval w.r.t. a Label Timestamp

To identify the largest enlarged query window, we need to examine the possible positions of the query interval with respect to the label timestamp. Figure 3.4 illustrates how three cases may be discerned: (i) the query interval ends before the label timestamp t_{lab}; (ii) the query interval intersects t_{lab}; and (iii) the query interval starts after t_{lab}. The different types of intersections are handled by different strategies for the query window enlargement. For the first case, we use the start time of the query interval to perform a backward enlargement. Similarly, for the third case, we use the end time of the query interval to perform a forward enlargement. In the second case, the query interval is partitioned into two parts by the label timestamp. Both forward and backward enlargements of the query window are performed. Then we compare the sizes of the enlarged query windows and keep the larger one.

Next, we discuss the time argument of the query. In the optimized version of the

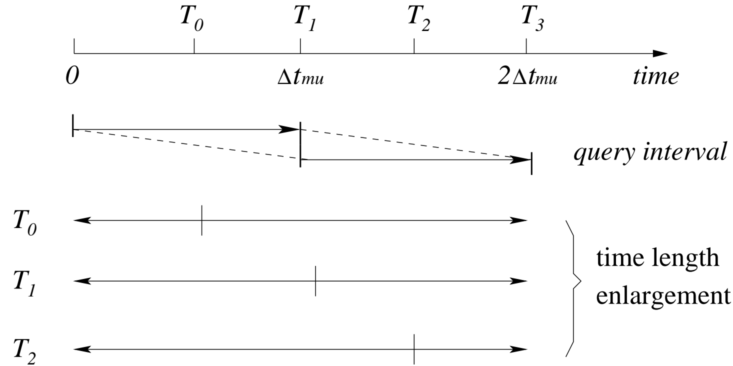


Figure 3.5: Time Length Enlargement

B^x-tree, we remove the constraint that the maximum enlargement is not allowed to exceed Δt_{mu}. Therefore, all the sub-trees are involved in a query. Considering the example in Figure 3.5, suppose queries are issued between $\frac{1}{2}\Delta t_{mu}$ and Δt_{mu} with the maximum predictive query interval equal to Δt_{mu}, and that T_0, T_1, and T_2 are the partitions corresponding to the label timestamps $\frac{1}{2}\Delta t_{mu}$, Δt_{mu}, and $\frac{3}{2}\Delta t_{mu}$, respectively. A query may have different time arguments of the query enlargement for the different partitions (or subtrees) it is applied to. The query on partition T_0 may be extended to time $2\Delta t_{mu}$ at most; the query on T_1 may be extended backward to time $\frac{1}{2}\Delta t_{mu}$ and forward to time $2\Delta t_{mu}$; the query on T_2 may be extended backward to time $\frac{1}{2}\Delta t_{mu}$ and forward to time $2\Delta t_{mu}$.

Having obtained the enlarged query window for each partition in the B^x-tree, we employ the algorithms for the range query in the B^+-tree with space-filling curves. The use of a space-filling curve means that a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space—see Figure 3.6. Hence, multiple range queries are to be processed. We optimize the processing by calculating the start and end points of the one-dimensional ranges and traverse the intervals by "jumping" in the index (as in [71]).

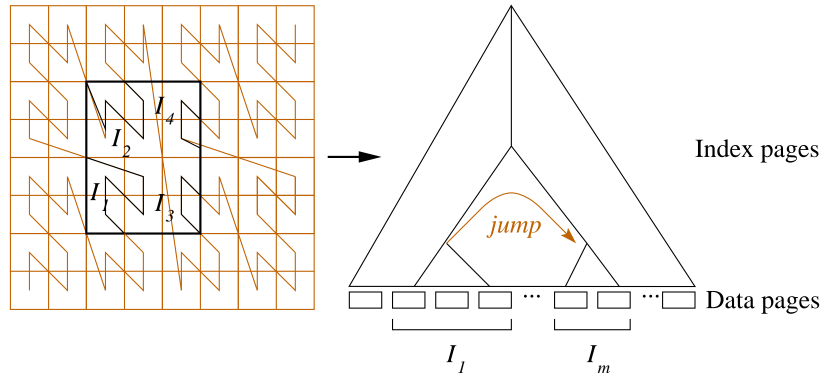


Figure 3.6: “Jump” in the Index

Let us step through the entire algorithm in detail. The pseudo code of the algorithm is shown in Figure 3.7.

For each partition of the B^x-tree, we first enlarge the query window q to q' by means of function *TimeParameterizedRegion* (see Figure 3.8), where the three cases of enlargement discussed earlier are tested.

Having obtained an enlarged query window, we traverse the index. Suppose the intervals of the transformed range query are given by the sequence of *x_values* $i_1, i_2, \ldots, i_{2m-1}, i_{2m}$ in ascending order. The pair of values (i_{2j-1}, i_{2j}) start and end interval I_j $(1 \leq j \leq m)$, and the number of intervals is m.

We first calculate the start and end points of this sequence of intervals (denoted as q'_s and q'_e). We have that q'_s equals i_1 and that q'_e equals i_{2m}. Then we start the search from the leftmost interval I_1. The leaf node containing the start point of I_1 is located, and all entries in this leaf node that are after the start point are checked. If the last entry is smaller than the end point, we continue traversing right siblings by the right link (lines 8–11). Otherwise, we compute the next leftmost interval I_j by function LeftmostInterval (line 6), where j can exceed 2 because intervals contained in the leaf nodes that have already been accessed will be skipped. In this way, we improve the query performance compared with the basic B^x-tree version

---

Algorithm Range_query (q, t_s, t_e)

Input: q is the query range and $[t_s, t_e]$ is the query time interval

```
1.    for i ← 0 to n
2.        q' ← TimeParameterizedRegion(q, t_s, t_e, i)
3.        calculate start and end points q'_s, q'_e for q'
4.        low ← 0
5.        repeat
6.            [start, end] ← LeftmostInterval([q'_s, q'_e], [low, ∞))
7.            locate leaf node containing start
8.            repeat
9.                store objects with x_rep ≥ start in L
10.               follow the right pointer to the sibling node
11.           until entry.x_rep ≤ end, for the node's last entry
12.           low ← entry.x_rep + 1
13.       until LeftmostInterval([q'_s, q'_e], [low, ∞)) = ∅
14.   for each object in L do
15.       if the object's position during [t_s, t_e] is inside q then
16.           add the object to the result_set
17.   return result_set
end Range_query.
```

Figure 3.7: Range Query Algorithm

which needs to check all the intervals and may lead to some redundant accesses.

To find the start point of I_j, we backtrack to a higher level where one "jumping" occurs; there we proceed to retrieve the objects with positions in interval I_j. This takes place in line 7 of the algorithm, where traversal from the root is avoided when possible. When all the intervals have been checked in this manner, we have obtained the set of all objects that may possibly belong to the result of range query q. For each object, we check its positions during $[t_s, t_e]$ and return only those objects whose positions are actually in the query window q (lines 14–17).

Algorithm TimeParameterizedRegion (q, t_s, t_e, i)
Input: q is the query range, $[t_s, t_e]$ is the query time interval,
and i is the partition number

1. $t^i_{lab} \leftarrow$ the label timestamp of partition i
2. **if** $t_e \leq t^i_{lab}$ **then**
3. enlarge q backward to t_s, and store the result in q'
4. **else**
5. **if** $t_s \geq t^i_{lab}$ **then**
6. enlarge q forward to t_e, and store the result in q'
7. **else**
8. enlarge q backward to t_s, and store the results in q_1
9. enlarge q forward to t_e, and store the results in q_2
10. $q' \leftarrow$ Max(q_1, q_2)
11. **return** q'

end TimeParameterizedRegion.

Figure 3.8: Function TimeParameterizedRegion()

Claim 1 *For any range query, our range query algorithm can find correct answers (i.e. all the qualified objects).*

Proof 1 *Our range query algorithm consists of two main steps: (i) enlarge original query range; (ii) retrieve objects in the enlarged query range. The second step is supported by an existing algorithm for the B^+-tree with the space-filling curve, and hence its correctness has already been proved. Here, we only need to prove that the enlarged query range contains all qualified objects. We first look at a timeslice range query and then extend our discussion to the interval range query.*

Given a timeslice query $q = ([qx^l_1, qx^u_1], [qx^l_2, qx^u_2])$ with query time t_s. Suppose for a partition labelled with t_{ref}, there is a qualified object $O(x_1,x_2,v_1,v_2)$ outside the enlarged query range $q' = ([eqx^l_1, eqx^u_1],\ [eqx^l_2, eqx^u_2])$. Without loss of generality,

we assume that t_s is after t_{ref} and the object O is on the left of the enlarged query range.

From the condition that the object O is outside the range of q', we have the following inequality:

$$x_1 < eqx_1^l = qx_1^l - v_1^u(t_s - t_{ref})$$

Since O is a qualified object, its position at t_s should inside the range of query q, which results in the following inequality.

$$qx_1^l - x_1 \leq v_1(t_s - t_{ref})$$

Consider the above two inequalities together, we have $v_1^u < v_1$, which contradicts the condition that $v_1^u < v_1$ (v_1^u is the maximum velocity of this dimension). Therefore, the hypothesis is invalid. As a consequence, the algorithm of the timeslice query is correct.

An interval query can be seen as a set of timeslice queries. For each timeslice query, there is a corresponding enlarged query range. Our algorithm selects the maximum enlarged query range which covers the enlarged ranges for all timesslice queries, and hence we can find the correct answers. □

k Nearest Neighbor Query

Assuming a set of $N > k$ objects and given a query object with position $q = (qx_1, qx_2)$, the k nearest neighbor query (kNN query) retrieves k objects for which no other objects are nearer to the query object at time t_q not prior to the current time.

We compute this query by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. The algorithm is

outlined in Figure 3.9. We first construct a range R_{q1} centered at q and with extension $r_q = D_k/k$, where D_k is the estimated distance between the query object and its k'th nearest neighbor; D_k can be estimated by the equation [91]:

$$D_k = \frac{2}{\sqrt{\pi}}\left[1 - \sqrt{1 - \left(\frac{k}{N}\right)^{\frac{1}{2}}}\right]$$

We compute the range query with range R_{q1} at time t_q, by enlarging it to a range R'_{q1} and proceeding as described in the previous section. If at least k objects

---

Algorithm k**NN_query**$(q(qx_1, qx_2), k, t_q)$
Input: a query point $q(qx_1, qx_2)$, a number k of neighbors,
    and a query time t_q

1. construct range R_{q1} with q as center and extension r_q
2. $R'_{q1} \leftarrow$ TimeParameterizedRegion(R_{q1}, t_q)
3. *flag* $\leftarrow$ true   *// not enough objects*
4. $i \leftarrow 1$ *// first query region is being searched*
5. **while** *flag*
6.   **if** $i = 1$ **then**
7.   find all objects in region R'_{q1}
8.   **else**
9.   find all objects in region $R'_{qi} - R'_{qi-1}$
10.   **if** k objects exist in inscribed circle of R_{qi} **then**
11.   *flag* $\leftarrow$ false
12.   **else**
13. $i \leftarrow i + 1$
14. $R_{qi} \leftarrow$ Enlarge(R_{qi-1}, r_q)
15. $R'_{qi} \leftarrow$ TimeParameterizedRegion(R_{qi}, t_q)
16. **return** k NNs with respect to q

end kNN_query.

Figure 3.9: kNN Query Algorithm

are currently covered by R'_{q1} and are enclosed in the inscribed circle of R_{q1} at time t_q, the kNN algorithm returns the k nearest objects and then stops. It is safe to stop because we have considered all the objects that can possibly be in the result.

Otherwise, we extend R_{q1} by r_q to obtain R_{q2} and an enlarged window R'_{q2}. This time, we search the region $R'_{q2} - R'_{q1}$ and adjust the neighbor list accordingly. This process is repeated until we obtain an R_{qi} so that there are k objects within its inscribed circle.

In some B$^+$-tree implementations, leaf nodes are not only chained left to right, but also right to left. The kNN search algorithm can exploit right to left sibling pointers to avoid always having to traverse the tree from the root when an interval is extended for a next iterative range search. This reduces the search cost but increases the update cost.

3.2.3 Insertion, Deletion, and Migration

The insertion algorithm is straightforward. Given a new object, we calculate its index key according to Equation 3.1 and then insert it into the B^x-tree as in the B$^+$-tree. To delete an object, we use the most recently provided positional information for the object. In particular, using the positional information provided when a new position for the object was most recently inserted and the time of that insertion, we calculate the object's index key and employ the same deletion algorithm as in the B$^+$-tree. Updates are simply combinations of deletions and insertions. It can be observed that these B^x-tree operations inherit the good properties of the B$^+$-tree, and we expect very good update performance.

Next, it should be noted that the B^x-tree does differ from the B$^+$-tree in how updates are applied. The B^x-tree clusters updates during a certain time period to one time point and maintains several sub-trees corresponding to different time

intervals. For example (see Figure 3.10), objects updated between t_0 and t_1 are stored in partition T_0; objects updated between t_1 and t_2 are stored in T_1; etc. T_0, T_1, and T_2 co-exist before t_3. From t_3 to t_4, T_1, T_2, and T_3 co-exist, and T_0 has expired. The total size of three sub-trees is equal to that of one tree indexing all the objects.

In some applications, there may be some object positions that are updated relatively rarely. For example, it may be that most objects are updated at least every 10 minutes, while a few objects are updated only once a day. Instead of letting outliers force a large maximum update interval, we use a "maximum" update interval within which a high percentage of the objects have been updated.

Object positions that are not updated within this interval are "migrated" to a new partition using their positions at the label timestamp of the new partition. In the example shown in Figure 3.10, suppose that some object positions in T_0 are not updated at the time when T_0 expires. At this time, we move these objects to T_3. Although this introduces additional update costs, the (controllable) amortized cost is expected to be very small since outliers are rare.

The forced movement of an object's position to a new partition causes no problems in locating the object, since the new partition can be calculated based on the original update time. Likewise, query efficiency is not affected.

For example, consider an object, the position of which has been migrated from

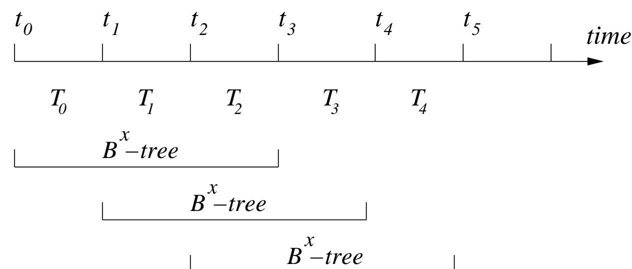


Figure 3.10: B^x-Tree Evolution

T_0 to T_3. If the object issues an update during the time when partitions T_1, T_2, and T_3 exist, we note that partition T_0 to which the old position belongs no longer exists. This implies that the object's position has been migrated. Then we replay the migration procedure to determine that the position is now in T_3.

Finally, we would like to mention that the correctness of the update algorithm is guaranteed since the B^x-tree has the same structure as the B^+-tree. The only difference is the key value computation which does not determine the correctness.

3.3 Performance Studies

In this section, we cover extensive experimental studies conducted on the B^x-tree.First, we introduce the experimental settings. Then we study the properties of the B^x-tree. Finally, we report on a comparison with the TPR*-tree and the B^{dual}-tree.

3.3.1 Experimental Settings

Two versions of the B^x-tree are implemented: B^x (Z-curve) and B^x (H-curve), denoting the B^x-tree using the Peano and the Hilbert curve, respectively. As a reference, we compare the B^x-trees against the TPR*-tree (the source code is obtained from the author's website) and the B^{dual}-tree (representing structures based on dual transformations). All the experiments are conducted on a 2.6G PentiumIV Personal Computer with 1 Gbyte of memory. The disk page size is 4K, which results in node capacities of 200, 256 and 332 of the TPR*-tree, the B^{dual}-tree and the B^x-tree, respectively. The default LRU buffer size is 50 pages, which is enough to hold all the internal nodes of the B^x-tree.

The workloads that we subject the indexes to use synthetic datasets of moving

objects with positions in a space domain of 1000×1000 distance units. In most experiments, we use uniform data, where object positions are chosen randomly, where the objects move in a randomly chosen direction, and where a speed ranging from 0 to 3 is chosen at random. A similar number of objects are updated at each time unit during a simulation, and all objects are updated at least once within the maximum update interval. One may think of the unit of space being kilometer and the unit of speed being kilometer per minute.

Other datasets were generated using an existing data generator, where objects move in a network of two-way routes that connect a given number of uniformly distributed destinations [76]. Objects start at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Whenever an object reaches one of the destinations, it chooses the next target destination at random. Objects accelerate as they leave a destination, and they decelerate as they approach a destination.

For each dataset, we construct the indexes at time 0 and measure the average query cost after the simulation has run for 120 time units (i.e. a maximum update interval), which is used as a standard workload in all experiments. Each query has three parameters: (i) query size, (ii) query interval, and (iii) predictive length. Unless noted otherwise, we test 200 square timeslice range queries with side length 50 and with predictive length uniformly distributed in the range $[0, 120]$.

The parameters used are summarized in Table 3.1, where values in bold denote the default values used.

3.3.2 Filter Rate

Before the comparison with the other indexes, we examine properties of the B^x-tree itself, starting with the filter rate. In this study, we consider the filter and

Parameter	Setting
Page size	4K
Buffer size	**50 pages**, ..., 250 pages
Number of phases	1, 2, **3**, 4, 5, 6
Maximum update interval	60, **120**, 180, 240
Maximum predictive length	120
Query interval length	**0**, 10, ..., 100
Query window size	10, . . . , **50**, . . . , 100
Number of neighbors, k	1, 10, 20, 30, 40, 50
Number of queries	100
Dataset size	**100K**, . . . , 1M
Dataset	**Uniform**, Network-based
Space-filling curve	Z, H

Table 3.1: Parameters and Their Settings

refinement steps separately in the B^x-tree, and we compute the filter rate in terms of both objects accessed and nodes accessed.

The object filter rate for a query is computed as N_{acto}/N_{reto}, where N_{acto} is the number of objects in the (final) query result, and N_{reto} is the total number of objects accessed during the query processing. The ideal filter rate is 1, meaning that no objects are accessed that are not in the final result.

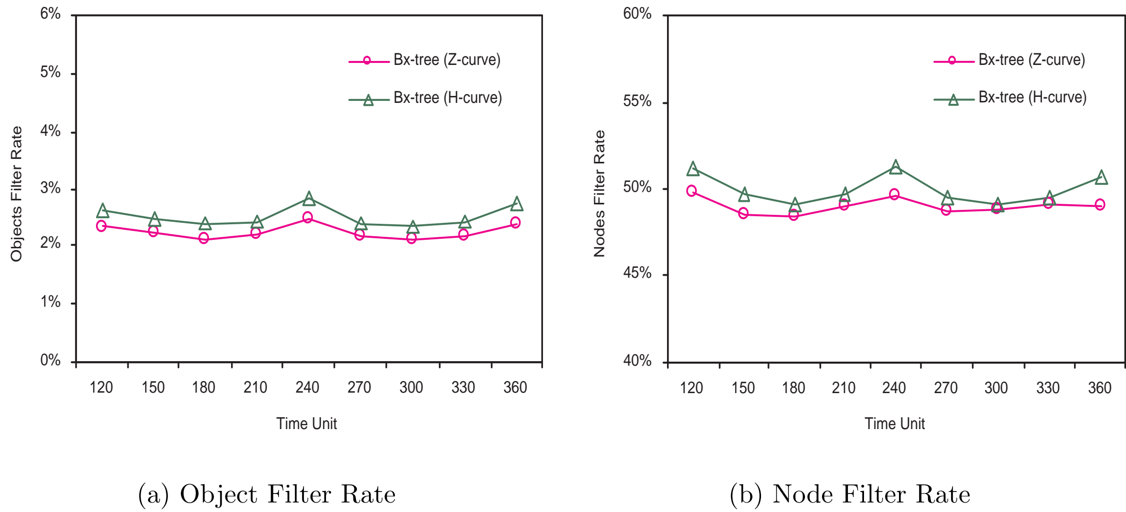
Similarly, the node filter rate of a query is computed as N_{actn}/N_{retn}, where N_{actn} is the number of nodes accessed containing objects contributing to the (final) result, and N_{retn} is the total number of nodes accessed during the query processing. The experiments that follow report the two types of filter rates with respect to various parameters.

Effect of Time

In this experiment, we compute the two types of filter rate after every 30 time units in a 100K dataset, and the indexes run for 360 time units. Since the first 120 time units are used to populate the trees with objects inserted at different times, we

only consider their performance from time 120.

Figures 3.11 (a) and (b) show the object filter rate and the node filter rate for the two B^x-trees, respectively. We can see that the B^x-tree (H-curve) exhibits a



(a) Object Filter Rate

(b) Node Filter Rate

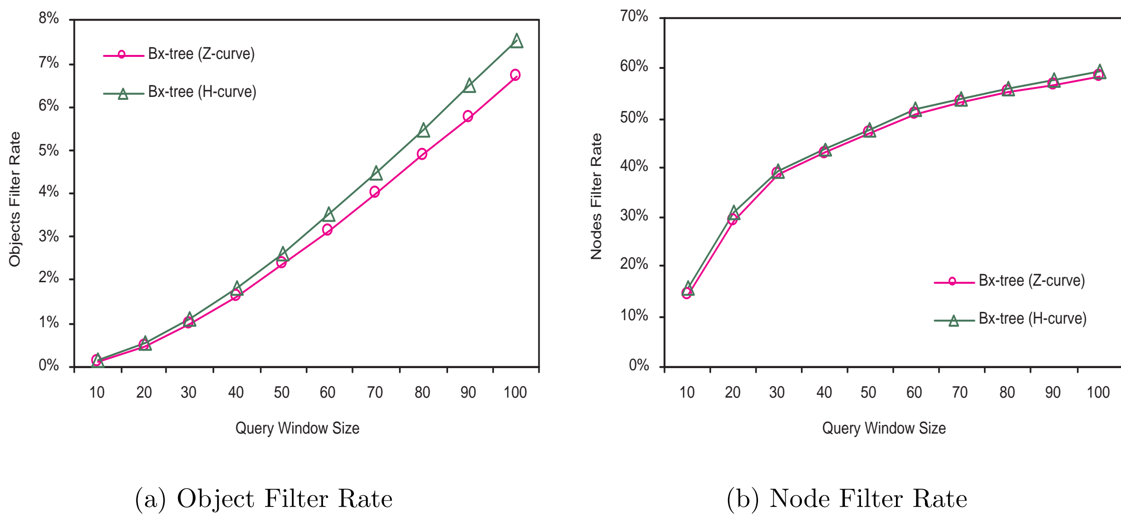
Figure 3.11: Filter Rates for Varying Query Time

better filter rate than does the B^x-tree (Z-curve). This is because the numbers of objects in the results are the same for these two, while the numbers of objects retrieved by the B^x-tree (H-curve) are smaller than those for the B^x-tree (Z-curve). This is so because the B^x-tree (H-curve) accesses fewer nodes than does the B^x-tree (Z-curve) during query processing. Later experiments will demonstrate this.

Moreover, the object filter rates are much smaller than the node filter rates. This is because each time a node is retrieved, all objects in the node are checked. In the B^x-tree, the node capacity is large, which means one node can contain many objects. Therefore, when the objects in the query answer set spread across several nodes, a large number of objects are retrieved.

Effect of Query Window Size

In this section, we vary the query window size and examine the resulting filter rates. The results are shown in Figure 3.12. When the query window size increases, both types of filter rates increase. The increase in the filter rates is mainly due to the increase in the query selectivity, i.e. the increase in the numbers of objects in the results.



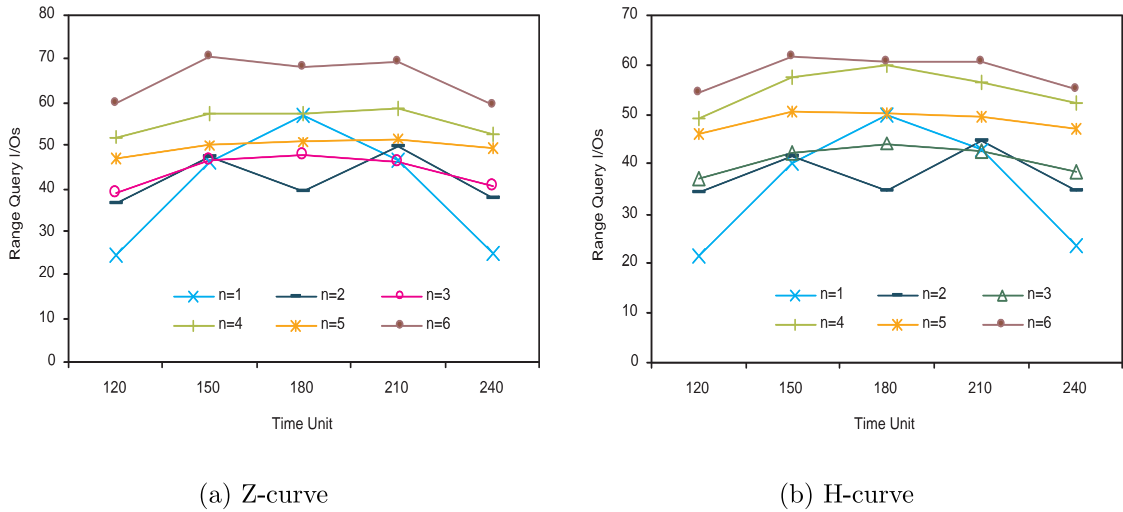
(a) Object Filter Rate (b) Node Filter Rate

Figure 3.12: Filter Rates for Varying Query Window Size

3.3.3 Number of Sub-intervals, n

The choice of the number of phases n in a B^x-tree affects the query performance. A large n results in smaller enlargements of query windows, but also more partitions and therefore a looser relationship among object locations. This experiment is aimed at estimating the best value of n for the experimental setting considered.

Figure 3.13 shows the range query cost for the two B^x-trees at different timestamps with n ranging from 1 to 6. From the figure, we may observe the following: First, the B^x-tree (Z-curve) and the B^x-tree (H-curve) exhibit similar trends in



(a) Z-curve

(b) H-curve

Figure 3.13: Range Query Performance for Varying n and Query Time

performance for the various values of n. Second, the performance curves of both B^x-trees oscillate as time passes. The bigger the value of n, the smaller the oscillation. Third, there are two types of patterns of performance curves corresponding to even and odd values of n. Finally, we observe that both B^x-trees with n equal to 3 perform best at most times (only at some points, the performance is worse than for n equal to 1 and 2).

To further explore the performance of the B^x-tree with various n, Figure 3.14 plots the average range query cost during the time period $[120, 240]$. We can see clearly that the B^x-trees achieve best performance when $n = 3$. Therefore, $n = 3$ is used as the default setting in the following experiments. Note that n is set to 2 in the basic version of the B^x-tree which may not be the optimal choice.

The behavior of the B^x-trees is influenced by two factors: the number of objects in each partition and the time length of the query window enlargement, both of which change with time. For example, when $n = 1$, there are at most two partitions in one B^x-tree. At time 0, the B^x-tree has only one partition. As time passes, objects in the first partition are updated gradually and inserted into the

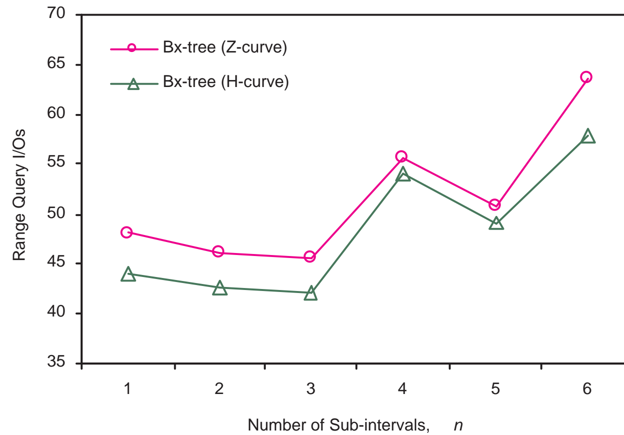


Figure 3.14: Average Range Query Performance for Varying n

second partition. Meanwhile, the average time length of the enlargement in the first partition increases, and in the second partition, this value decreases first and then increases. Viewed separately, if the time length of the enlargement is fixed, the fewer the objects there are in the partition, the lower the query cost will be; if the number of objects is fixed, the shorter the time length of the enlargement is, the better the query performance will be. The resultant performance of the B^x-trees can be seen as the combination of the two effects. Moreover, the performance of the B^x-tree stabilizes with the increase of n since there are less differences in numbers of objects and time lengths due to more partitions.

3.3.4 Range Query

We now compare the range query performance of the TPR*-tree and the B^{dual}-tree against that of the two B^x-trees. We experiment with a wide range of workloads with different combinations of parameter settings. We evaluate the effect of varying the buffer size, query time, query window size, query interval length, object speed, data distribution and dataset size.

Effect of Buffer Sizes

We employ an LRU buffer and study the effects of different buffer sizes, by varying the number of buffer pages (see Figure 3.15). As is generally the case for indexes,

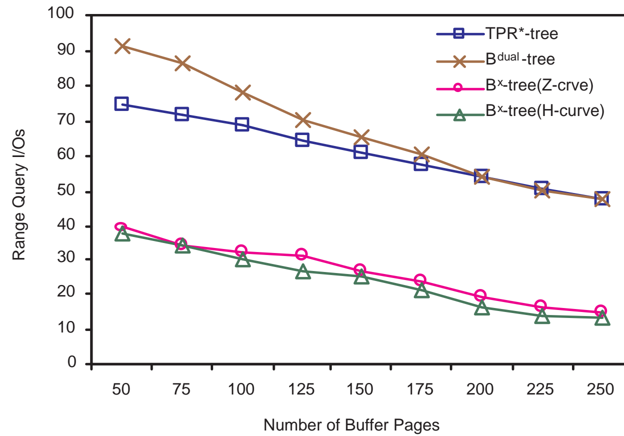


Figure 3.15: Effect of Varying Buffer Size

all indexes experience reduced I/Os as the buffer size increases. Since the B^x-tree incurs fewer page reads originally, the effect of an increasing buffer size on the index is consequently more pronounced. Specifically, the B^x-trees save about 60% I/Os by using 250 buffer pages compared with that using 50 buffer pages, whereas the TPR*-tree saves 40% and the B^{dual}-tree saves 50%.

Effect of Query Time

To study the search performance of the indexes evolving with the passage of time, we compute the query cost using the same 100 timestamp range queries with a query window size of 50, after every 30 time units on a 100K dataset. The workload duration is 360 time units (three maximum update intervals).

Figure 3.16 summarizes the results. As shown, the B^x-trees always perform best among all the indexes, and the TPR*-tree and the B^{dual}-tree have the similar query cost. This is because the TPR*-tree experiences continuous enlargements of the

MBRs which are not updated as time passes, no matter how carefully it chooses the position to insert an object. The B^{dual}-tree employs similar search algorithms as the TPR*-tree and hence also suffers from the MBRs overlapping problem. Moreover, the B^{dual}-tree even performs a little worse than the TPR*-tree since each node in the B^{dual}-tree is associated with multiple MBRs which may increase the chance of overlapping.

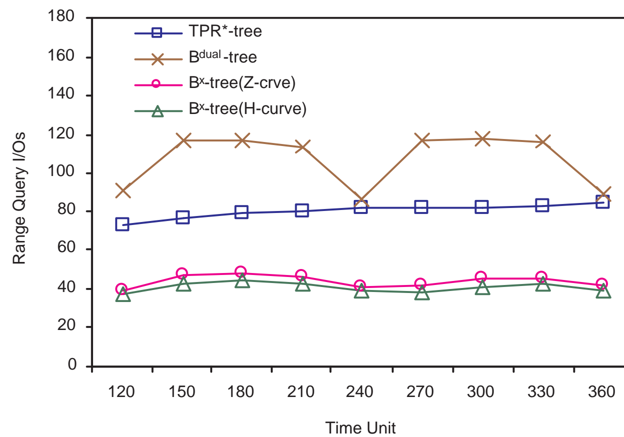


Figure 3.16: Effect of Varying Query Time

We also note that the query cost of all indexes is almost not affected by the query time. The periodical behavior of the B^{dual}-tree and the B^x-trees is caused by the use of index partition (i.e. multiple subtrees).

In addition, the B^x-tree (H-curve) achieves better performance than the B^x-tree (Z-curve) because the Hilbert curve generates a better distance-preserving mapping than does the Peano curve, and hence yields fewer search intervals on the B^x-tree, i.e., fewer disk accesses. This result also suggests that Bx-trees with better mapping techniques may achieve better performance.

Effect of Query Window Sizes

We next investigate the effect of varying query window sizes by varying the square window length from 10 to 100 for a dataset of size 100K. As expected, the results in Figure 3.17 show that the query costs increase with an increasing query window size. Larger windows contain more objects and therefore lead to more node accesses.

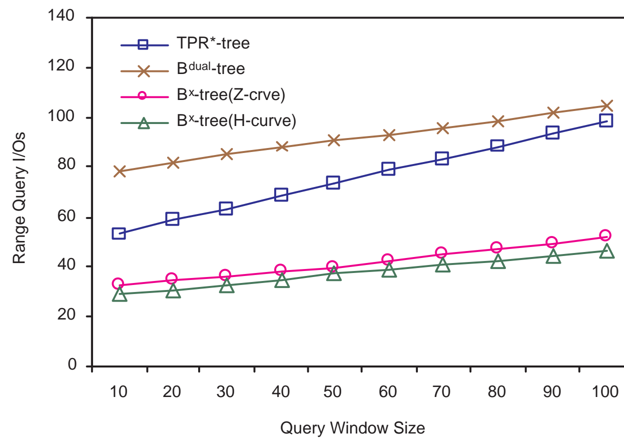


Figure 3.17: Effect of Query Window Size

However, the effect is slightly more obvious on the TPR*-tree. This is so because the TPR*-tree benefits relatively more from small query windows than do the other two types of trees. This ability of the TPR*-tree leads to performance degeneration when the query window becomes large. Specifically, when the window size reaches 100, the TPR*-tree cost is twice those of the B^x-trees.

Effect of Query Interval Lengths

In addition to considering the effect of varying query window size, we also study the effect of query interval length, varying it from 0 to 60 time units. The results are shown in Figure 3.18. It is not surprising that the query cost of all the indexes increases with the query interval length, as more moving objects may intersect the query window during a longer query time interval. Here, all the indexes degrade at

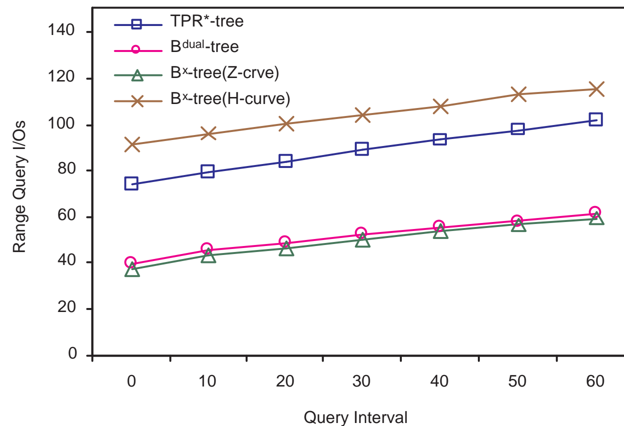


Figure 3.18: Effect of Varying Query Interval Length

similar rates. This is because both the B^{dual}-tree and the TPR*-tree are optimized for query intervals within the horizon parameter (i.e., how far into the future the insertion algorithm of the index assumes that queries "see" the bounding rectangles created; here, this parameter is 240 time units), and the increased query cost of all the indexes is mainly due to the increased number of the answers.

Effect of Object Speeds

Another factor that may affect the performance of the B^x-tree is object speed, which co-determines the enlarged query window sizes, together with the time length of the enlargements. Hence, in this experiment, we study the effect of the speeds of the moving objects on the TPR*-tree, the B^{dual}-tree and the B^x-trees, by varying the maximum speed from 1 to 3, choosing object speeds at random from 0 to the maximum speed.

As shown in Figure 3.19, all the indexes yield better performance when the speeds of the moving objects decrease. This is because the MBRs in the TPR*-tree and the B^{dual}-tree obtain smaller expanding speeds and because the enlargements made to query windows for the B^x-trees also become smaller.

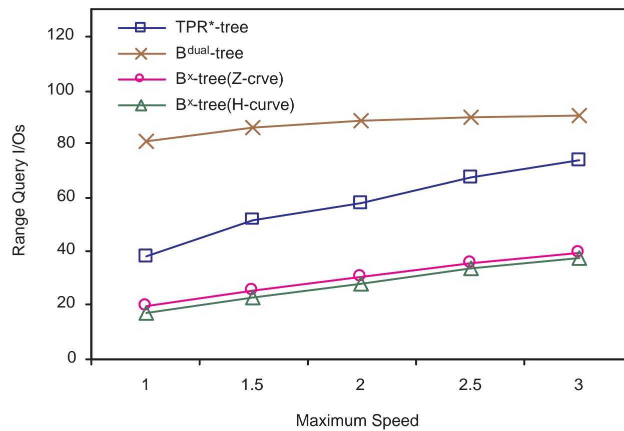


Figure 3.19: Effect of Maximum Speed on Range Query Performance

Effect of Data Distributions

This experiment uses the road network dataset to study the effect of data distributions on the indexes. The dataset contains 100K data points. Figure 3.20 shows the range query cost when the number of destinations in the simulated network of routes is varied. The term "uniform" in the figure indicates the case where the objects can choose their moving directions freely.

Observe that the query cost in the TPR*-tree increases with the number of destinations, and the cost is sometimes larger than that of the uniform dataset. The

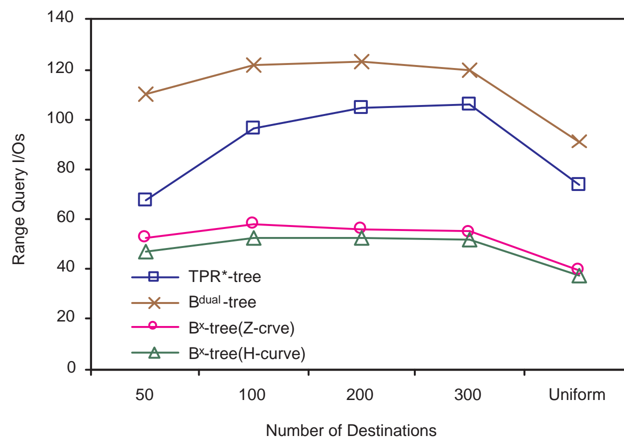


Figure 3.20: Effect of Data Distribution on Range Query Performance

possible reasons of the behavior are as follows. When there are only a few number of destinations, objects moving in the same directions may be well clustered by the TPR*-tree. However, when the number of destinations is big, objects assigned to each road decreases, and hence the TPR*-tree may put objects on different roads in one MBR which results in either more dead space or heavier overlaps. In contrast, the performance of the B^x-trees is not much affected by the data skew because objects are stored using space-filling curves, meaning that the density has less of an effect on the index. For the B^{dual}-tree, though it stores the objects by space-filling curves, its query still uses the MBRs, and therefore its performance pattern is in-between the TPR*-tree and the B^x-tree.

Effect of Data Sizes

At the end of the experiments on the range query, we study the scalability of the TPR*-tree, the B^{dual}-tree and the B^x-trees. We vary the number of moving objects from 100K to 1M. Figure 3.21 shows the average number of I/O operations for each index. We can observe that all the indexes scale well. When the dataset size is 10 times bigger, the query cost only increases about 6 times. This behavior may

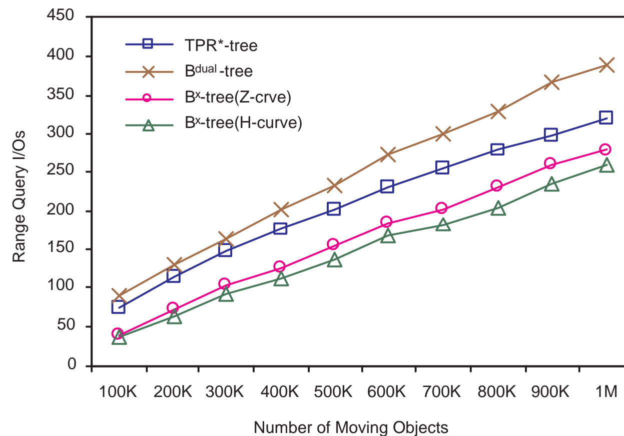


Figure 3.21: Effect of Data Sizes on Range Query Performance

be explained as follows: The TPR*-tree carefully chooses the path to insert an object which leads to less overlap during a certain time interval. In the B^{dual}-tree and the B^x-trees, every object has a linear order which is determined by the space domain (or also velocity domain) and is relatively independent of the number of moving objects. As the dataset grows, the range query cost of the B^{dual}-tree and the B^x-trees increases mainly due to the increase in the number of objects inside the range.

3.3.5 *k*NN Query

We proceed to evaluate the efficiency of *k*NN queries using the same settings as for range queries. The performance difference between the TPR*-tree and the B^x-tree of the *k*NN queries exhibits a behavior similar to that of range queries. The B^x-tree's *k*NN search algorithm is essentially an incremental range query algorithm; hence the results exhibit similar patterns as the results for range queries. Here, we present a representative result which is the effect on performance of the number of k of required nearest neighbors. As shown in Figure 3.22, with the increase of k, the search cost increases slightly for both indexes. Due to the data size and side

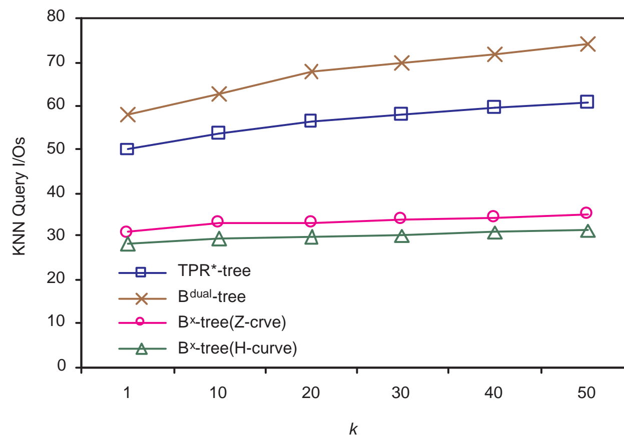


Figure 3.22: Effect of k on kNN Query Performance

effect of the query and MBR enlargement, the effect of k is not very significant.

3.3.6 Update

We now compare the average update cost of the B^x-trees against that of the TPR*-tree and the B^{dual}-tree. Note that for each update, one deletion and one insertion are issued, which leaves the size of the tree unchanged.

Effect of Time

First, we investigate performance degradation across time. We measure the performance of all the indexes after every 30 time units. Figure 3.23 shows the update cost as a function of the time. We can see that the B^{dual}-tree and both variants

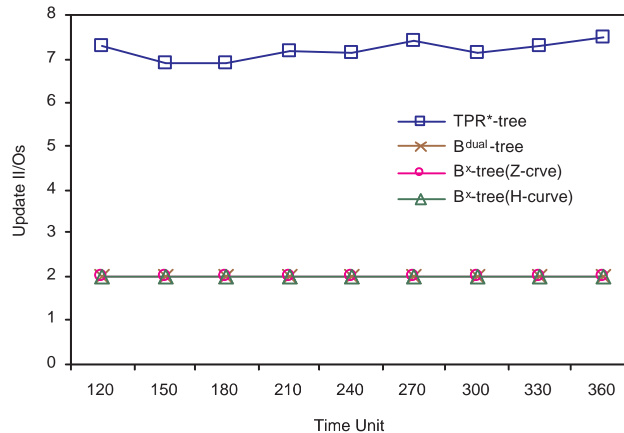


Figure 3.23: Effect of Varying Update Time on the Update Cost

of the B^x-tree achieve significant improvement over the TPR*-tree. In most cases, one update in the B^{dual}-tree or B^x-trees only incurs two I/O operations since they are all B^+-tree based indexes. However, the update cost in the TPR*-tree keeps increasing with time for the time duration considered in the experiment. This is because in the B^x-trees, given the key, an insertion, and a deletion need to travel down one path only. As all the internal nodes of the B^x-tree are stored in the buffer,

the update cost is reduced to one leaf node access. In the TPR*-tree, each deletion entails a search to retrieve the object to be removed, which results in traversing multiple (partial) paths from the root toward the leaf level. We attribute the degrading performance to MBRs that overlap increasingly with the passing of time.

In this experiment, the performance of the B^x-tree (Z-curve) and the B^x-tree (H-curve) are comparable, since the update efficiency is independent of the spatial proximity preservation.

Effect of Update Interval Length

Next, we study the effect of the maximum update interval length on the indexes, by varying this parameter from 60 to 240. Figure 3.24 shows the average update cost after the workload has run for one maximum update interval. We observe

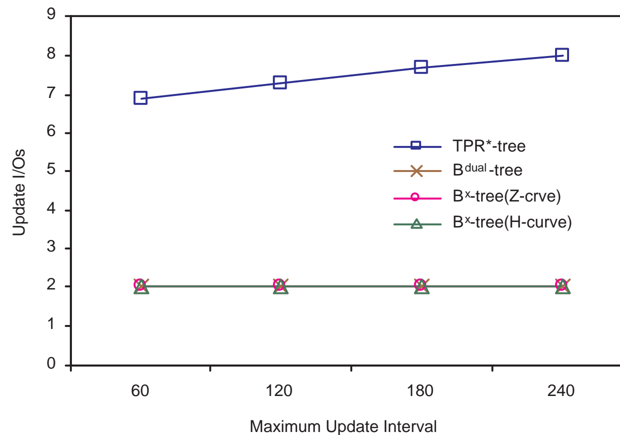


Figure 3.24: Effect of Varying Maximum Update Interval on Update Performance

that the performance of the TPR*-tree degrades as the maximum update interval increases, whereas the B^{dual}-tree and the B^x-trees are not affected. The main reason is that, as the update interval increases, the overlap among MBRs becomes more severe and thus affects the performance of the TPR*-tree. In contrast, the update operations in the B^{dual}-tree and the B^x-trees depend only on the key values, which

do not change over time.

Effect of Data Sizes

In this experiment, we examine the update performance for varying dataset size. We compute the average update cost after the maximum update interval of 120 time units. From Figure 3.25, we can see that the B^+-tree based indexes, i.e. the B^{dual}-tree and the B^x-trees, consistently maintain very good performance. The

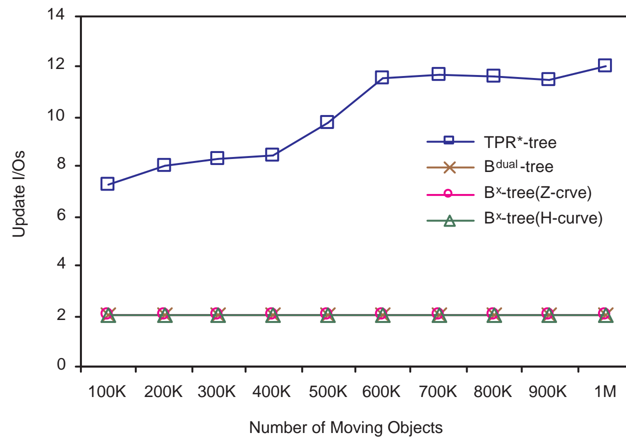


Figure 3.25: Effect of Data Sizes on Update Cost

gap between the TPR*-tree and the B^+-tree based indexes widens as the dataset grows in size. For the dataset of 1M objects, the cost of the TPR*-tree is nearly six times that of the other three trees. One reason for the degeneration of the TPR*-tree is that the MBRs have higher probabilities of overlapping when the number of objects increases, which results in multiple search paths. We note that this cost can be reduced by maintaining a hash-table to locate objects quickly, and performing a bottom-up update as in [46]. However, such an auxiliary structure incurs an additional storage overhead and complicates memory management and concurrency control.

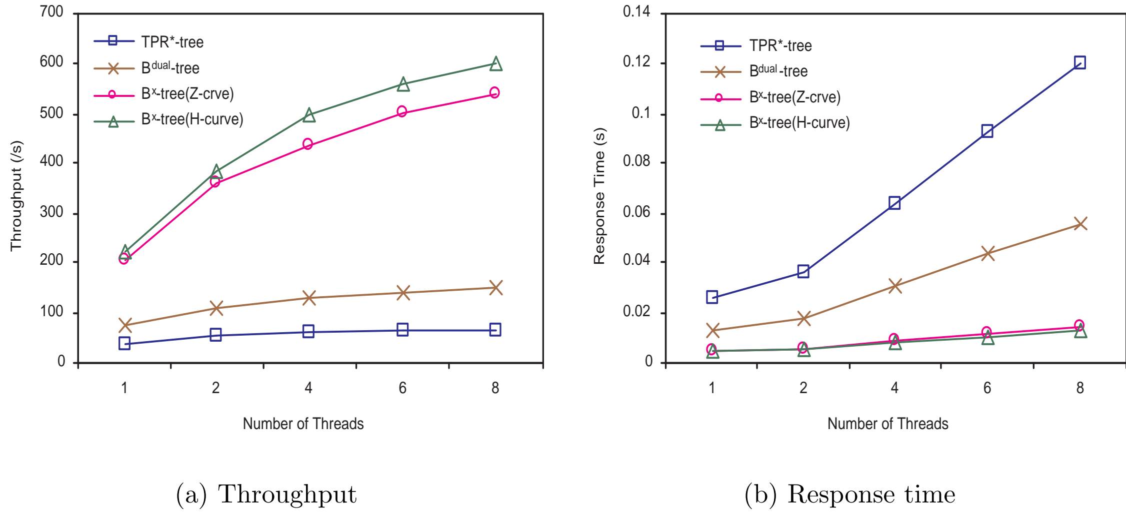
For the B^{dual}-tree and the B^x-trees, the update operation needs to traverse only

one path, no matter how large the dataset is. Thus the cost of updates in the B^{dual}-tree and the B^{x}-tree is only related to the height of the tree.

3.3.7 Effect of Concurrent Accesses

In this section, we compare the concurrent performance of the TPR*-tree, the B^{dual}-tree and the B^{x}-trees. We implemented the R-link technique for the TPR*-tree and the B-link technique for the B^{dual}-tree and the B^{x}-tree. We use multi-threaded programs to simulate a multi-user environment. The number of threads varies from 1 to 8. Workloads contain 20% queries and 80% updates. We investigate the throughput and response time of search and update operations. The throughput is the rate at which operations could be served by the system (i.e., numbers of operations per second). The response time is the time interval between issuing an operation and obtaining the response from the system when the task has been successfully completed.

Figure 3.26 shows the throughputs and response times for the three indexes for varying numbers of threads. The throughputs of the B^{x}-trees are the highest (up to 10 times than those of the TPR*-tree), and the response times of the B^{x}-trees are always least among all the indexes. There are two reasons. First, the average query and update cost of the B^{x}-tree are less than those of the the TPR*-tree and the B^{dual}-tree. Second, the B^{x}-trees seldom lock internal nodes. Recall that in the query processing, we will first travel down to the leaf level, then retrieve the leaf nodes for the answers by following the left-to-right sibling links. We may occasionally ascend to an internal node for a "jump," but this often happens at the lower levels of the index. Also, in the update processing, the B^{x}-trees only need to travel down one path. In contrast, in the TPR*-tree, deletions and queries lead to searching along multiple paths, which introduces more frequent locks on the internal



(a) Throughput (b) Response time

Figure 3.26: Effect of Concurrent Operations

nodes. Further, the TPR*-tree may hold locks longer due to its complicated update algorithms. All these factors reduce the parallelism of concurrency operations on the TPR*-tree. For the B^{dual}-tree, its query algorithm is quite similar to that of the TPR*-tree and hence it shares the similar problems.

3.3.8 Storage Requirements

Index size is an important issue in moving object databases since a small index size may enable the caching of the entire index in main memory in order to improve performance. Figure 3.27 shows the storage requirement of the indexes, in which the B^x-trees require the least storage space. For the 1M dataset in particular, the storage space of the TPR*-tree is more than three times larger than that of the B^x-trees. And the storage space of the B^{dual} tree is also larger than that of the B^x-tree. The main reason is that the B^x-trees have relatively large node capacity (332) and high utilization of this capacity (about 73%). For the TPR*-tree, the capacity is about 200. For the B^{dual}-tree, the capacity is a little less than that of the B^x-tree due to the longer key value obtained from the higher-dimensional

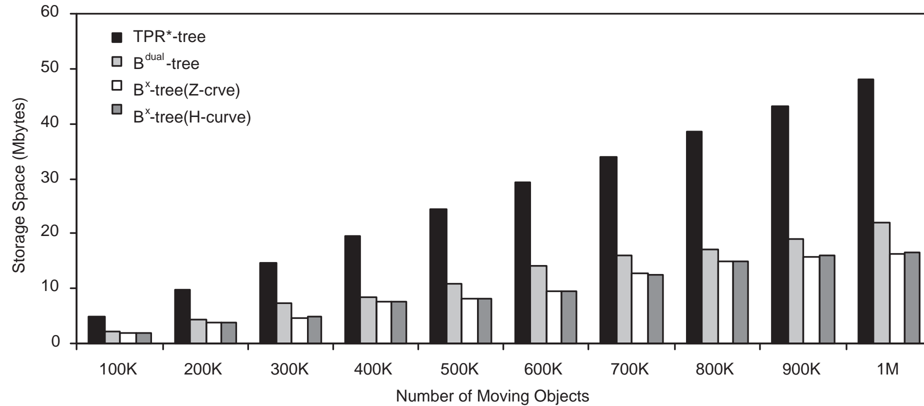


Figure 3.27: Storage Requirement

space-filling curves.

3.4 Summary

Database applications that entail the storage of samples of continuous, multidimensional variables pose new challenges to database technology. This chapter addresses the challenge of providing support for indexing that is efficient for querying as well as update.

We propose a new indexing scheme, the B^x-tree, which is based on the B^+-tree. This scheme uses a new linearization technique that exploits the volatility of the data values being indexed, i.e. moving-object locations. Specifically, data values are first partitioned according to their update time and then linearized within the partitions according to a space-filling curve, e.g. the Peano or Hilbert curve. Algorithms are provided for interval range queries and k nearest neighbor queries on the current or near-future positions of the indexed objects. Queries that reach into the future are handled via the query region enlargement, as opposed to the MBR enlargement used in TPR*-trees.

Our extensive performance studies indicate that the B^x-tree is both efficient

and robust. In fact, it outperforms the TPR*-tree, especially when it comes to update operations. Further, being a B^+-tree index, the B^x-tree may be more easily integrated into existing database systems than its competitors.

CHAPTER 4

Effective Density Queries on Moving Objects

Continuing advances in consumer electronics, mobile communications, and positioning technologies combine to render it increasingly realistic to assume that entire populations of users of mobile services, i.e. moving objects, can be tracked accurately. These developments offer a foundation for the delivery of increasingly sophisticated location-enabled mobile services. Motivated by this scenario, one line of research aims to develop appropriate data management foundations like indexing techniques presented in the previous chapter. The other line of research aims to provide efficient query services with the aid of data management systems. In this chapter, we will study a novel type of location-based service, the querying for dense regions, termed *density query*.

The rest of the chapter is organized as follows. Section 4.1 states our motivation. Section 4.2 gives the problem statement. Section 4.3 describes our proposed frame-

work. Section 4.4 presents the algorithm used to realize the framework. Section 4.5 reports the experimental results. Finally, Section 4.6 makes a conclusion.

4.1 Motivation

The objective of the density query is to find regions in space along with associated points in time where the regions have a density that exceeds a given threshold. Figure 4.1 illustrates an example where three square-shaped windows compose the

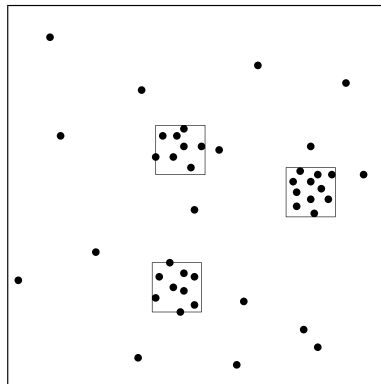


Figure 4.1: An Example of Density Query Results

answer to a density query. The density query may have applications in a range of areas. For example, in traffic management systems, density queries may be used for identifying regions with potential for congestion and traffic jams.

Density queries for moving objects was first considered by Hadjieleftheriou et al. [29]. They define the *Region Density* as: $density(R, \Delta t) = \min_{\Delta t} N/\text{area}(R)$, where $\min_{\Delta t} N$ is the minimum number of objects inside R at any time during Δt and $\text{area}(R)$ is the area of R. They define the *Period Density Query* as: given N moving objects, a horizon H, and thresholds α_1, α_2, and ρ, find regions $R = \{r_1, ..., r_k\}$ and associated maximal time intervals $\Delta t = \{\delta t_1, ..., \delta t_k | \delta t_i \subset [t_{now}, t_{now} + H]\}$ such that $\alpha_1 \leq \text{area}(r_i) \leq \alpha_2$ and $density(r_i, \delta t_i) > \rho$ (where t_{now}

is the current time, $i \in [1, k]$, and k is the query answer cardinality).

For the applications we envision, finding dense regions *for a period of time* appears to be less useful than simply finding dense regions for a point in time. For example, once a traffic jam occurs in some region, all the objects around the region will slow down, and their velocities will change dramatically. As a result, predicting the density at following timestamps according to the original velocities reported for the objects does not seem to be of much value. Therefore, we focus on the identification of dense regions as of a timestamp $t_q \in [t_{now}, t_{now} + H]$ that is given as a parameter to the density query.

Hadjieleftheriou et al. [29] find the general density-based queries difficult to answer efficiently and hence turn to simplified queries. Specifically, they partition the data space into disjoint cells, and the simplified density query report cells, instead of arbitrary regions, that satisfy the query conditions. This scheme may result in what we term *answer loss*. Consider the example shown in Figure 4.2 where each cell (of solid lines) is a unit square and the density threshold ρ is 3. There actually exists a dense region (the dashed square) in the center of the space, but the simplified query reports no regions. In our density query definition, we guarantee that there is no answer loss.

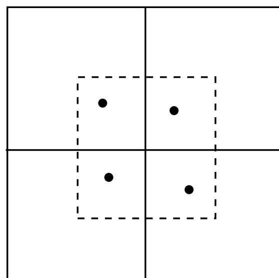


Figure 4.2: An Example of Answer Loss

We proceed to formulate the problem setting and define the notions of *density*, *dense region*, and *effective density query*.

4.2 Problem Statement

We assume that a population of moving objects exists, where each object is capable of transmitting its current location to a central server. A moving object transmits a new location to the server when the deviation between its real location and its server-side location exceeds a threshold, dictated by the services to be supported. In general, the deviation between the real location and the location assumed by the server tends to increase as time passes. In keeping with this, we define a *maximum update time* (U) as a problem parameter. This quantity denotes the maximum time duration in-between two updates of the position of any moving object.

We model the position of a moving object as a linear function from time points to points in two-dimensional Euclidean space. The position of a moving object at time t, $\bar{x}(t)$, is thus given by a triple $(\bar{x}, \bar{v}, t_{upd})$ of parameters as follows: $\bar{x}(t) = \bar{x} + \bar{v}(t - t_{upd})$, where $\bar{x}$ and $\bar{v}$ are the two-dimensional position and velocity, respectively, of the object at the latest update time t_{upd} and $t \geq t_{upd}$.

The motivation for this choice of modeling is threefold. First, the extent of a moving object is typically considered to be of little relevance for the query type we are considering. Second, studies of real positional information obtained from GPS receivers installed in cars show that representing positions as linear functions of time reduces the numbers of updates needed to maintain a reasonable accuracy by as much as a factor of three in comparison to using constant functions [17]. Linear functions are thus much better than constant functions. Third, several indexing techniques exist that index this representation and which can be reused.

With this general data setting in place, we proceed to define the density query.

Definition 2 (**Density**)*: The density of a region R at a time t is the number of objects in the region at time t divided by the area of the region.*

Definition 3 (**Dense Region**)*: A region is dense at time t if its density at time t is higher than a density threshold ρ.*

Note that a part of space that contains one dense region is likely to contain many such regions. Most of these may overlap substantially, as illustrated in Figure 4.3(a). Reporting all such regions is not helpful. We proceed to propose an *effective density query* that only reports non-overlapping regions. The resulting answer set clearly identifies regions of high density, as shown in Figure 4.3(b).

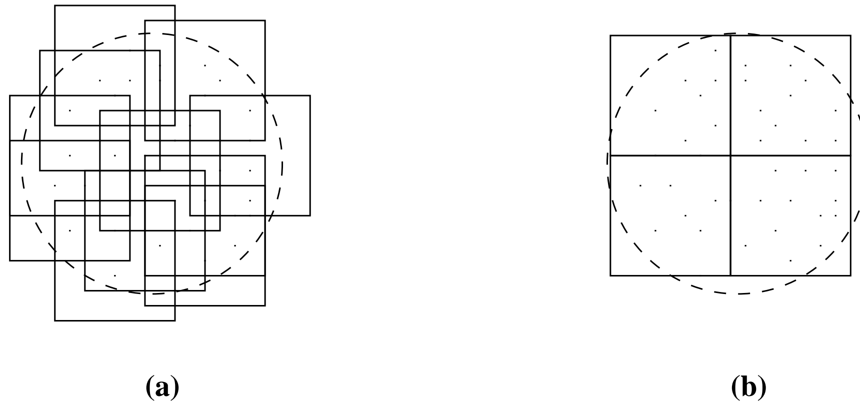


Figure 4.3: Overlapping vs. Non-overlapping Regions in a Density Query

Definition 4 (**Effective Density Query**)*: Find all dense regions at time t that satisfy the following conditions:*

1. *Any reported region is constrained to a certain shape and an area range.*
2. *No two regions in the result overlap.*
3. *Any dense region in the argument data is in the result, or is represented in the result by a region that overlaps with it.*

The first condition provides mechanisms for ensuring that meaningful answers are reported. For example, an arbitrarily small region that contains one point object is infinitely dense, but makes little sense as a result. Therefore, we may want to

specify a lower bound of the area of the result regions. Similarly, we may want to limit the region area so that it is not too large. Also not all shapes of result regions may be desirable. Imagine a (space filling) curve that goes through all the point objects. Therefore the user may require the dense regions to be squares, circles, etc. The second condition guarantees that a non-redundant result is produced. The third condition guarantees that the query result contains evidence of any dense region in the data, ensuring that query results do not suffer from answer loss.

We purposefully define the effective density query so that different, but equally valid and useful, results may be produced for the same data argument and query parameters. An algorithm implementing the query may exploit this flexibility. For convenience, we abbreviate the term *effective density query* to *density query* in the sequel.

We assume that the time parameter t_q of a density query Q is not earlier than the current time and that it only reaches at most W time units into the future. Thus, with $iss(Q)$ being the time that query Q is issued, $iss(Q) \leq t_q \leq iss(Q) + W$. The lower bound indicates that we are not considering past data. The assumption that an upper bound exists is considered reasonable. Updates are inherently frequent, making it meaningless to look too far into the future.

Finally, define *time horizon* $H = U + W$ as the maximum duration of time that the representation of a moving object can be queried. Figure 4.4 illustrates the relationships among parameters U, W and H. We can see that H represents how

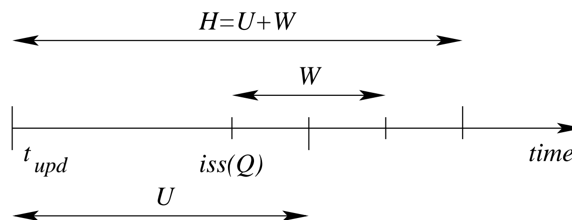


Figure 4.4: Problem Parameters

far into the future a query may reach. In other words, a technique for computing the density query support queries that reach up to H time units from the update of an object into the future.

4.3 The MODQ Framework

As a precursor to considering the processing of density queries, we constrain the general setting for density queries as presented in Section 4.2 with the objective of rendering the query processing manageable. We term this setting the Moving Object Density Query (MODQ) framework.

In this framework, we constrain the dense regions to be square-shaped and of certain sizes. We maintain all moving objects in an index structure. During the query processing, we first partition the data space into equal-sized, square-shaped cells of the smallest size that satisfies the query constraint. Then we issue range queries that explore these cells, and we report the dense regions found.

Note that a dense region is not necessarily a cell in the partitioning—it might instead intersect with cell partitions, as does the dashed, square region in Figure 4.2. Therefore we may need to issue a range query that is defined by two or more adjacent cells. By exploring all cells (and maybe all combinations of two or more adjacent cells), we are able to report evidence of all dense regions.

While the framework applies to dense regions of a range of sizes, we focus on finding dense regions of one size in this paper.

4.4 Density Computation

4.4.1 Overview

If we process the density query straightforwardly using range queries on the index, we are likely to end up issuing too many range queries. Instead, we propose a two-phase algorithm that efficiently computes the density query as stated in the previous section.

The algorithm relies on certain information that needs to be maintained. We maintain a counter that records the number of objects for each cell at point in time. As each object is updated, we calculate the trajectory of the object and obtain the cells it intersects during the query time range $[t_{now}, t_{now} + H]$. For each $(\text{cell}, \text{time})$ pair of a cell intersected and the time of intersection, we increase the corresponding counter by one (in case of deletion, we decrease the counter).

At the same time, the object is maintained and updated in an index structure. This index may be well maintained already for other types of queries, such as range and nearest neighbor queries, so the major space overhead is that of the space for maintaining the counters. When the numbers of cells and time windows are large, the number of counters needed is huge. We describe a method to efficiently maintain them in a compressed fashion in Section 4.4.2.

Next, query processing consists of two phases:

1. **The filtering phase:** We use the counters to quickly prune the cells that are surely not in the answer set and produce a set of candidate cells for the next phase.
2. **The refinement phase:** To extract the final answers from the candidate cells obtained in the filtering phase, we issue range queries corresponding to the cells on the index and this way determines the actual positions of the

objects in the cells. Then we can determine the dense regions. Without loss of generality, we exploit the B^x-tree (in Chapter 3) to maintain the moving objects.

The query processing algorithm is given in Section 4.4.3.

4.4.2 Density Histogram

We maintain a two-dimensional *density histogram* (DH) equal sized, square cells, where each cell contains a counter of the number of objects in the cell at all times in $[t_{now}\ , t_{now} + H]$. The DH is the main structure used for the filtering phase. As we need to maintain a histogram for a long time period for each cell, the total memory use may be prohibitively large. Moreover, if the DH is too big, it may need to be stored on disk, and hence substantial I/O is needed to maintain it and use it during the query processing. Therefore, it is critical to reduce the size of the DH. We propose to use the Discrete Cosine Transform (DCT) [72] for compression. The detailed algorithm and analysis are as follows.

Histogram Construction

For each cell, the number of objects in the cell varies across time, but the number is not likely to change greatly for adjacent time points. We thus view the time-varying count in each cell for a time range as a signal $s(t)$, and we then perform the Discrete Cosine Transform (DCT) on $s(t)$. As the DCT is a good approximation of the Karhunen-Loève Transform (KLT) [72], the first few components of the DCT of s carry the major information in s. We store only the first few (typically 10–20%) components and discard the rest. We can restore the $s(t)$ by an inverse DCT when we maintain the histograms or use them for the query processing.

The DCT of a signal $s(t)$ of length H is also a signal $G(k)$, of length H. The transform is defined as follows:

$$G(k) = c(k) \sum_{t=0}^{H-1} s(t) \cos \frac{\pi(2t+1)k}{2H} \qquad \text{where} \tag{4.1}$$

$$c(0) = \sqrt{1/H}, \quad c(k) = \sqrt{2/H}, \text{ and } k = 0, 1, ..., (H-1)$$

The inverse DCT is defined as follows:

$$s(t) = \sum_{k=0}^{H-1} c(k) G(k) \cos \frac{\pi(2t+1)k}{2H}, t = 0, 1, ..., (H-1) \tag{4.2}$$

Figure 4.5 shows an example of transform between the signal and the DCT.

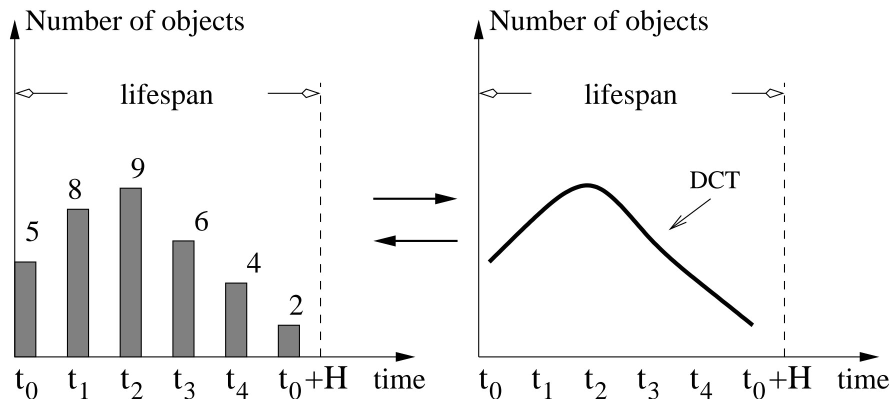


Figure 4.5: An Example of the DCT

After trimming some components of the DCT, the restored signal $s'(t)$ differs from $s(t)$ because of the information loss. Therefore, the results of the query is approximate. However, the DCT has the good property that even though we trim off a great portion of the components, the difference between $s'(t)$ and $s(t)$ is still quite small.

Note that the difference between $s'(t)$ and $s(t)$ may be positive or negative, that is, $s'(t)$ may overestimate or underestimate $s(t)$. As a result, we may get

both false positives and false negatives when we choose candidate cells for further examination in the refinement phase. False positive candidate cells increase the query processing cost, while false negatives would cause answer loss (although the loss could be very small). To avoid false negatives, we may add something to $s'(t)$ so that $s'(t)$ is guaranteed to overestimate $s(t)$. We derive the bound of $s(t) - s'(t)$ next. We assume that we use the first g coefficients of the DCT to compress and trim the remaining ones.

$$\begin{aligned} s(t) - s'(t) = & \sum_{k=g}^{H-1} c(k)G(k)\cos\frac{\pi(2t+1)k}{2H} \\ = & \sqrt{2/H}\sum_{k=g}^{H-1} G(k)\cos\frac{\pi(2t+1)k}{2H} \\ \leq & \sqrt{2/H}\sum_{k=g}^{H-1} |\max_{k=g}^{H-1}\{G(k)\}\cos\frac{\pi(2t+1)k}{2H}| \\ \leq & \sqrt{2/H}\max_{k=g}^{H-1}\{|G(k)|\}\sum_{k=g}^{H-1}|\cos\frac{\pi(2t+1)k}{2H}| \end{aligned} \tag{4.3}$$

where $t = 0, 1, ..., (H-1)$ and $\max_{k=g}^{H-1}\{|G(k)|\}$ denotes the maximum absolute value of $G(k)$ for $k = g, ..., H-1$. Therefore if we estimate $s(t)$ by $s'(t)$, the *error bound* E_b is given as follows.

$$E_b = \sqrt{2/H}\max_{k=g}^{H-1}\{|G(k)|\}\sum_{k=g}^{H-1}|\cos\frac{\pi(2t+1)k}{2H}| \tag{4.4}$$

Before trimming the coefficients from the DCT of $s(t)$, we get $\max_{k=g}^{H-1}\{|G(k)|$ and store it with the g remaining coefficients; and $\sum_{k=g}^{H-1}|\cos(\pi(2t+1)k/2H)|$ can be calculated on the fly, giving us the error bound. To guarantee no false negatives, we just need to add the error bound to $s'(t)$. However, this increases the number of false positives and hence increases the query processing cost. In some applications,

we may be willing to trade a small number of false negatives for better performance.

To capture the degree to which we are willing to tolerate false negatives, we introduce the parameter *error factor* $e_f \in [0, 1]$ that can be specified by the user. We then estimate $s(t)$ by $s'(t) + e_f \cdot E_b$. When $e_f = 1$, we guarantee no false negatives. As e_f decreases, the probability of false negatives increases, and when $e_f = 0$, we estimate $s(t)$ by $s'(t)$. In the experiments reported in Section 4.5, there are no false negatives in most of the cases, even when $e_f = 0$.

Histogram Maintenance

A location update contains the old and new information of a moving object, including the position, velocity and the time when these apply. When such an update is received, we compute both the old and new trajectories of the object. Then we adjust the DCT functions in the cells that the moving object passes by.

The adjustment comprises three steps. First, we unwrap the DCT function, i.e., we compute the number of moving objects during each time point within the lifespan of the function. The second step treats deletion and insertion differently. For a deletion, we simply decrease the number of moving objects by one during the period that the old trajectory intersects with the cell and then modify the start time of the lifespan of the function to the current time. For an insertion, we set the lifespan from the current time t_{now} to $t_{now} + H$, and initialize the number of moving objects exceeding the old period to zero. Then we increase the number of moving objects during the intersection period by one. Third, we calculate new DCT functions for the affected cells.

Figure 4.6 summarizes the maintenance algorithm. First, note that deletion and insertion may affect the same cell if the new trajectory does not deviate much from the old one. In such a situation, we only unwrap and recompute the DCT

Algorithm DH_maintenance$(P_o(x, v, t), P_n(x, v, t))$
Input: P_o and P_n are the old and new object, respectively

1. compute the trajectory of P_o during $[P_o.t, P_o.t + H]$
2. $L_o \leftarrow$ list of cells intersected by P_o
3. compute trajectory of P_n during $[P_n.t, P_n.t + H]$
4. $L_n \leftarrow$ list of cells passed by P_n
5. $L \leftarrow L_o \bigcup L_n$
6. **for** each cell in L **do**
7. set the start time of lifespan to current time
8. $V \leftarrow$ value list of DCT function in its lifespan
9. **if** there is a deletion in this cell **then**
10. decrease corresponding value in V by one
11. **if** there is an insertion in this cell **then**
12. set the end time of lifespan to $P_n.t + H$
13. extend V to $P_n.t + H$, adding value 0
14. increase the corresponding value in V by one
15. compute new DCT function from V

end DH_maintenance.

Figure 4.6: DH Maintenance Algorithm

functions of the cell once, since we do the deletion and insertion together in the second step. Second, the lifespan we maintain is no longer than H, either in the deletion or in the insertion. This makes it possible to use the same number of parameters to represent the DCT function.

To exemplify, Figure 4.7(a) depicts an original DCT function of a cell before updates, and Figures 4.7(b) and (c) illustrate an independent deletion and an insertion in this cell. As shown in Figure 4.7(b), an object intersecting the cell during time t_2 to t_3 is deleted, and hence, counters at corresponding time points are decreased by one. Figure 4.7(c) shows an insertion of an object at time t_1.

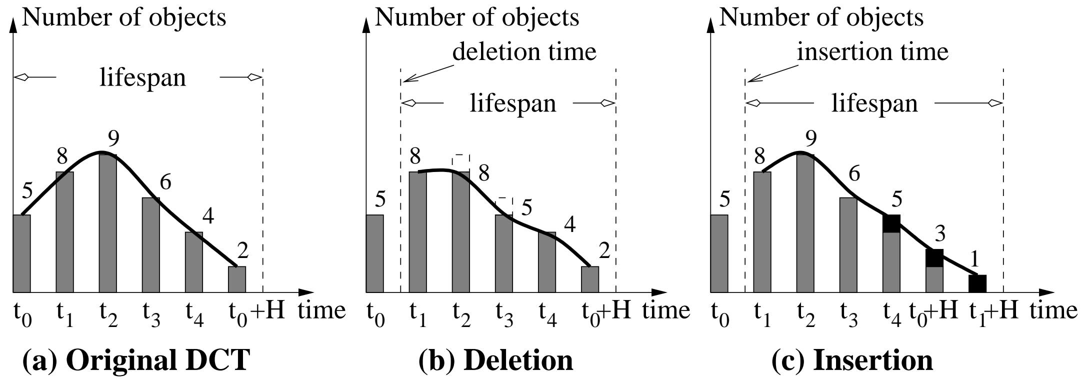


Figure 4.7: Maintenance in DH

The lifespan of the function is then changed to $[t_1, t_1 + H]$. The trajectory of this object intersects with the cell from time t_4 to $t_1 + H$, and hence the corresponding counters are increased by one.

4.4.3 Query Processing

The Filtering Phase

The filtering phase aims to identify areas that may possibly contain answers to the density query. The output of this step is a list of grid cells of sizes one to four times larger than the query range size. This is because the dense squares may intersect with one to four cells, as shown in Figure 4.8 (the shaded areas represent dense squares).

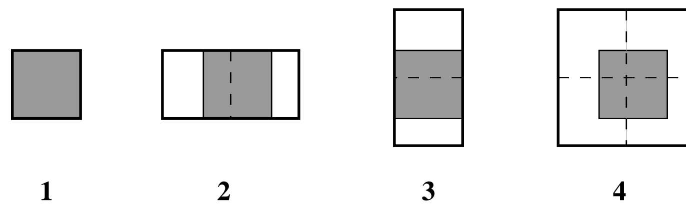


Figure 4.8: Intersection between the Final Answer and DH Cells

We examine the cells of the DH in the order from left to right and top to bottom. The algorithm is shown in Figure 4.9.

Algorithm Density_query(ρ, R, t_q)
Input: threshold ρ, query range R, query time t_q

```
1.  N_min ← R · ρ
2.  for each cell in the space do
3.      N_b ← number of objects in the cell at t_q
4.      if N_b > N_min then
5.          report this cell as a final answer
6.      else
7.          N_s ← number of objects in square S_4        // S_4 consists of four cells
8.          if N_s ≥ N_min then
9.              flag ← true
10.             for each combination of two cells S_2 do
11.                 N_2 ← the density in the two cells
12.                 if N_2 ≥ N_min then
13.                     invoke Refinement(S_2, ρ, R, t_q)
14.                     if an answer is found then
15.                         modify histogram
16.                         flag ← false
17.             if flag then
18.                 invoke Refinement(S_4, ρ, R, t_q)
19.                 if an answer is found then
20.                     modify histogram
end Density_query.
```

Figure 4.9: Density Query Algorithm

Given a query range R and a query threshold ρ, we have $N_{min} = R \cdot \rho$, which is the minimum number of objects that should occupy a dense square. This way, we transform the density threshold ρ to the number of objects N_{min}. Then, for each cell, we compute the number of objects it contains at the query time t_q. If it contains at least N_{min} objects, it is added to the final answer list directly. Otherwise, we check the *square* consisting of four cells and having the current cell

at the top left corner. If this square has less than N_{min} objects, it is obvious that this square does not contain any dense square. We can safely prune the current cell and set the tags of combinations of any cells in this square to false (i.e., we need not take into account these combinations next time).

If the number of objects in the square is larger than N_{min}, we check the density of each cell in this square and report those cells satisfying the density threshold themselves. In the remaining cells, we check the cells of types 2 and 3 as shown in Figure 4.8. If the number of objects in the combinations is no smaller than N_{min}, we pass them to the refinement phase. Only when all three types of cells fail to contain any final answer, we pass the whole square to the refinement phase. Each time we get an answer from the refinement phase, we decrease the number of objects in the corresponding cells.

To avoid overlaps among final reported ranges, the area covered by the answer is tagged and will not be considered during the following search. We also adopt heuristics to help speed up the processing. As shown in Figure 4.10, in one square, cells of types 2 and 3 are not allowed to coexist; the fourth type of cell is not allowed

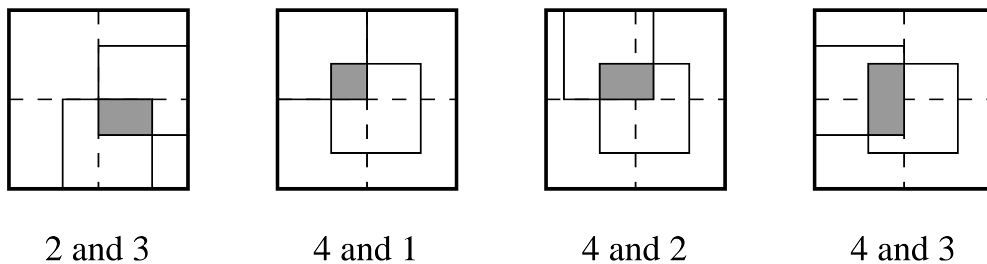


Figure 4.10: Conflicting Types of Cells

to coexist with any other type of cell. Once an answer of one type is confirmed, we do not need to search the conflicting type.

The Refinement Phase

We introduce a new structure, which we term an *object pool*, that temporarily stores information for the objects in retrieved cells at the query time.

The refinement phase needs to obtain the objects in each candidate area. The algorithm first checks whether any part of a given candidate area has ever been retrieved. If this is the case, we load the objects from the object pool and tag this part. Then a general range query covering the untagged parts of the candidate area is issued on the index. We store the newly retrieved objects in the object pool.

After obtaining objects, algorithms that differ only slightly are applied to the different types of cells in order to identify final answers. There are cells of types 2, 3, and 4.

For cells of type 2, we sort the positions of the objects according to their x coordinates. Then we count the number of objects every l length units (l is the query range size) along the x axis until the count reaches or exceeds the threshold N_{min}, which means we have identified an answer. We handle the cells of type 3 similarly to those of type 2, except that we sort the object positions along the y axis this time. For type 4 cells, we first sort the object positions along the x axis. When we count the number of objects every l length units along the x axis, we maintain an array that stores the number of objects along the y axis. As the starting point of the counting moves forward, we decrease the corresponding values in the array. Once we obtain a count that reaches or exceeds N_{min}, we will check along the y axis as in the case of type 2.

Objects in the top-left cell are discarded from the object pool since this cell may not be accessed any more according to our scan order. Therefore, the object pool only needs to store up to a row of cells. Moreover, each time we identify an answer, we remove the objects in the answer set from the object pool.

The detailed algorithm is shown in Figure 4.11.

Algorithm Refinement(S, ρ, R, t)

Input: candidate area S, density threshold ρ, query range R, query time t

1. $N_{min} \leftarrow R \cdot \rho$, $S_r \leftarrow \phi$, $L_1 \leftarrow \phi$
2. **for** each cell B in S **do**
3.   **if** the cell B has been retrieved **then**
4.   load objects from object pool to L_1
5. $S_r \leftarrow S_r \bigcup B$
6. $L_2 \leftarrow$ RangeQuery$(S - S_r, t)$
7. $L \leftarrow L_1 \bigcup L_2$
8. $l \leftarrow \sqrt{R}$
9. **if** S is of type 2 or 4 **then**
10.   sort objects in L along x-axis
11.   project objects to x-axis
12. **else**
13.   sort objects in L along y-axis
14.   project objects to y-axis
15. $N \leftarrow$ the number of objects within each l length
16. **if** any N larger than N_{min} **then**
17.   **if** S is not of type 4 **then**
18.   report an answer
19.   **else**
20.   project objects to y-axis
21. $M \leftarrow$ number of objects within each l length
22.   **if** any M larger than N_{min} **then**
23.   report an answer

end Refinement.

Figure 4.11: Refinement Algorithm

4.5 Performance Studies

In this section, we present the results of an extensive performance study on our proposed technique.

4.5.1 Experimental Settings

All the experiments were run on a 2.6G Pentium IV desktop with 1 Gbyte of memory. The page size is 4K.

We employ the B^x-tree (H-curve) (presented in Chapter 3) as the index for the refinement phase. An LRU page buffer of 50 pages is used [48], with the internal nodes of a tree being pinned in the buffer.

The space domain is 1000 $\times$ 1000 units. The datasets are generated using an existing data generator, where objects move in a network of two-way routes that connect a given number of uniformly distributed destinations [76]. Objects start at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Whenever an object reaches one of the destinations, it chooses the next target destination at random. Objects accelerate as they leave a destination, and they decelerate as they approach a destination. In most experiments, the average interval between two successive updates of an object equals 60 time units. Unless noted otherwise, the number of moving objects is 100,000.

The query workload is 100 density queries. Each query has three parameters: (i) the density threshold ρ; (ii) the squared-shaped query range side length l; and (iii) the prediction lengths q_l. The query cost is measured in terms of CPU time and I/O.

The parameters used are summarized in Table 4.1, where values in bold denote

the default values used.

Parameter	Setting
Page size	4K
Buffer number	50
Max update interval	**60**, 120, 240
Density threshold	0.8, 0.9, **1.0**, 1.1, 1.2
Max prediction length	30
Density query range size	20, **25**, 50
Number of density queries	100
Dataset size	**100K**, ..., 1M
Number of destinations in dataset	**50**, 100, 200, 300

Table 4.1: Parameters and Their Settings

4.5.2 DCT Compression Accuracy

First, we look at a representative result of the DCT compression of the histograms. We execute 100 density queries with query range size of 25 in a 100K dataset. Figure 4.12 shows the actual numbers of objects ($s(t)$) and the restored numbers of objects from the DCT compression ($s'(t)$, which we term the DCT compression in the sequel) in cells of density 0.8 and 1.2, respectively, as a function of elapsed time. We see that the curves for the predicted number of objects match the curves

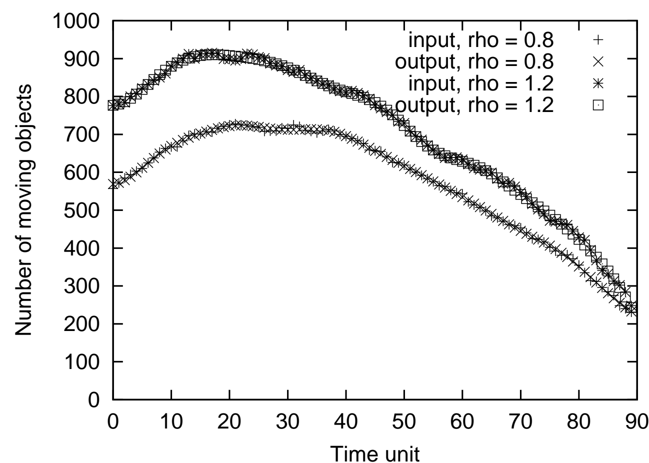


Figure 4.12: DCT Compression Accuracy

for the actual number of objects very well.

Next, we evaluate the accuracy of the DCT compression using two metrics, the rates of false positives and false negatives, while varying the number of DCT coefficients used, elapsed time, and the error factor e_f. The *rate of false positives* indicates the percentage of squares in the answer set that have lower density than the given threshold, and the *rate of false negatives* is the percentage of squares that are missing in the answer set to the total number of correct answers. The correct answers are obtained by using the histograms without compression by the DCT.

Effect of the Number of DCT Coefficients

First, we investigate the effect of the number of DCT coefficients. We create a density histogram of 100K data at time 0, and then issue 100 square density queries of size 25, prediction length in the range $[0, 30]$, and density threshold equal to 0.8, 1.0, and 1.2, respectively.

Figure 4.13 shows the rates of false positives and negatives for varying numbers of DCT coefficients. As expected, the more DCT coefficients we used, the better

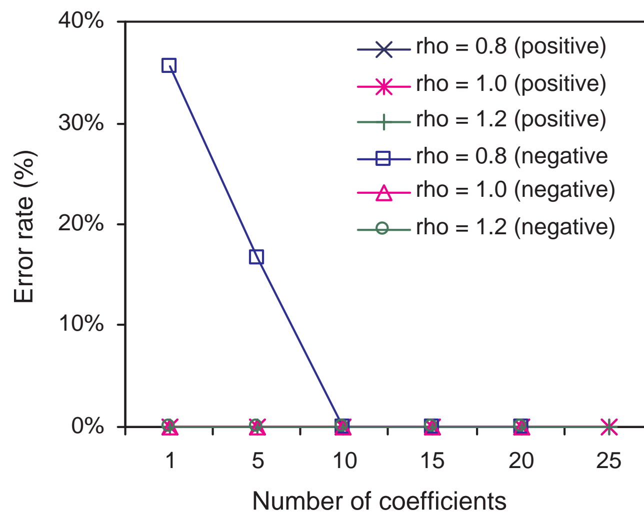


Figure 4.13: False Positives and Negatives for Varying DCT Coefficients

the accuracies of the results. Both types of errors virtually disappear when the number of DCT coefficients exceeds 10. The results suggest that our method is highly accurate while saving about 90% of the space (the original DCT has 90 coefficients).

In addition, the DCT functions work well when the density threshold is large. When the density threshold is close to the average density, both types of errors increase since a small deviation in the DCT function may wrongly report or prune many squares.

Effect of Time

We use 20 DCT coefficients in the following experiments. The density histogram and the index of 100K objects are created at time 0 and are then maintained until time 240. To avoid frequent transformations between the DCT and the real data, we employ the batch update technique where each batch contains 1,000 updates. After each maximum update interval (60 time units), density queries with the same parameters as in the previous experiment are issued. Figure 4.14 plots the false

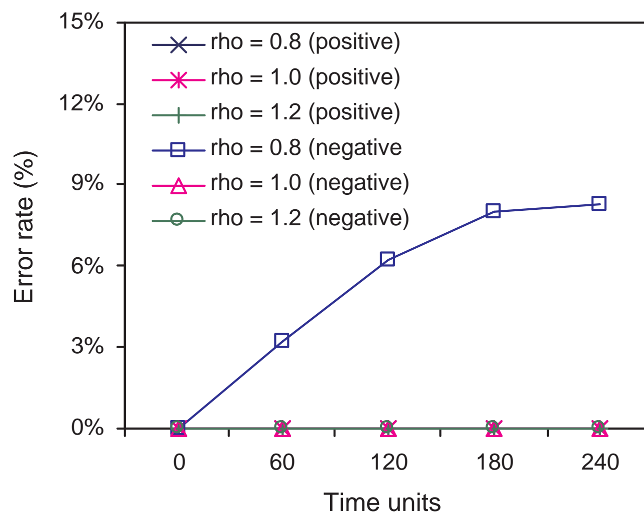


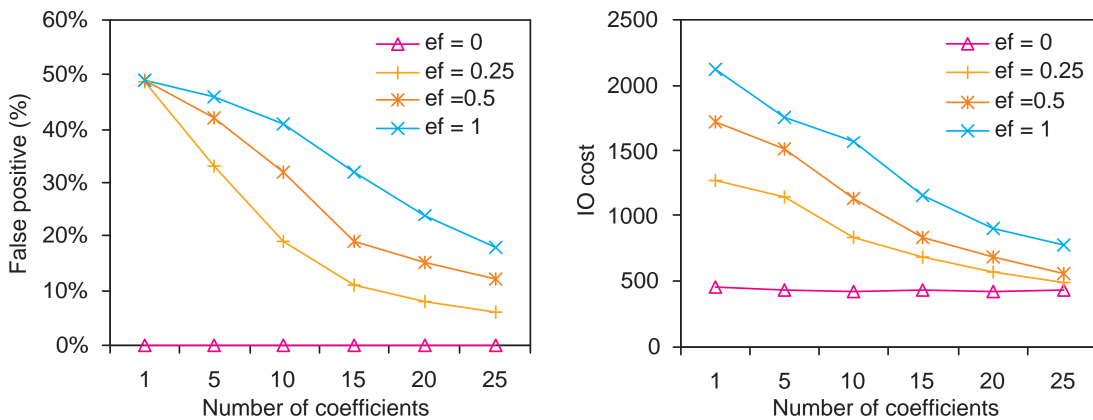
Figure 4.14: False Positives and Negatives with Elapsed Time

positives and negatives.

We observe that the false positive decrease to 0% as time passes, while the false negatives approach 10%. This is because the DCT compression underestimates the real number, resulting in fewer false positives, but also missing answers. Moreover, new coefficients are computed based on existing ones, leading to an increasing underestimation.

Effect of the Error Factor

In order to avoid false negatives, we can add a positive value to the restored information. This experiment examines the effect of overestimating the restored. Figure 4.15(a) and (b) show the false positives and the query I/O cost, respectively, for varying error factors and numbers of DCT coefficients.



(a) False positives (b) Query I/O cost

Figure 4.15: Effect of the Error Factor and DCT Coefficients

The false positives decrease as the number of DCT coefficients increases—with more coefficients, the DCT becomes more accurate. We also observe that when the error factor increases, the false positives also increase. This behavior is expected since the error factor indicates how much we overestimate the number of objects. When we guarantee no false negatives, that is $e_f = 1$, the false positives

are at about 50% when using only one DCT coefficient, but decrease to about 20% when additional coefficients are used (but still significantly smaller than the total number).

The actual false negatives for these experiments are at 0%. This means that we may need to overestimate to guarantee no false negatives, although the actual numbers of false negatives are very low, even when we do not overestimate at all. The query I/O cost shown in Figure 4.15(b) exhibits a similar trend to that for the false positive. This is because the more the false positives, the more times we need to search the index, which increases the I/O cost. Similarly, there is a tradeoff between the error factor and query I/O. When we use a smaller error factor, that is, larger probability of false negatives, we obtain better query performance. In the following experiments, we always keep the false negatives to 0 and then evaluate the query and maintenance performance.

4.5.3 Density Queries

We proceed to evaluate the efficiency of the density query processing algorithm while varying different parameters. We start by showing an example of the results of a density query.

An Example of the Density Query

Figure 4.16(a) shows a snapshot of the dataset with 50 destinations in the simulated road network. Given a density query with query range size 25 and density threshold 1.0, Figure 4.16(b) shows the density query result (denoted by squares) obtained. We can clearly see that the algorithm identifies all the dense regions.

(a) Data space

(b) Dense squares

Figure 4.16: Density Query Example

Histogram versus Non-Histogram

This experiment evaluates the pruning effectiveness of the DH. We compare our method, which uses the DH, with a straightforward method as described in Section 4.3, which uses the MODQ framework and the same refinement algorithm as in our histogram-based algorithm, but does not use histograms and the two-phase query algorithm (we simply call it the non-histogram algorithm). To locate the region of the required density, the non-histogram algorithm executes a series of range queries covering the whole space, where each range query covers a square consisting of four cells. As the cost of refinement phase dominates the overall performance, we mainly compare their I/O costs of executing range queries.

Figure 4.17 shows the average number of I/O operations per density query for varying density thresholds. Our algorithm improves over the non-histogram at a factor of 4 in terms of I/O cost.

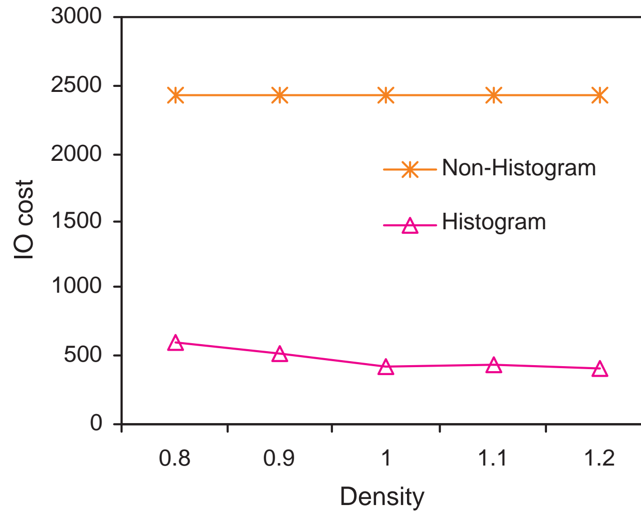


Figure 4.17: Histogram vs. Non-histogram

The MODQ versus the DCF

We also compare our algorithm with the *dense cell filter* (DCF) algorithm [29], which exhibits the best performance among other algorithms. Due to the different definitions of the density query, the DCF is not able to identify dense squares across cells. Figure 4.18 shows the percentages of lost answers for the DCF and our algorithm. We can see that the number of lost answers of the DCF increases quickly as the density threshold increases. This is because the result set size decreases as the density threshold increases, with the effect that there are relatively more lost answers. As expected, our algorithm has no answer loss.

Effect of Density Threshold and Query Size

Next, we investigate the effect of the density threshold and the query range size. Since the I/O cost incured in the refinement phase always dominates the cost of a query, in order to study the behavior of the filter and refinement algorithms independently on the index structure, we partition the density query cost into the

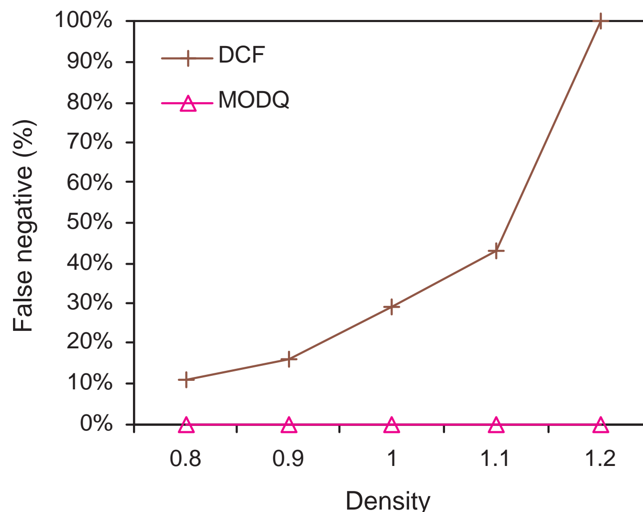
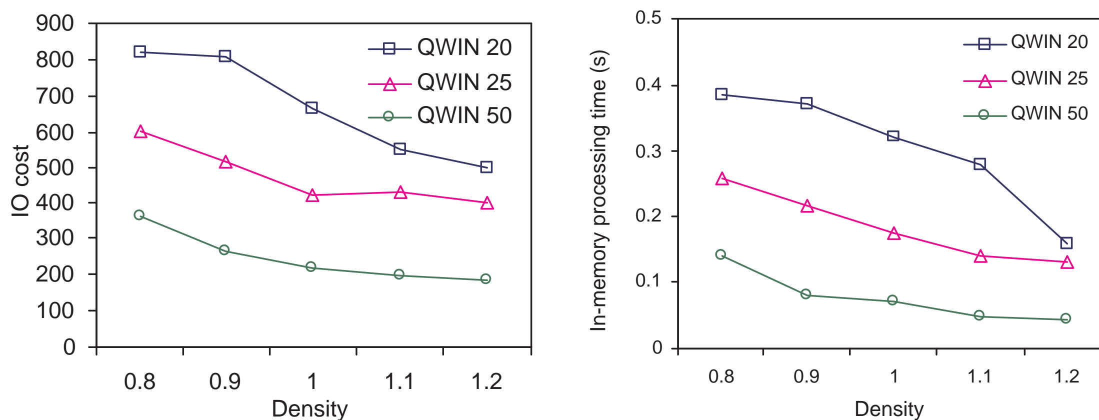


Figure 4.18: The MODQ vs. the DCF

range query cost (I/Os) and in-memory processing cost (CPU time), and then plot them separately.

Figure 4.19(a) and (b) measure the density query performance in the same 100K dataset for varying density thresholds and query range sizes. Figure 4.19(a) shows



(a) Query I/O cost

(b) In-memory processing time

Figure 4.19: Effect of Density Threshold and Query Size

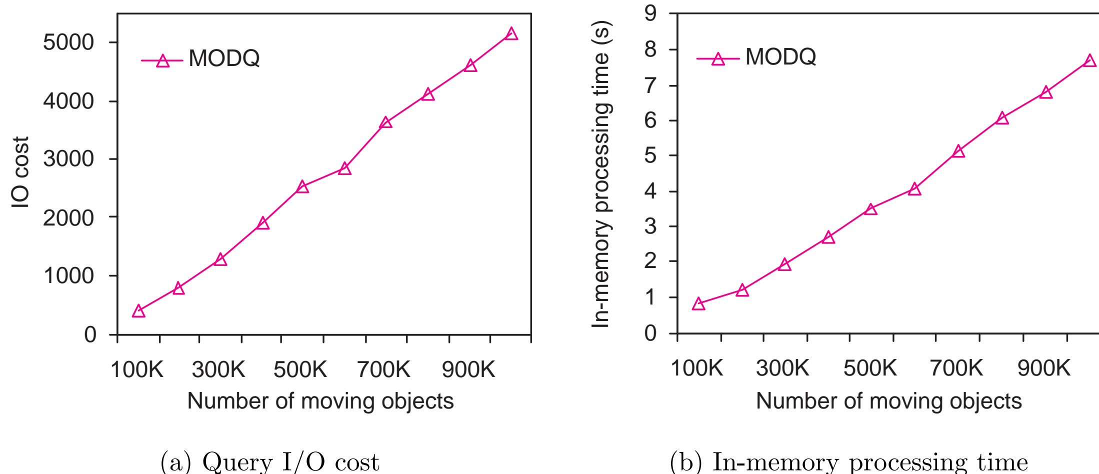
the range query I/O cost per density query. The I/O cost decreases as the density threshold increases. The possible reason is as follows.

When the density threshold is close to the average density (about 0.5 for a query range size of 25), the pruning ability of the filtering phase decreases. This is so because most regions of the data space have average density. Even if the density of one cell is lower than the threshold, its combinations with other cells still have high probabilities of satisfying the density threshold. Hence, many range queries are issued in the refinement phase, which results in higher query cost. When the density threshold is high, the filtering phase can prune cells that have few objects. Thus, the total range query cost is smaller. Moreover, we observe that the query cost decreases as the query range becomes larger. This is because the number of answers is smaller for larger query ranges; hence, fewer range queries are issued in the refinement phase.

Figure 4.19(b) shows the corresponding in-memory processing cost. The trends are similar to those seen for the range query I/O cost. The reasons for this behavior are similar to those given for the I/O cost.

Effect of Database Size

To test the scalability of our technique, we performed the density query while varying the number of moving objects from 100K to 1M. We fix the density threshold at 1.0 and use a query range size of 25. Figure 4.20(a) and (b) show the range query I/O cost and the in-memory processing cost per density query, respectively. We observe that both the I/O cost and the in-memory processing cost grow linearly as the number of moving objects increases. This is because the average density in each cell increases as the number of moving objects increases. More regions that satisfy the density requirements need to be checked.



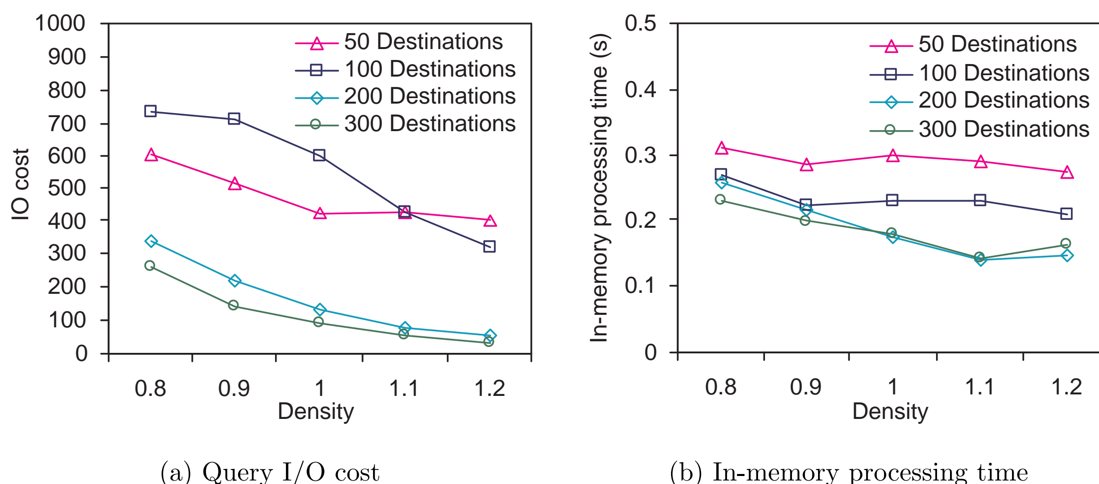
(a) Query I/O cost

(b) In-memory processing time

Figure 4.20: Effect of Database Size

Effect of Data Distribution

In this experiment, we evaluate the density query performance for different data distributions by using numbers of destinations in the simulated route network in the range from 50 to 300. The fewer the destinations, the more skewed the dataset becomes. Figure 4.21(a) and (b) plot the query I/O cost and in-memory processing cost, respectively. We observe that the query costs of the 50- and 100-destination datasets are higher than those of the 200- and 300-destination datasets. This is



(a) Query I/O cost

(b) In-memory processing time

Figure 4.21: Effect of Data Distribution

because skewed data tend to result in high density in more regions so that more queries in the index are needed. In addition, it is interesting to see that the density query cost of the 100-destination dataset is sometimes higher than that of the 50-destination dataset. This behavior can be explained as follows. Places around the destinations may have high densities since moving objects usually assemble there. In the 100-destination dataset, the densities at these places are possibly close to the given density threshold. Thus, there are more possible candidate dense cells that need to be further checked in the refinement phase, which causes the higher query cost.

4.5.4 Maintenance Cost

Finally, we evaluate the maintenance cost of our histogram-based algorithm while varying the length of the maximum update interval U (the query prediction length is fixed at 30). We create the index at time 0 and then perform object updates for the duration of a maximum update interval. Figure 4.22 shows the average maintenance cost per insertion or deletion. We observe that the average maintenance

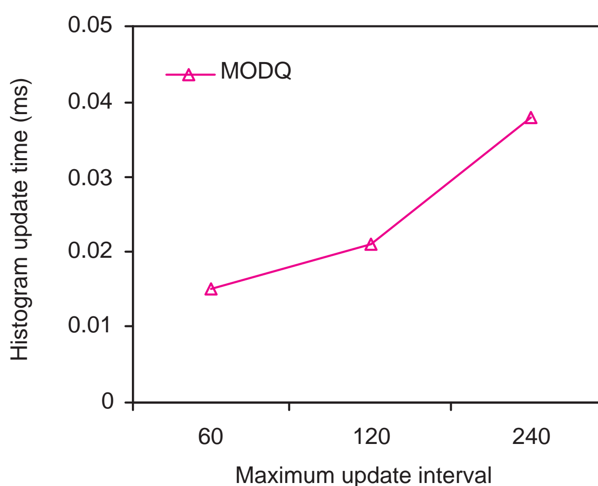


Figure 4.22: Maintenance Cost

cost increases as the maximum update interval increases. The reason is that each update of an object incurs updates on the cells that its trajectory intersects. As the maximum update interval increases, the length of the trajectory increases, and therefore the maintenance cost also increases.

4.6 Summary

In this chapter, we give a pragmatic definition of the density query for moving objects that avoids answer loss. Based on this definition, we proposes a specialization of the density query that returns useful answers and is amenable to efficient computation.

We then propose an algorithm that aims to process the resulting density query efficiently. The algorithm utilizes temporal histograms of counters for each partition in a partitioning of the data space. These histograms help prune the majority of regions in an initial filtering phase of the query processing. As the histograms consume substantial storage space, which in turn increases the I/O cost, techniques are proposed that use the Discrete Cosine transform (DCT) to compress the histograms. This compression incurs very few errors in the answer set, but offers space savings of up to 90%, which also reduces the I/O cost.

We also report on extensive empirical studies of the behavior of the proposed algorithm. The results suggest that the algorithm offer an improvement of a factor of 4 in terms of I/O, compared to a naive algorithm. The results also indicate that although we reduce the storage usage greatly by using the DCT, the answers are still highly accurate. It is shown that it is easy to trade a small number of false negative answers for performance.

CHAPTER 5

Location Privacy in Moving-Object Environments

The expanding use of location-based services has profound implications on the privacy of personal information. If no adequate protection is adopted, information about movements of specific individuals could be disclosed to unauthorized subjects or organizations, thus resulting in privacy breaches. In this chapter, we propose a framework for preserving location privacy in moving-object environments.

The rest of the chapter is organized as follows. Section 5.1 gives a synopsis of our proposal. Section 5.2 describes our system architecture. Section 5.3 presents the detailed algorithms for location updates and queries. Section 5.4 presents the system analysis. Section 5.5 covers comprehensive performance experiments. Finally, Section 5.6 concludes.

5.1 Synopsis of Our Proposal

Today people are increasingly aware of privacy issues and do not want to expose their personal information to unauthorized subjects or organizations. An important problem is represented by the possibility that a piece of personal information released by an individual to a party be combined by this party, or other parties, with other information, leading to the disclosure of sensitive personal information. In other cases, even if an individual does not directly release personal information to another party, this party may still become aware of this information if it has to provide a service to such an individual. This is in particular the case of location-based service providers that, because of the very nature of the services they provide, need to track user movements and locations. It is then easy, based on this information, to discover user habits and other personal information. There is therefore an important concern for *location privacy* in location-based services, that is: "how can we prevent other parties from learning one's current or past location? [11]". By looking more closely at the privacy problem in such a context, we can see that there are at least two important requirements, that is, keeping movement and location information private from service providers and from other users. For example, GPS users who do not want to disclose their locations to the system may still require service such as "is there any of my friends close to me now?" There are two privacy requirements for this query. First, service providers are not allowed to know the real locations of users. Second, users can only query an authorized dataset (e.g. a list of their friends).

In this chapter, we address such a problem by developing a framework to preserve location privacy in moving-object environments. The basic idea of our approach is to send transformed user location data to the service provider. We support a number of different types of transformations, such as scaling, translation, rota-

tion, to cloak user information. These transformations are performed by agents interposed between users and service providers. Agents are only responsible for transforming information either received from the users or the server. They serve as intermediaries and do not store user information. The service providers receive the transformed data and compute answers to queries on these transformed data.

An important feature of our approach, which is critical for privacy assurance, is the use of multiple agents. The user can randomly choose the agent to receive his information each time he issues an update. Thus, each agent only has a part of the information concerning the user. Such an approach is crucial for enhancing privacy. For example, if an adversary hacks one agent, it is still unable to track the user; if some agents illegally store user information, they cannot determine the trajectories of users without colluding with other entities. Here our approach closely adheres to an important security principle, dictating that sensitive information should not be entrusted to a single entity; rather such information should be spread among several entities.

In our framework, the server stores for each agent a sub-dataset specific to the agent. A query is thus executed by the server separately on the sub-dataset of each agent. Note that location-based queries require that relative distance among users through the same agent be maintained after the transformation. The transformations we adopt have such a property. Specifically, we employ a combination of the three basic types of transformations, that is, scaling, rotation, translation, as our transformation functions. It is however important to notice that maintaining the relative distance after the transformation may reveal the map topology. Therefore, we introduce the concept of *multiple transformation* that applies slightly different transformation functions to users' positions updated at different time instants. This makes the relative distance hard to be inferred. Correspondingly, the multiple

transformation also needs to be applied to queries. To avoid handling the increased number of queries, a *super query* is then proposed, which covers all queries after the multiple transformation. As explained later, a super query is essentially an approximate version of the original query, which facilitates efficient evaluation of this framework with a small amount of filtering costs.

Our technique not only prevents service providers from inferring the exact locations of users, but also keeps information about the location of an individual private from other individuals not authorized to access such information. Specifically, users have a list of group IDs that indicate which groups they belong to. Based on these group IDs, the server can remove the query answers that are not in the qualified groups, so that users can avoid their privacy leaked to other users not belonging to the same group. A key characteristic of our approach is that privacy is achieved without degrading service quality. Based on the experiments that we have carried out, our approach does not affect any update performance, which is very important in moving-object environments where update frequency is always high.

Finally, we develop metrics to measure the level of privacy achieved by our framework. In particular, we investigate the threats posted by the agents and the query server from discovering the users' true locations and movement patterns. We then propose intuitive methods to quantify the level of protection against these threats in our system.

Before go into details of our techniques, we summarize the difference between our work and most related approaches in the following subsection.

5.1.1 Comparison to Existing Approaches

Cheng et al. [13] adopt the idea of location cloaking, which represents a user location by a region. Their approach cannot provide accurate query answers, whereas accuracy is guaranteed by our approach.

Gruteser et al. [26] propose a quad-tree based algorithm to partition the space so that each sub-space contains more than k objects, i.e., satisfies the anonymization requirement. The main drawback of this method is that it fails to meet the anonymization criterion for skew data. Our approach does not have any constraints on the data distribution.

k-anonymity model has also been used in [24, 39, 52]. They all utilize a centralized anonymizer which collects all user information. Such an anonymizer may become the target of attacks by malicious parties. In our approach, we impose multiple agents in-between users and service providers and hence reduce the risk of compromising a single point. Moreover, unlike existing approaches where agents must store users' locations in order to carry out post-processing, agents in our proposed system do not need to store any user information which further reduce the chance of information leak.

Most recently, Ghinita et al. [25] propose a distributed architecture for anonymous location-based queries. They did not use any agent. Instead, they treat each user as a peer which is eligible to become an anonymizer for other users. Their approach certainly avoids the bottleneck caused by centralized techniques both in terms of anonymization and location updates. However, the proposed system poses more responsibilities and burdens to users. A user selected as a cluster head needs to work as a server, which may not be feasible if users are only equipped with lightweight devices. Also, the system requires users to trust one another in the same cluster. This is different from creating trust between users and agents where

agents can be well verified. If a user does not want to trust any other users, he/she will not subscribe to the location-based services provided by the proposed system.

In addition, there is a common problem of the approaches discussed above. They may not be able to support anonymizations around sensitive areas such as home addresses in non-anonymous applications. For example, if a user's ID is known, the cloaking region around his home address will tell attackers that the user is probably at his home. However, in our system, this type of attack will be shown to be a very hard task.

5.2 The Strategies and the Architecture of the Location Privacy Protection System

In this section, we describe the strategies and the architecture of our Location Privacy Protection (LPP) system. Figure 5.1 illustrates this architecture. The basic

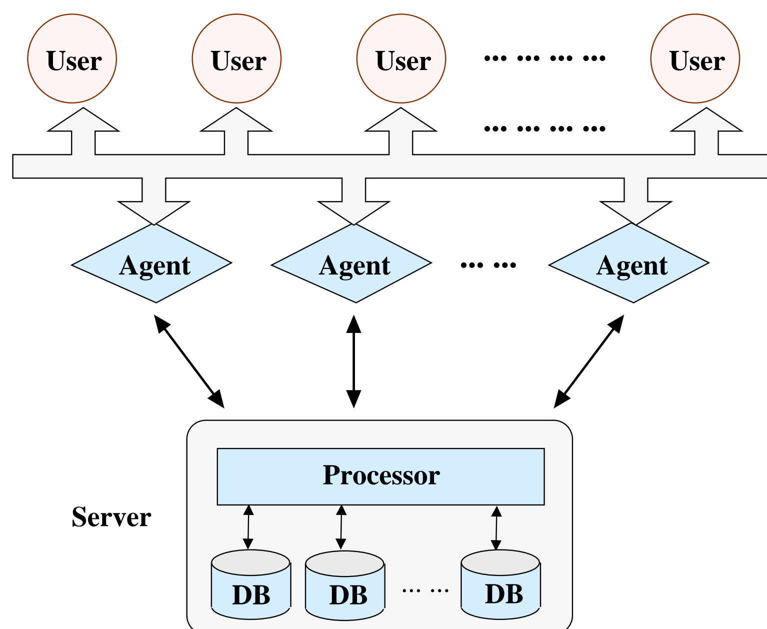


Figure 5.1: LPP System Overview

strategy underlying our approach is to reduce the leaking of private information by using data transformation and employing m agents in-between users and servers. Each time a user[1] needs to update his position, he does not directly contact the server; instead, he *randomly* selects an agent to which he sends his data. When querying, the user has to send the query to all agents. Then the agents will execute a transformation on the user data or queries and pass the transformed data to the server. The server handles the data processing and returns the query results to the agents. After receiving the results from the server, the agents perform a reverse transformation before returning the results to the user. We now proceed to describe in details how each component of our system works.

- **User**

  Users are position providers or query issuers. Users' positions are assumed to be unchanged until next update, that is, the *location database* at the service provider keeps the latest position of each user. Users may have a list of qualified agents, and they are assumed to have the ability to randomly choose agents and perform some postprocessing.

  Different policies can be adopted to protect information about a given user from other users. One policy is a global ranking, which allows users with high ranks to query location and movement information about users with equal or lower ranks. Another policy is a group policy, under which users can query location and movement information about users in the same group. A user can be a member of multiple groups and hence he may have a list of group IDs. In our system, we adopt the latter policy. Hence, in location databases, users are represented by records of the form $\langle uid, gids, loc, t_{up} \rangle$, where uid is

[1] We use the term 'user' in the discussion. In reality the described activities are carried out by some client software residing at the user's device.

the user ID, *gids* is a list of group IDs, and *loc* is the user's location at time t_{up}.

- **Agent**

  Agents are a critical component in our approach. An agent transforms the data received from the users and sends them to the server. It also executes reverse transformations on the data obtained from the server and then forwards them to the users.

  The types of transformations supported by an agent includes the transformation of the user ID, the group IDs, and the user locations. Agents periodically change their transformation functions in order to prevent the server from analyzing the data from the same agent. Thus, agents need to maintain transformation tables for each type of data. Such tables store records of the form $\langle t_{id}, f_{id}, count_{id} \rangle$ and $\langle t_{loc}, f_{loc}, count_{loc} \rangle$, where t records the time instant at which the transformation function f has started to be used, and *count* is the number of objects being transformed by f.

  There are three important features about our agents. First, for the security purpose, agents are independent of the main server, which means they are not under control of the server. Second, agents do not store any user data and hence they are lightweight computers. Therefore, it is possible to verify their code in order to provide assurances about their correct behavior. Third, transformation functions for different types of data do not need to be changed at the same time.

- **Server**

  The server is responsible for data storage, maintenance and query processing. It also maintains datasets transferred by various agents separately. Any index

for moving objects that supports efficient updates and queries can be adopted to manage the datasets in the server.

The main advantage of our approach is that no single entity (m agents or server) is able to track the movement of any user without colluding with other entities in the system. Because each agent only collects a subset of the locations of each user in the system, the level of trust required from each agent does not need to be high. Moreover, the use of m agents allows multiple transformations to be applied to the data by the same user. This makes it much harder for the server to keep track of the relative distance among users. In essence, the server is only a computing engine for the various agents.

Finally, we would like to mention that we focus on queries over moving objects in this study. For queries over static objects (e.g. restaurants, gas stations), our framework can be extended in the following way. We can store the static objects in a separate database in the server since such objects may not have any concern over location privacy, and then we use slightly modified query algorithms (which will be explained later). Unless specified otherwise, we assume the data of interest are moving objects in subsequent discussions.

5.3 Algorithms

In this section, we present the detailed algorithms for data transformation, queries and updates in the LPP system.

5.3.1 Data Transformation

Data transformation includes transformation of user IDs, group IDs, user locations, and queries. We address each of them respectively in the following sections.

ID Transformation

The main purpose of ID transformation is two-fold. First, we need to prevent the server from identifying the same users through different agents. This can be easily achieved by choosing different transformation functions for different agents. There are no restrictions on the transformation function itself. It could be a simple encryption. Also, we need to prevent the server from tracking the positions of the same user from one agent. We thus propose to periodically change the transformation functions for each agent, which can assign different pseudo-IDs to the same user who sends data at different time instants. A transformation table is then maintained for each agent. As mentioned previously, the transformation table consists of records of the form $\langle t_{id}, f_{id}, count_{id} \rangle$. Algorithms for its maintenance are covered in section 5.3.2.

Location Transformation

Just transforming IDs is not enough to provide location privacy for users because some locations (e.g. homes) are strongly associated with user IDs and may thus cause information leak. Therefore, we introduce the notion of location transformation, which is a crucial feature of our system.

The main challenge in the development of suitable functions for location transformation is to keep the relative distance in each sub-dataset (the dataset obtained from the same agent) unaltered by the transformation in order to support location based services (e.g. nearest neighbor queries). Possible transformation functions include scaling, rotating, translation, and their combinations. In our system, we employ a combination of scaling, rotation and translation. We represent the transformation function through its parameters denoted by the tuple $[s, \theta, (t_x, t_y)]$, where s is the scaling factor, θ is the rotation angle, and t_x, t_y are the translation distance

along the x and y axes respectively.

However, the preservation of the relative distance among objects could disclose the map topology. For example, if the server tries to connect objects close to one another, it may be able to discover the joint distribution of objects and then determine the road network. Figure 5.2 gives a simple example. Suppose that the

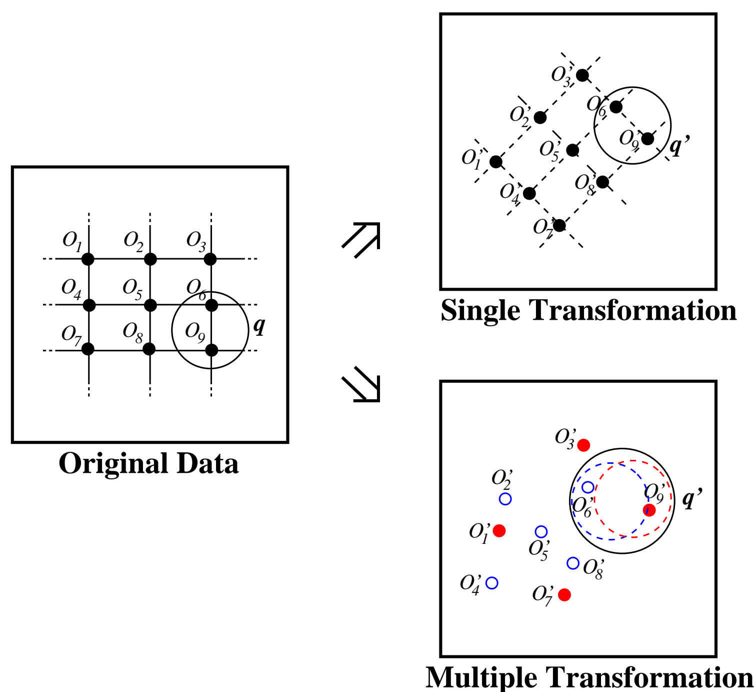


Figure 5.2: An Example of Position Transformation

original data lie on a grid-like road network. If they are transformed by a single transformation function, the server may discover the grid by connecting objects on the same lines (dashed lines in the figure). To address such a problem, our approach is to make the relative distance hard to be inferred. We thus adopt a strategy that requires each agent to periodically change the location transformation function. We refer to such strategy as *multiple transformation*. The bottom-right part of Figure 5.2 shows the effect of the multiple transformation strategy. Assume that objects O_1, O_3, O_7 and O_9 are transformed by a function f_1; O_2, O_4, O_5, O_6 and

O_8 are transformed by another function f_2 that is only a little bit different from f_1. From the transformed objects, it is hard to discover the original data distribution.

We now proceed to present the generation of the multiple transformation. The first transformation function can be an arbitrary one, while the following transformation functions need to fulfill some constraints. The differences among the transformed positions obtained by various transformation functions should be kept within a small range. Such a constraint is crucial in order to provide good quality answers to queries based on the relative distance among objects.

A simple strategy to satisfy the above constraints is to apply the translation operations with different parameters to the first transformation function. Moreover, to achieve efficient queries, multiple transformation should preserve the following property.

Property 1 *Let $\langle x, y \rangle$ be a point, $\langle x_0, y_0 \rangle$ be the position obtained by applying the initial transformation function to $\langle x, y \rangle$, and $\langle x_1, y_1 \rangle$, $\langle x_2, y_2 \rangle$,..., $\langle x_{n-1}, y_{n-1} \rangle$ be the positions obtained from subsequent multiple transformation functions. The distance between $\langle x_0, y_0 \rangle$ and $\langle x_i, y_i \rangle$ ($1 \leq i \leq n-1$) must be less than or equal to a threshold λ.*

The detailed algorithm for multiple transformation is summarized in Figure 5.3. The first step selects an initial transformation function $[s_0,\ \theta_0,\ (t_{x0},\ t_{y0})]$, sets its counter $count_0$ to 0, and stores the values in the transformation table. After a period of time t_{int}, we generate a new transformation function. We first randomly choose a value d in $(0, \lambda)$, and then randomly generate the parameter d_x (the translation distance of x axis) in the range of $(-d, d)$. The parameter for the y axis d_y can be computed by $d_y = \pm(d^2 - d_x^2)^{\frac{1}{2}}$. Then we insert a new tuple $\langle t_1, [s, \theta, (t_{x0}+d_x, t_{y0}+d_y)], 0 \rangle$ in the transformation table. This process is repeated every t_{int} time interval. There are two things worth noting. First, each agent

---

Algorithm Multiple_Transformation($Ttable$**,** t_c**)**
Input: $Ttable$ is a transformation table, t_c is current time

1. **if** $(t_c = 0)$ **then**
   // select the first transformation function
2. randomly generate s_0, θ_0, t_{x0}, t_{y0}
3. insert $\langle 0, [s_0, \theta_0, (t_{x0}, t_{y0})], 0\rangle$ into $Ttable$
4. **else**
5. randomly generate d in the range of $(0, \lambda)$
6. randomly generate d_x in the range of $(-d, d)$
7. randomly select d_y from $\{-(d^2 - d_x^2)^{\frac{1}{2}}, (d^2 - d_x^2)^{\frac{1}{2}}\}$
8. insert $\langle t_c, [s_0, \theta_0, (t_{x0} + d_x, t_{y0} + d_y)], 0\rangle$ to $Ttable$

end Multiple_Transformation.

Figure 5.3: Multiple Transformation Generation Algorithm

can choose his own λ. Second, the transformation table will not keep growing. Functions that are no longer used by users will be removed during the update operations (as addressed in Section 5.3.2).

Query Transformation

We now address how to transform queries. In the discussion, we focus on range queries. A range query retrieves all objects the location of which falls within the circular range $q = (c(x, y), r)$ at time t_q not prior to the current time, where $c(x, y)$ is the center and r is the radius of the query.

Due to the multiple transformation on the users' positions, a query has to handle data from different transformations. One solution is to transform the query using all transformation functions, and then execute multiple queries. However, this is not efficient and may disclose the relationship among transformation functions. Therefore, we introduce the concept of *super query*, which covers all queries after

multiple transformations. For example, in Figure 5.2, a range query q is first transformed into two queries (represented in the figure as dashed circles) by function f_1 and f_2. Instead of answering these two queries, we propose answering a super query q' that covers the regions of these two queries. In this case, the query efficiency mainly depends on the extra area covered by the super query. In the following, we first describe how to generate the super query, and then analyze the characteristics of the super query.

Given a query $q = (c(x, y), r)$, we can obtain a set of transformed queries by using the multiple transformation functions. Since the transformation functions change with time, to compute a super query that tightly bounds all transformed queries requires the checking of all the transformation functions and thus involves extensive computations. We propose to use an easily-computed super query (denoted as q_s) which is always a superset of the transformed queries unless the parameter λ changes. Specifically, q_s is computed as: $q_s = (c(f_0(x)\ ,\ f_0(y)),\ f_0(r) + \lambda)$, where f_0 is the first (initial) transformation function. Figure 5.4 illustrates an example, where the black point is the transformed query center by using the first transformation function, white circles are positions after other transformations, and the transformed radius of the query is r'.

The generation of the easily-computed super query is based on Property 1 (see

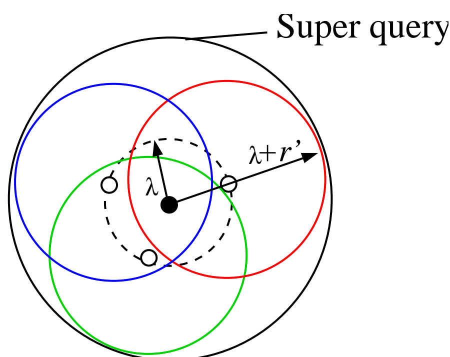


Figure 5.4: Super Query

previous section). Property 1 prevents the super query from growing arbitrarily large. It guarantees that the radius of the super range query is at most λ larger than that of any transformed query. It is true that the super query may incur some overhead due to the search of a larger space compared to the query transformed by any one of the transformation functions. To characterize the super query, we define its *false negative rate* as the number of missing query answers divided by the number of correct query answers, and define its *false positive rate* as the number of false query answers divided by the number of correct query answers. Esstimates for false positive and false negative rates are established by the following theorem.

Theorem 1 *Let* $q = (c, r)$ *be a query, and* f_0, f_1, ..., f_{n-1} *be a set of transformation functions, where* f_0 *is the initial transformation function. Its super query* $q_s = (c_s, r_s)$ *satisfies the following properties*

(i) false negative rate fn *is 0;*

(ii) false positive rate fp *is approximately* $2\lambda/f_i(r)$ $(0 \leq i \leq n-1)$.

Proof 2 *We denote the query transformed by* f_i *as* $q_i = (c_i, r_i)$ $(0 \leq i \leq n-1)$. *We denote the correct answer set as* A.

(i) To prove the false negative rate is 0, we need to prove that for any $a \in A$, a *can be captured by* q_s.

We know that a *is transformed by one of the transformation functions, say* f_i $(0 \leq i \leq n-1)$*. Then,* a *can be captured by the query* q_i *which is transformed by the same transformation function.*

According to Property 1, the distance between the centers of q_i *and* q_0 *(transformed by* f_0*) is less than* λ*. According to the generation algorithm of the super query, the center of* q_0 *is the same as the center of* q_s*, and the radius of the* q_s *is* λ *more than that of the* q_i*. Consequently, we have* $r_s - r_i \geq distance(c_s, c_i)$*, which indicates that* q_s *covers* q_i*. Hence* a *can be captured by* q_s.

(ii) Assume the data points are evenly distributed, then we may use the areas to see how more points can be covered by the super query compared with the query by a single transformation (i.e., the number of false positives is proportional to the extra area).

The area S_i *covered by a query* q_i *is* πr_i^2 *. The area* S_s *covered by the super query is* $\pi(r_i + \lambda)^2$*. Then the percentage of increase in the area of the super query is:*

$$fp = \frac{S_s - S_i}{S_i} = \frac{\pi(r_i+\lambda)^2 - \pi r_i^2}{\pi r_i^2} = \frac{\lambda(2r_i+\lambda)}{r_i^2}$$

When $\lambda \ll r_i$*,* $fp \simeq 2\lambda/r_i$*.* $\square$

Theorem 1 demonstrates the correctness of the super query (no false negatives) and points out a way to tune the performance of the query. Given a false positive rate, we can choose a proper λ.

Note that from the users' point of view, there will be no false positive because the agent will filter the data rerturned by the server in order to eliminate the false positives.

5.3.2 Updates

Generally, an update is interpreted as a deletion followed by an insertion. Figure 5.5 shows the detailed update algorithm.

To insert a tuple $\langle uid, gids, loc, t_{up}\rangle$ of a user, three steps are executed. First, the user randomly selects an agent and sends his information to the agent. Second, the agent transforms the user ID, the group ID list and the location, and then sends the transformed data to the server. During the transformation, the agents will adjust the counters of the transformation functions, and remove the ones with counters equal to 0 which will not be used in the future. Finally, the server tags the data with the agent ID and stores them.

Algorithm Update

User:

Insertion:

1. randomly select an agent with ID aid
2. send $\langle uid, gids, loc, t_{up},' i' \rangle$ to the agent aid

Deletion:

1. send $\langle uid, gids, loc, t_{up},' d' \rangle$ to the agent aid

Agent:

1. receive $\langle uid, gids, loc, t_{up}, op \rangle$ from the user
2. $f_{id} \leftarrow$ ID transformation function of time t_{up}
3. $(uid', gids') \leftarrow f_{id}(uid, gids)$
4. $f_{loc} \leftarrow$ location transformation function of time t_{up}
5. $loc' \leftarrow f_{loc}(loc)$
6. send $\langle uid', gids', loc', t_{up}, op, aid \rangle$ to the server
7. **if** $(op ==' i')$ **then** // this is an insertion
8. $\quad count_{id} \leftarrow count_{id} + 1$
9. $\quad count_{loc} \leftarrow count_{loc} + 1$
10. **else** // this is a deletion
11. $\quad count_{id} \leftarrow count_{id} - 1$
12. $\quad count_{loc} \leftarrow count_{loc} - 1$
13. $\quad$ **if** ($count_{loc}$ is 0 and f_{loc} is not 1st function)
14. $\qquad$ delete the tuple of f_{loc} from transformation table
15. invoke Multiple_Transformation every t_{int}

Server:

1. receive $\langle uid', gids', loc', t_{up}, op, aid \rangle$ from the agent
2. **if** $(op ==' i')$ **then** // this is an insertion
3. $\quad$ insert $\langle uid', gids', loc', t_{up}, aid \rangle$
4. **else** // this is a deletion
5. $\quad$ delete $\langle\ uid',\ gids',\ loc',\ t_{up},\ aid\ \rangle$

Figure 5.5: Update Algorithm

For the deletion, the user needs to submit his old information to the same agent which handled the insertion of this information. The agent will check the transformation table and look for the corresponding function at the update time. Then, the agent will use this function to transform user information, and decrease the counter of this function by one. If the counter is 0, the function (except for the first one) will be removed from the transformation table. The remaining process for deletion is similar to the insertion.

It is worth noticing that users can send deletion message to the old agents and insertion message to the new agents.

Consider the example shown in Figure 5.6. Suppose there are four users O_1, O_2, O_3 and O_4, and two agents A_1 and A_2. O_1 and O_3 select agent A_1, and O_2 and O_4 select agent A_2. The transformed data of O_1 and O_3 is O'_1 and O'_3, and the transformed data of O_2 and O_4 is O'_2 and O'_4, respectively.

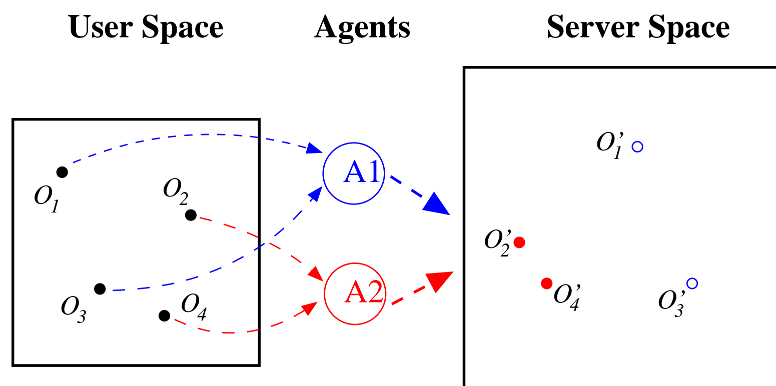


Figure 5.6: An Example of Update Operation

We also consider the situation when an object disappears accidentally without being able to notify the server. The information of such objects will soon be outdated. We define that an object is outdated if difference between its latest update time and current time is larger than a given threshold. During each insertion or deletion, we identify and delete outdated entries in accessed nodes.

5.3.3 Queries

Our model supports various types of queries, such as range queries and k nearest neighbor queries. In the following, we outline the query execution strategies.

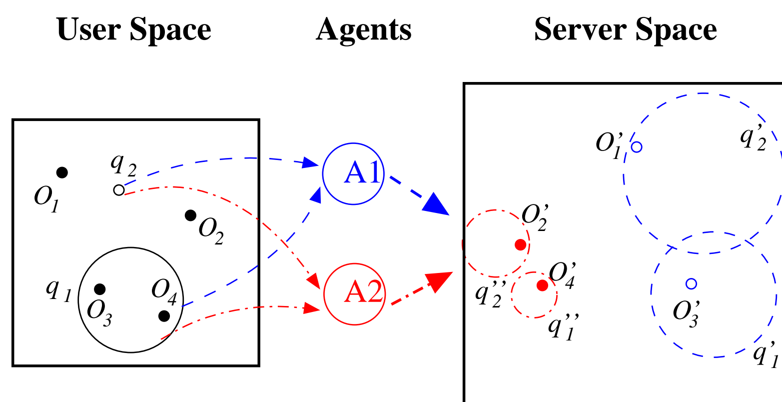


Figure 5.7: An Example of Query Operation

Range Query

A range query retrieves all objects whose location falls within the circular range $q = (c(x, y), r)$ at time t_q not prior to the current time, where $c(x, y)$ is the center and r is the radius of the query.

As object positions are transformed in different ways through different agents, we have to send a query to all agents. Each agent will generate and send a super query to the server. After receiving the query answers from the server, the agent needs to transform them back and to check whether they are the correct answers to the original query. Finally, users will aggregate the partial results obtained from the agents. If user ranks or group IDs are to be taken into by the query, one more filtering step will be carried out by the server in order to prune unqualified answers. Note that the server can filter the results based on transformed IDs before sending any results to agents. Figure 5.8 shows the outline of the algorithm.

Algorithm Range_Query

User:

1. **for** $(i \leftarrow 0)$ **to** $(i < m)$ **do**
2.   send $\langle uid, gids, c(x, y), r\rangle$ to the ith agent

Agent:

1. receive $\langle uid, gids, c(x, y), r\rangle$ from the user
2. $gids' \leftarrow f_{id}(gids)$
3. $\langle c'(x', y'), r'\rangle \leftarrow \langle c(f_0(x), f_0(y)), f_0(r) + \lambda\rangle$
4. send $\langle c'(x', y'), r', gids', aid\rangle$ to the server

Server:

1. receive $\langle c'(x', y'), r', gids', aid\rangle$ from the agent
2. find users in the query range
3. remove users that not in any group of $gids'$
4. return query result $\langle qresult\rangle$ to agent aid

Agent:

1. receive the query result $\langle qid', qresult\rangle$ from the server
2. $qresult' \leftarrow$ reverse transform $qresult$
3. **for** each result qr in $qresult'$ **do**
4.   **if** (qr not an answer of the original query) **then**
5.   remove qr from $qresult'$

6.return $qresult'$ to user uid

User:

1. **for** $(i \leftarrow 0)$ **to** $(i < m)$ **do**
2.   receive $qresult$ from the ith agent
3. aggregate all the query results

Figure 5.8: Range Query Algorithm

Figure 5.7 gives a simple query example, where q_1 is a current circular range query and the dataset in Figure 5.6 is reused. We can see from the user space that O_3 and O_4 are the query answers. Since O_3 and O_4 are transformed by different

agents, in order to capture their transformed positions in the server space, q_1 needs to be transformed through all agents. The transformation generates queries q_1' and q_1''. Then q_1' will return the answer O_3' to agent A_1, and q_1'' will return the answer O_4' to agent A_2. Agents execute reverse transformations on the obtained answers and send the final answer O_3 and O_4 back to the user.

If a range query about static objects that have no privacy (e.g., restaurants) is submitted by the user, the algorithm in Figure 5.8 is simplified as follows. First, the query does not contain any user group information. Second, the user only sends it to any one of the agents. The agent does not need to do any transformation (i.e., steps 2 and 3 are skipped). The server then evaluates the query as usual, but this time using the static object database. Finally, the agent simply passes back the result obtained from the server to the user without doing any transformation.

K Nearest Neighbor Query

Given a query object with position (qx, qy), the k nearest neighbor query (kNN query) retrieves k objects for which no other objects are nearer to the query object at time t_q not prior to the current time.

One way to compute this kind of query is to transform the position of the query object using all the functions in the agent's transformation table. And the server needs to consider kNN for each transformed query position. For simplicity, we propose to compute the kNN query by iteratively performing range queries with an incrementally expanded search region until k answers are obtained.

The conversion from a kNN query to a range query is as follows. The first range query q_0 is centered at (qx, qy) with radius $r_0 = D_k$, where D_k is the estimated distance between the query object and its k'th nearest neighbor; D_k can be estimated

by the equation [91]:

$$D_k = \frac{2}{\sqrt{\pi}}\left[1 - \sqrt{1 - \left(\frac{k}{N}\right)^{\frac{1}{2}}}\right]$$

where N is the number of objects. The radius will be enlarged by $r_q = D_k/k$ at each iteration in query processing, until k answers are found.

Like the range query, a kNN query also needs to be sent to all agents. The main difference is that each agent needs to convert the kNN query to a range query first. Then the agent transforms the range query and the expansion parameter r_q, and sends them to the server (the transformed query and r_q are denoted as q' and r'_q respectively). The server will keep processing the range query q' with the radius extended by r'_q each time, and return the query result to the agent once it obtains k qualified answers. Finally, each agent computes the correct distance, and sends the distance along with the user IDs to the user that issued the query. The user then combines these to find his true k nearest neighbors.

For example (see Figure 5.7), q_2 is a nearest neighbor query, and q'_2 and q''_2 are corresponding queries in the server space after the transformation. From q'_2, agent A_1 gets a candidate nearest neighbor O'_1. From q''_2, agent A_2 gets a candidate nearest neighbor O'_2. Then the user will receive two candidates O_1 and O_2. After comparing the real distance between candidates and the query object, the user finally obtain its nearest neighbor O_1.

If a kNN query is executed over non-private static objects, the query just needs to be submitted to one of the agent, which does not do any transformation and forward the query to the server. The server executes the kNN query over the static object database and returns the result to the user through the help of the agent. If the query object of the kNN query is a private property (e.g., it is the current location of the user), then the kNN query can be converted to a range query in

order to hide the actual position of the query object.

5.4 System Analysis

This section analyzes the privacy protection, communication costs and concurrent processing in the LPP system.

5.4.1 Privacy

For the privacy analysis, we provide a formal model for better understanding and evaluation of the LPP system. We focus on location breach rather than ID protection in the following discussion. Several assumptions are adopted in the model. First, we assume that agents are trustable since they are lightweight systems and may be easily verified. This assumption is commonly used in many other location privacy protection methods (e.g. [13, 52]). Second, we assume that the server knows the overall architecture of the LPP system, which means the server knows from which agent an update or a query is sent. Based on these assumptions, we define our privacy model, *Spatial* $\Gamma-$*anonymity*, as below.

Definition 5 Spatial $\Gamma-$*anonymity*

Given a user U*,* U *is said to satisfy* Spatial $\Gamma-$*anonymity if the probability that the server can infer the position of this user is less than or equal to* Γ*.*

In the LPP system, a global privacy threshold Γ is guaranteed for all users by properly setting system parameters. Given a privacy requirement, there could be more than one applicable system settings. An important step of the system configuration is to define an analytical model of the privacy achieved by our approach. In what follows, let Γ_{LPP} denote the spatial anonymity achieved by the LPP system. We describe how Γ_{LPP} is formulated.

First, let us review the multiple transformation strategy. At each agent, the first transformation function is randomly selected and the following transformation functions are developed from the first function by using λ. We define Γ_{tr_i} as the probability that the first transformation function of agent i is disclosed, and Γ_{λ_i} as the probability that the λ value of agent i is disclosed. To guess one location of a user, the server needs to know the reverse transformation function of the corresponding agent, of which the probability is $\Gamma_{tr_i} \cdot \Gamma_{\lambda_i}$. Then for any user, we have Γ_{LPP} as follows:

$$\Gamma_{LPP} = max_{i=1}^{m}(\Gamma_{tr_i} \cdot \Gamma_{\lambda_i}) \tag{5.1}$$

where m is the number of agents. We now proceed to present how to obtain Γ_{tr} and Γ_{λ} and analyze possible threats in the LPP system (for convenience, we drop the subscript i from Γ_{tr_i} and Γ_{λ_i}). We will mainly introduce two types of privacy issues: privacy against location discovery and privacy against pattern discovery.

Γ_{tr} largely determines the **privacy against location discovery** since the data transformation is dominated by the first transformation function. To compute Γ_{tr}, we classify the servers into three categories: (i) Servers without any prior knowledge; (ii) Servers with weak prior knowledge; and (iii) Servers with strong prior knowledge.

We denote the user's original position as (x, y). After applying a combination of translation, scaling and rotation (i.e., the first transformation function), we obtain the transformed position (x', y'). The transformation process is formalized as follows:

$$\begin{cases} x' = R_\theta(d_x + s \cdot x) \\ y' = R_\theta(d_y + s \cdot y) \end{cases} \tag{5.2}$$

where R_θ denotes the rotation (θ is the angle), d_x and d_y are translation parameters, and s is the scaling parameter. The original domains of θ, d and s are denoted as $\mathbb{R}_0$, $\mathbb{D}_0$ and $\mathbb{S}_0$.

If the server does not have any prior knowledge, and in particular it does not even know the type of applied transformation, it is unable to determine (x, y) from (x', y') because the right side of the equation 5.2 is totally unknown to it. In this case, the probability Γ_{tr} that the server can infer the user's location at this agent is close to 0, which means that user locations have the maximum degree of privacy.

If the server has some weak prior knowledge, for example it knows the type of transformation and some constraints on the application, the original domain of the parameter can be narrowed to some extent. Let $\mathbb{R}$, $\mathbb{D}$ and $\mathbb{S}$ denote the new domains. To find the original location (x, y), the server needs to try all the combinations of the three transformation parameters in the new domains. Here, Γ_{tr} represents an estimate of the possibility of determining the original position. If the values in the domain are discrete, Γ_{tr} can be evaluated by equation 5.3, where $|\mathbb{R}|$, $|\mathbb{D}|$ and $|\mathbb{S}|$ are the cardinalities of the domains.

$$\Gamma_{tr} = \frac{1}{|\mathbb{R}| \cdot |\mathbb{D}| \cdot |\mathbb{S}|} \tag{5.3}$$

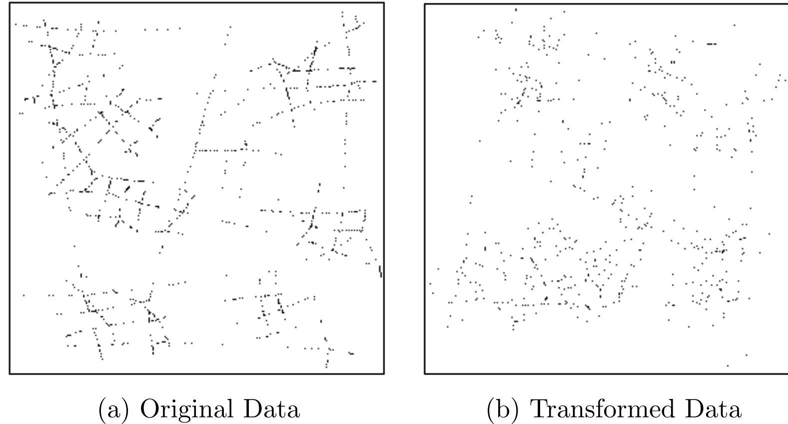
If the values in the domain are continuous, Γ_{tr} can be estimated by the volume of the three domains. Given the range of each domain to be $\mathbb{R} = [R^-, R^+]$, $\mathbb{D} = [D^-, D^+]$ and $\mathbb{S} = [S^-, S^+]$, and the granularity that an application requires to be G, we measure Γ_{tr} by equation 5.4.

$$\Gamma_{tr} = \frac{1}{(\frac{|R^+ - R^-|}{G} + 1)(\frac{|D^+ - D^-|}{G} + 1)(\frac{|S^+ - S^-|}{G} + 1)} \tag{5.4}$$

Just having the knowledge of the first transformation function, the server can only infer that the user location is within a certain circle with radius λ. λ is the value that indicates how much a transformed location will deviate from the one strictly preserving the relative distance. Therefore, we define Γ_λ in equation 5.5. The larger the λ, the harder it is to discover the real location of the user, and privacy is thus better protected.

$$\Gamma_\lambda = \frac{1}{\pi\lambda^2/G} \tag{5.5}$$

On the other hand, λ also protects **privacy against pattern discovery**. If the server has strong prior knowledge, such information may not only provide information on parameter constraints of the transformation functions, but may also indicate the pattern of distribution of users' locations. However, the identification of such patterns is still a difficult problem for both statisticians and computer scientists [19, 33], and after using our proposed multiple transformation strategy, the problem could become even harder as illustrated in Figure 5.9. Figure 5.9(a) shows the original data (about 1K user locations), from which we can clearly observe the road topology. Figure 5.9(b) shows the transformed data from one agent (3 agents in total), which is transformed by the combination of scaling, rotation and translation. We can see that after transformation, it is hard to identify the pattern; only some dense regions can be seen. To have better understanding of this advantage in the LPP system, let us look at the following example. Given that the server might use publicly available information (e.g. home addresses), to find out the corresponding person, the server first needs to map the home address to the transformed space, i.e., the server needs to find out the transformed home address, which is a very hard task as discussed above.



(a) Original Data (b) Transformed Data

Figure 5.9: Original Data vs. Transformed Data

To sum up, Γ_{tr} gives the probability that the server discover the single transformation function at each agent; and $\Gamma_{tr} \cdot \Gamma_{\lambda}$ gives the probability that the server discover the multiple transformation strategy. Then, the final Γ_{LPP} is the maximum value of $\Gamma_{tr} \cdot \Gamma_{\lambda}$ of m agents, which is the probability that the server knows about data transformation at any agent. We would like to mention that Γ_{LPP} is generally very small and can satisfy most privacy requirements. To have some idea of how small this Γ_{LPP} could be, let us look at the following example. Suppose at the agent with the most prior knowledge, the rotation domain has been constrained within 0 degree to 60 degree, the translation domain is [0..10], the scaling value is chosen from 1 to 3, $\pi\lambda^2$ is 10, and the granularity G is 10^{-6}. Thus Γ_{tr} is 5.6×10^{-10} and Γ_{λ} is 0.1, and consequently Γ_{LPP} is about only 6×10^{-11}. On the other hand, we can also see that by adjusting domain size or λ value, the LPP system can achieve a given privacy requirement. The detailed configuration is left to the future work.

Another common threat in network services is eavesdropping during communications. However, we do not consider it in our paper since this type of threat can be mitigated or avoided by data encryption.

5.4.2 Communication Cost

In our system, there are two types of operations: update and query operations. An update needs one round of communication between a user (agent) and an agent (server). Its communication cost is independent of the number of agents. A range query needs one round between a user and m agents, and a server and the m agents. A k nearest neighbor query may need several rounds of communications between a server and the m agents because agents need to inform the server if the received results contain enough answers. The notification messages sent to the server are very short compared to the entire query result set, and hence we do not consider their cost here. Suppose the message sizes of a query and a query result set are S_q and S_r respectively. The subquery result size is S_r in the worst case. Then the communication cost of a query is $2m(S_q + S_r)$. Since m determines the privacy level ($m = 1$ i.e. no privacy), the larger the value of m, the higher the privacy level would be. Therefore, a trade-off exists between the communication costs and the privacy level.

The trade-off issues between privacy and communication costs have been widely studied in context of network-level privacy protection. In particular, techniques have been devised to enhance network privacy by increasing the communication costs. For example, in [12, 73], in order to conceal the IP address, network packets have to go through m agents before reaching the receiver. In this case, a complexity of $O(m)$ for communication costs is required.

5.5 Performance Studies

5.5.1 Experimental Settings

All the experiments were run on a 2.6G Pentium IV desktop with 1Gbyte of memory. The page size is 4K. At the server side we employ the B^x-tree(H-curve) (presented in Chapter 3) to index moving objects. The original range query algorithm for the B^x-tree only supports rectangle ranges. We modified it to support the circle ranges by executing a regular rectangle range query which tightly covers the circle range, and then filtering the extra results. We compare both query and update performance of our model against the pure B^x-tree. Performance is measured in terms of disk page I/O.

We use the same synthetic datasets as used in the Chapter 3. The space domain is 1000×1000. In most experiments, we use uniform data, where users' positions are chosen randomly. The maximum interval between two successive updates by a user is 120 time units. Unless noted otherwise, we create the initial dataset for all users at time 0, and then evaluate the system performance after one maximum update interval during which each user has issued at least one update.

Parameter	Setting
Page size	4K
Buffer pages	50
Number of agents	2, **3**, 4, 5, ... , 20
$\lambda/f(r)$	0.01, ..., **0.05**, ... , 0.1
Time interval of changing function	0, 5, **10**, 15, ... , 50
Max update interval	120
Query size (diameter)	10, ..., **50**, ... , 100
Number of neighbors, k	1, 10, 20, 30, 40, 50
Number of queries	100
Data size	**100K**, ..., 1M
Data distribution	**uniform**, network-based

Table 5.1: Parameters and Their Settings

The parameters used in the experiments are summarized in Table 5.1, where values in bold denote the default values.

5.5.2 Range Queries

Impact of Super Queries

The notion of super query is an important component of our approach with respect to the protection of the map topology. However, super queries may introduce some false positives that may adversely affect performance. In the experiments reported here, we thus investigate the performance impact of the super query by examining the false positive rate. Recall that the false positive rate is the number of query answers filtered by the agent divided by the number of query answers received from the server. The smaller the false positive rate, the less additional work the server and the agent have to carry out.

First, we use the same size of range queries in a 100K dataset, and test the false positive rate when varying the values of λ. Figure 5.10 shows the results, where

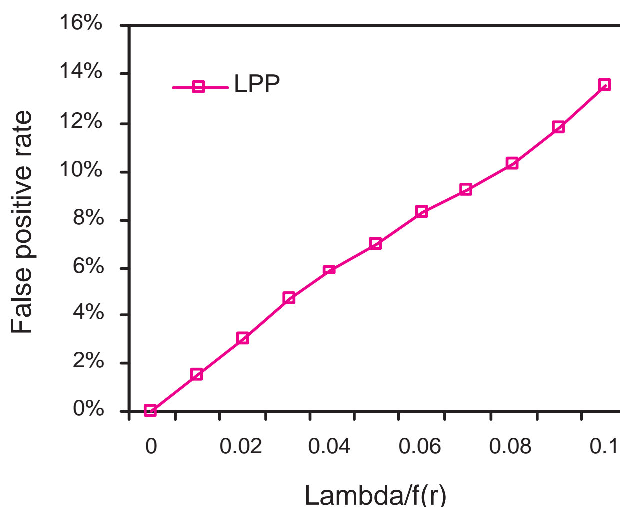


Figure 5.10: False Positive Rate for Varying λ

the x-axis is the rate of $\lambda/f(r)$ ($f(r)$ is the query radius after transformation). As expected, the false positive rate increases linearly with $\lambda/f(r)$; a larger λ results in a larger searching space.

Then, we fix the value of $\lambda/f(r)$ to 0.05 and vary the query range diameter from 10 to 100. Figure 5.11 shows the corresponding false positive rate. We can observe that the false positive rate decreases when the query size increases. As we know, the higher the value of λ is, the more obscure the transformed data pattern would be. This indicates that the LPP system provides higher privacy and with smaller performance overhead when the query size is large.

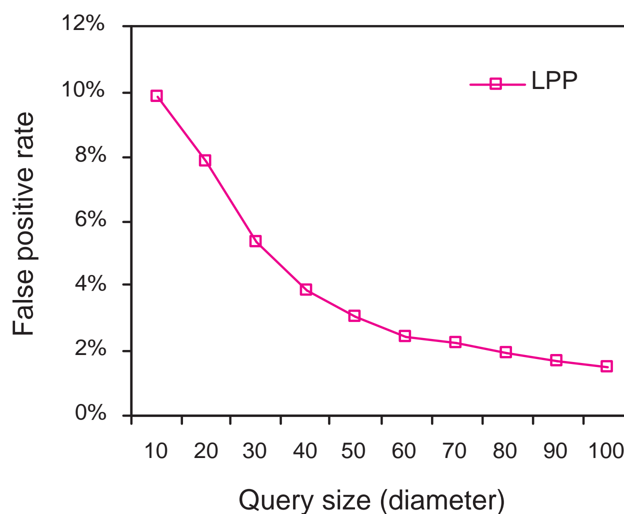
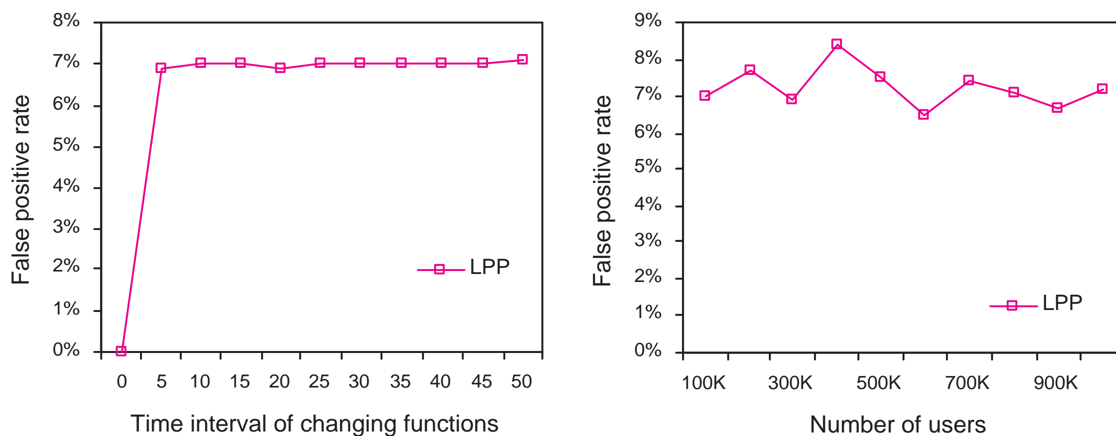


Figure 5.11: False Positive Rate for Varying Query Size

Next, we vary the time interval t_{int} between each pair of consecutive transformation functions. As shown in Figure 5.12(a), the false positive rate for different t_{int} is almost the same. The reason is that the super query is computed based on the first transformation function and the value of λ, and hence the frequency of the transformation function changes does not affect performance.

We also evaluate the false positive rate for values of data size ranging from 100K to 1M. Figure 5.12(b) shows that the false positive rate oscillates around 7% for



(a) Varying Time Interval

(b) Varying the Number of Users

Figure 5.12: False Positive Rate

different sizes of dataset. This again shows that the false positive rate is dominated by the rate of $\lambda/f(r)$ as stated in Theorem 1.

Impact of Data sizes

In this set of experiments, we vary data sizes and analyze the range query performance of the single B^x-tree and two versions of our model. "LPP (superquery)" denotes the version that uses the concept of the super query; "LPP (non-superquery)" denotes the version that uses the single transformation. The reason for comparing these two versions is to investigate the possible performance degradation incurred by the super query.

Figure 5.13 compares the query cost of the B^x-tree and the sum of query cost of all agents in our model. Based on the results reported in the figure, we can make the following observations. First, the performance of the approach based on the super query is quite similar to that of the approach based on the single transformation. The difference between them is less than 3%, which indicates that the use of the super query provides increased privacy protection without compromising query

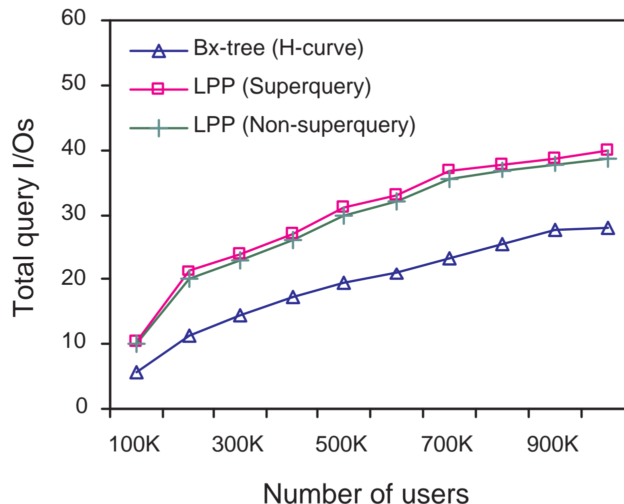


Figure 5.13: Impact of Data Sizes on Range Query Performance

performance. This is an important experimental result that validates a key idea of our approach. In the experiments reported in what follows, we thus only consider the version of our techniques that uses the super query. Second, given m agents, the total query cost of our approach is sometimes a little bit higher but not m times more than that of the B^x-tree. This is because one query will be sent to all agents according to our schema, and the server needs to compute the transformed queries from all agents. The cost of computing a query from an agent is less than that of evaluating a query in the single B^x-tree since the query from an agent is executed on a smaller dataset that maintains transformed data from the same agent.

Although our model may incur a little bit higher total query costs, the query response time of our model could be better given that the server supports multi-tasks or there are multiple servers; it can run multiple queries in parallel since each sub-dataset is relatively independent. As shown in Figure 5.14, the query cost corresponding to each agent is much smaller than that of the B^x-tree, and the difference increases with growing data size. This behavior is not surprising. As mentioned previously, the cost to compute a query from an agent is smaller because

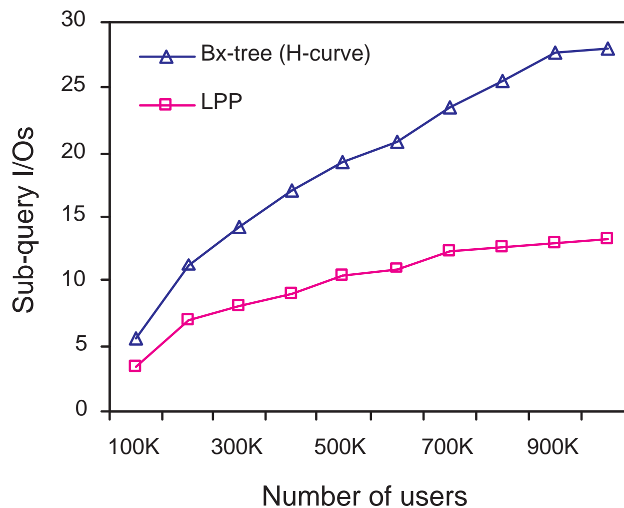


Figure 5.14: Query Cost of One Agent with Varying the Data Size

it is executed on a small sub-dataset.

Impact of Number of Agents

We next study the impact the number of agents has on the query performance. The B^x-tree is used as the baseline for comparison.

Figure 5.15 shows the total query cost as a function of the number of agents. We

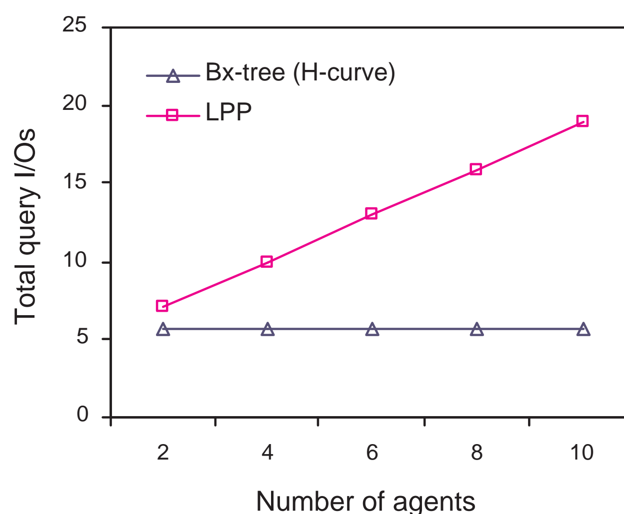


Figure 5.15: Impact of Number of Agents on Range Query Performance

observe that, for our model, the total query I/O cost increases with the number of agents. The underlying causation of this behavior is more complicated than what it looks, which is addressed in the following. The total query cost is determined by two factors: the query cost of one agent and the number of agents. When the number of agents increases, the query cost for one agent (i.e. sub-query cost) decreases due to the decreased dataset size with respect to one agent. However, the sub-query cost does not decrease as fast as the increase of the number of agents (as can be seen from Figure 5.16). Therefore, the resultant total query cost keeps increasing.

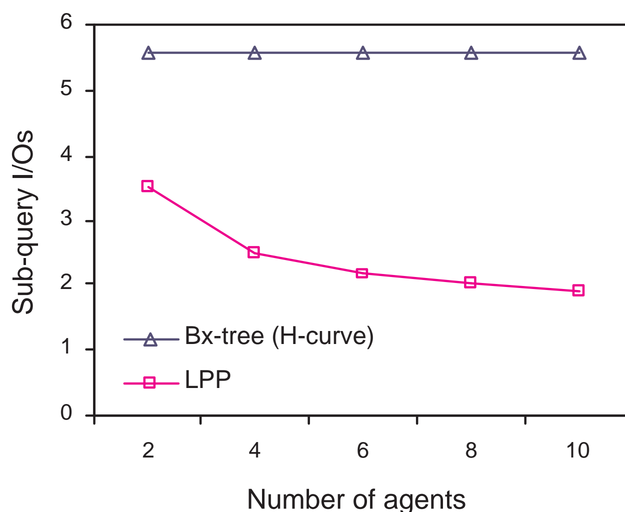


Figure 5.16: Query Cost of One Agent for varying number of agents

Figure 5.16 also indicates that our model may achieve better response time compared with the B^x-tree. This is because queries in our model may have more potentials of being executed in parallel.

In the following experiments, we explore the combined effect of the number of agents and data sizes. Figure 5.17 shows the results in the 100K and 500K dataset by using up to 20 agents. We can observe that the performance of 100K and 500K dataset demonstrates similar patterns.

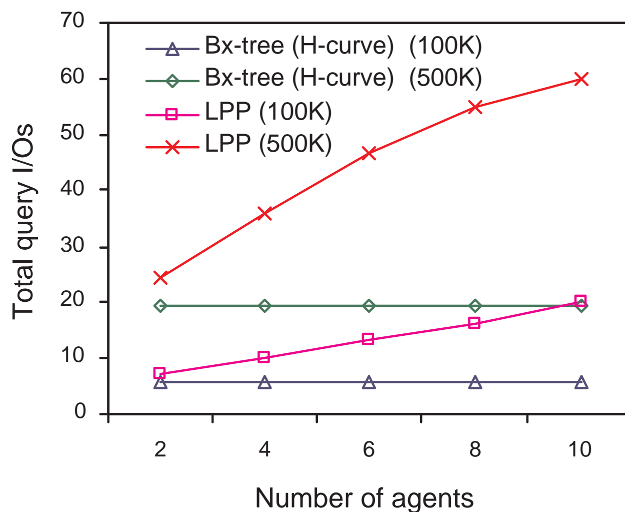


Figure 5.17: Impact of Number of Agents and Data Sizes on Range Query Performance

Impact of Query Sizes

In this section, we analyze the effect of query sizes by varying the query diameter from 10 to 100 for a dataset of size 100K. Figure 5.18 shows that the query costs of both B^x-tree and the LPP system increase with the query size. The reason is straightforward. Larger query ranges contain more objects and hence lead to more

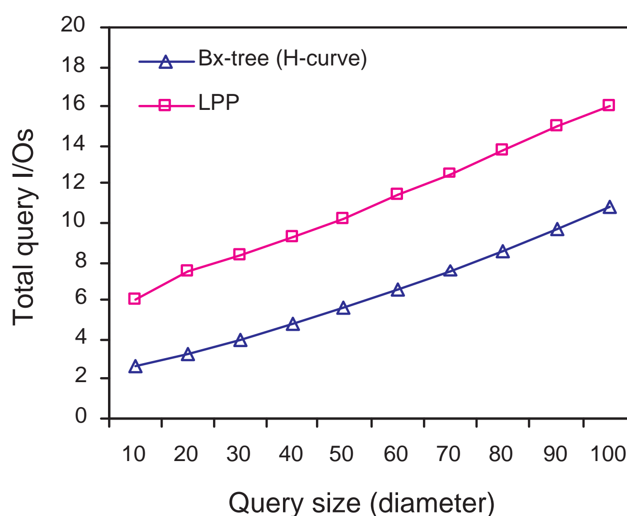


Figure 5.18: Impact of Query Size on Range Query Performance

tree node accesses.

Figure 5.19 compares the sub-query cost of each agent with that of the B^x-tree, which shows the similar performance patterns as that of previous experiments.

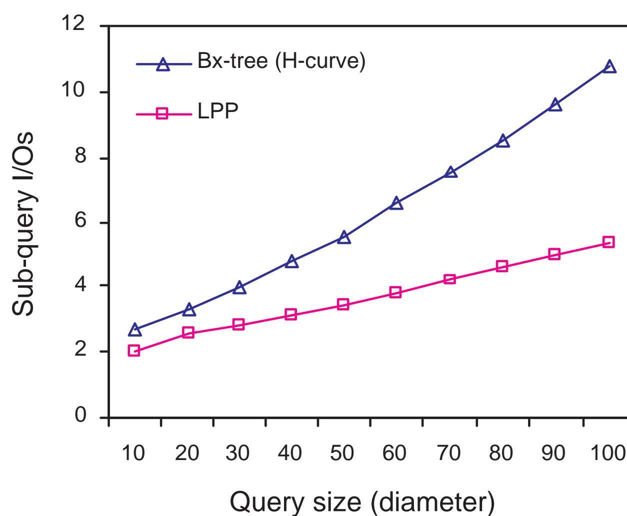


Figure 5.19: Query Cost of One Agent for Varying the Query Size

Impact of Skewed Data

To analyze the query performance on skewed data, we use network-based datasets with various destinations. The fewer the number of destinations, the more skewed the data is. Figure 5.20 shows the experiment results. We can observe that the difference of I/O costs between the B^x-tree and the LPP system keeps almost constant for different skewed data. The main reason could be that the performance of the B^x-tree is nearly independent of the data skewness (as examined in Chapter 3). Therefore, the performance of the sub-queries sent by the agents are also not affected much by data skewness.

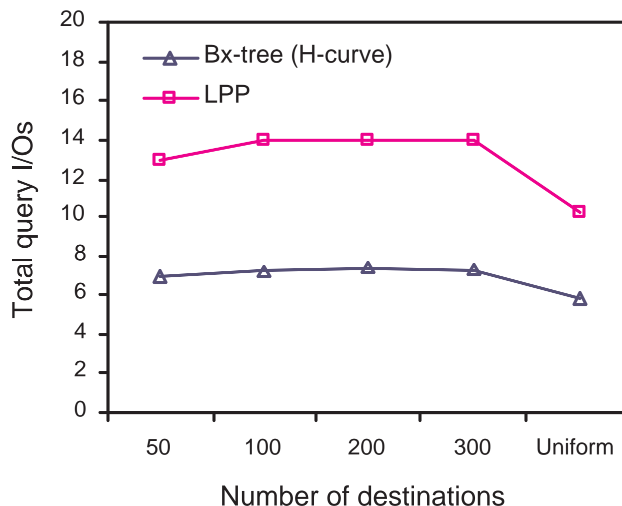


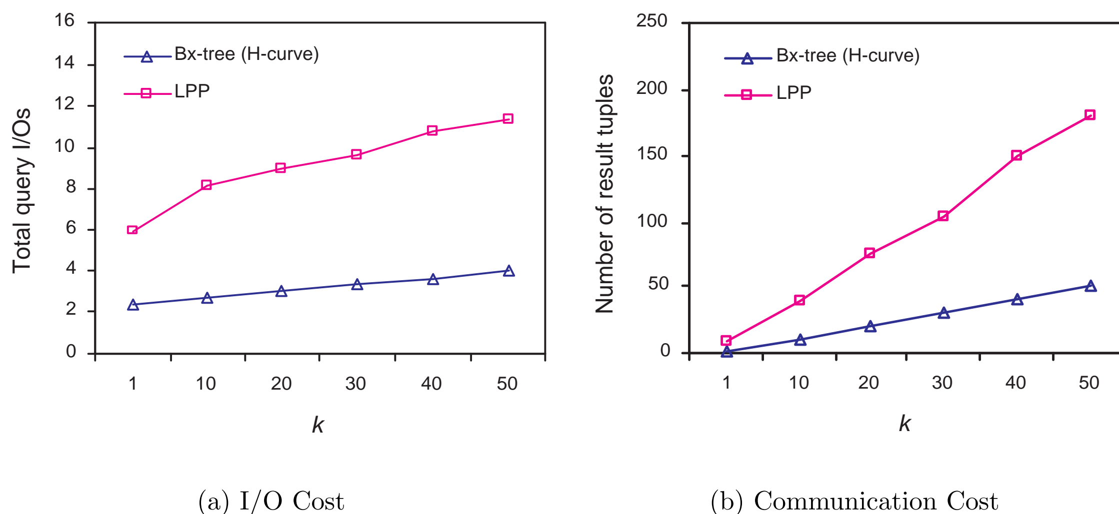
Figure 5.20: Impact of Skewed Data on Range Query Performance

5.5.3 K Nearest Neighbor Query

We proceed to evaluate the efficiency of kNN queries. Because the kNN query is treated as an incrementally expanded range queries, the difference of the query cost at the server side between the B^x-tree and the LPP system exhibits a behavior similar to that of range queries when considering the effect of the super query, data sizes, number of agents and so forth. Here, we present a representative result which is the impact of the value of k, that is, the number of required nearest neighbors.

As shown in Figure 5.21(a), the total query costs increase for both the B^x-tree and the LPP system as k increases. The LPP system has higher query cost because the server must execute a kNN query in each sub-dataset corresponding to each agent, and the search range would be bigger for the same k in a smaller dataset.

Next, we evaluate the communication cost between the server and the agents. Figure 5.21 shows the total number of tuples sent by the server. Compared to the B^x-tree, the LPP system has higher communication cost mainly because each agent needs to obtain k answers to ensure that the aggregate final result is correct. Given m agents, the communication cost of the LPP system is about m times higher than



(a) I/O Cost

(b) Communication Cost

Figure 5.21: Impact of k on kNN Query Performance

that of the B^x-tree. In addition, we also note that there may be multiple rounds of communications between the server and the agents since the agents need to inform the server whether the received result sets contain enough final answers. However, the notification message is extremely small (e.g. one bit would be enough to tell the server to continue querying or stop querying); and in our experiments, we observe that most kNN queries (more than 90%) only require one round of communication, and very few queries require two rounds. Therefore, the cost of sending notification messages can be ignored compared to the cost of sending query result sets.

5.5.4 Update

We now compare the average update cost (amortized over insertion and deletion) of our model against the B^x-tree.

Impact of Data Sizes

First we examine the update performance with respect to the dataset size. We compute the average update cost after the maximum update interval of 120 time units. From Figure 5.22, we can see that our model achieves same performance as the B^x-tree. The reason is that each update in the LPP system is sent to only one

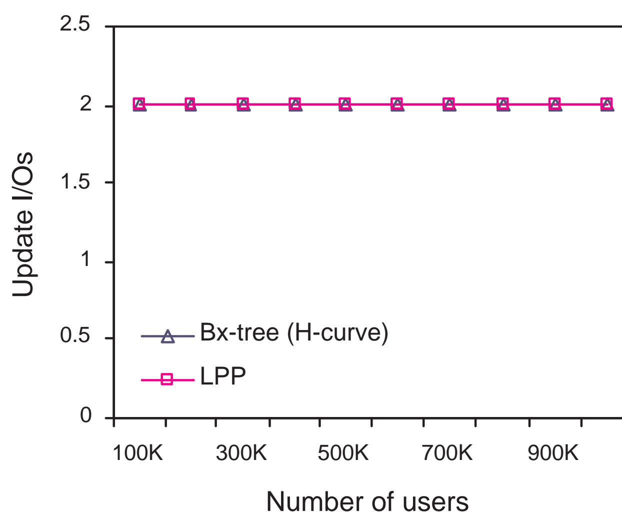


Figure 5.22: Impact of Data Sizes on Update Performance

agent. That means each update affects only one sub-dataset stored by the server, i.e., only one B^x-tree corresponding to that sub-dataset needs to be updated.

Impact of the Number of Agents

In this section, we investigate the update performance of our model when using varying values for the number of agents in the system. Figure 5.23 shows the update I/O cost. Observe that the update cost of our model is still the same as that of the B^x-tree. With the increase of the number of agents, the dataset of each agent becomes smaller. Although an update executed in a smaller dataset in the LPP system is expected to be more efficient, we cannot see this difference from the figure since the B^x-tree already achieves near minimum update cost which is

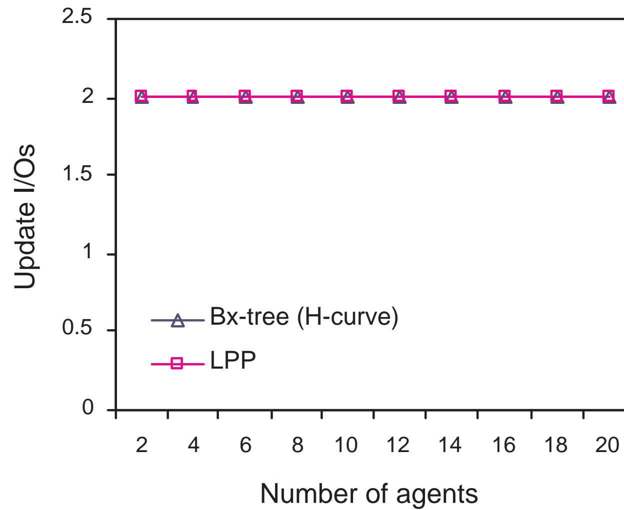


Figure 5.23: Effect of Number of Agents on Update Performance

corresponding to one read and one write on the leaf node containing the updated object.

Impact of Skewed Data

Finally, we evaluate the update performance in the skewed datasets that we used in the experiments on queries. As shown in the Figure 5.24, both the B^x-tree and the LPP system have a performance similar to that of the uniform datasets. This is becase the update cost of the B^x-tree (or the small B^x-trees in the LPP system) is only subject to the height of the tree and almost independent of other factors.

5.6 Summary

In this chapter, we propose a novel system framework to address the problems of location privacy in moving-object environments. Our framework achieves both high assurance privacy and good performance. Specifically, our framework uses a number of agents in-between users and servers. Agents are lightweight systems

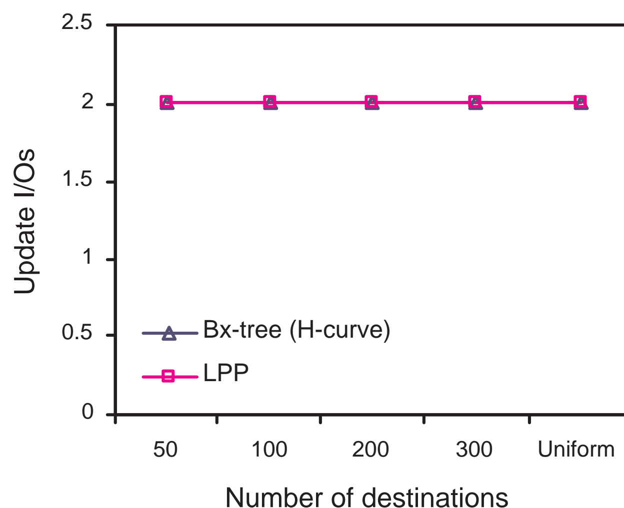


Figure 5.24: Effect of Data Distribution on Update Performance

which do not store any user information, but only perform data transformation. In this way, our system can prevent servers from knowing exact locations of users, and even the map topology. We have also developed metrics to analyze the degree of privacy protection.

We have carried out extensive performance studies to assess the impact of various parameters. We have tested our technique on both uniform and skewed data, and have analyzed the impact of various parameters, such as data size, number of agents and query sizes. We have also compared the performance of our technique with the B^x-tree without any privacy protection. The results show that our approach has a little higher query cost but the same update performance as the B^x-tree.

CHAPTER 6

Adapting Relational Database Engine to Accommodate Moving Objects

Past few years have witnessed considerable research efforts on moving objects databases. These previous works so far have been emphasizing on different aspects including data modelling, novel indexing, efficient query processing, etc. Whereas from a practical perspective, a real system supporting moving objects storage and retrieval needs to take into account all these aspects and to integrate them in a systematic way. There are two approaches to implementing an MOD system. One, a specialized system could be developed from scratch. Such a system is lean and efficient. However, it may offer less functionalities that are required in conventional business processing, and such approach is time consuming and costly. Two, an existing DBMS is extended to provide the required capability for handling spatio-temporal data and processing. Such an approach is less costly and its existing core database engine can continue to serve the conventional business processing, and

complement the MOD applications. This is in line with the approach being taken in the commercial DBMS products where different cartridges are built on top of the data server for different applications. However, such an approach may be tricky as any major alteration to the data server may cause many of its core components to be affected. Further, it may not yield the best performance since riding on top of already bulky system offers less room for optimization. In this chapter, we will discuss our implementation for moving objects on top of a popular relational database system MySQL. The B^x-tree is a B^+-tree index, and it could even be implemented as a stored procedure in existing DBMS without touching the code. Although we have the source code of the MySQL, we keep the modification to the minimal. In our implementation, moving object data is transformed and stored directly on MySQL, and queries are transformed into standard SQL statements which are efficiently processed in the relational engine. Most importantly, all these are achieved neatly and independently without infiltrating into the MySQL core.

The rest of the chapter is organized as follows. Section 6.1 presents the system architecture with MySQL as the underling relational engine. Section 6.2 concretizes the integration of the B^x-tree to the MySQL. Section 6.3 shows the system performance. Finally, Section 6.4 concludes.

6.1 System Overview

This section gives an overview of our system – the SpADE system which stands for "*Sp*atio-temporal *A*utonomic *D*atabase *E*ngine for managing moving objects". Figure 6.1 demonstrates the system architecture.

Our proposal assumes a client/server architecture, within which moving objects are regarded as mobile clients and data server as the server respectively. Non-static

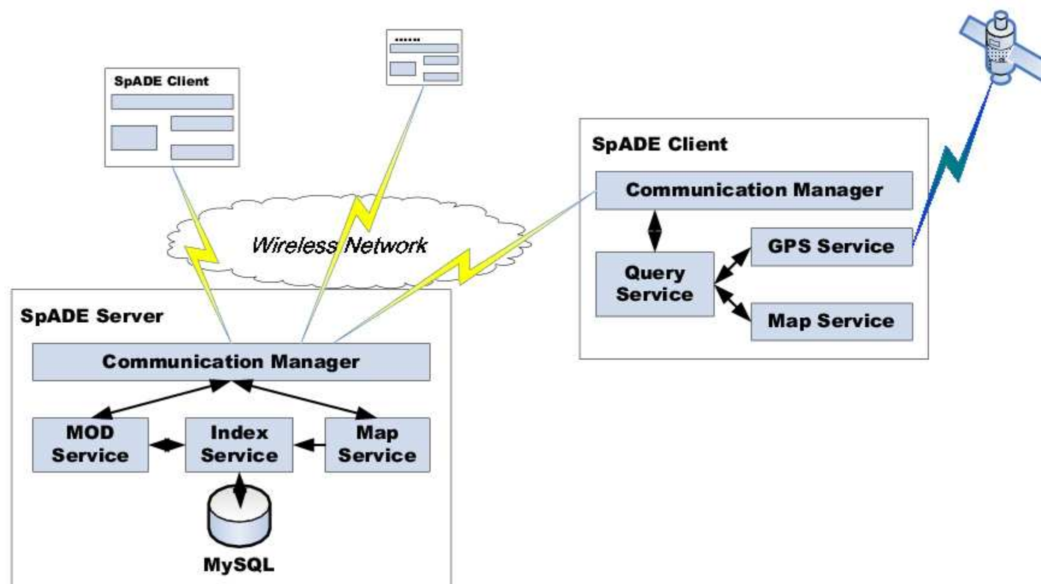


Figure 6.1: System Architecture

entities like vehicles and pedestrians are abstracted as moving objects. They obtain positioning information with GPS (Global Positioning System) receivers installed, and are able to communicate via wireless network with the server, sending queries to and receiving results from it. The server is responsible for managing moving object information and processing queries from mobile users. By employing the industry standard JDBC for the data access, our server can also support providing services for other application interfaces such as the Web.

6.1.1 The SpADE Client

In the SpADE system, each client has a mobile computing device (e.g., PDA or high-end mobile phone) that is equipped with the SpADE client software and a GPS receiver. Thus, all these moving objects are able to report their latest velocity and position information to the SpADE server. Specifically, a SpADE client includes the following four components:

- *GPS Service Module* is responsible for receiving and parsing the GPS information. Therefore, each moving object can know its own latest velocity and position information.

- *Map Service Module* is responsible for displaying moving objects on a city map, listening to all events from the user, and downloading the latest city map from the SpADE server.

- *Query Service Module* is in charge of constructing the spatial-temporal queries according to the user's input or pen event on the map.

- *Communication Manager* listens to all network messages occurring between the client and the server and then dispatches them to the corresponding service module for further processing.

6.1.2 The SpADE Server

In general, the SpADE server is responsible for storing the velocity and position data of all moving objects and processing spatial-temporal queries issued by clients. Specifically, the SpADE server comprises the following five modules:

- *Communication Manager* receives query messages or GPS data updating requests from the clients, dispatches them to other modules, and returns query results to the client user.

- *Map Service Module* maintains the map data displayed at the client side and notifies all clients to update their map if a new map is provided.

- *MOD (Moving Object Data) Service Module* is used to define the motion pattern of the moving objects for different moving objects with different motion patterns. A distinguishing feature of the SpADE system is that it allows users

to plug their preferred motion patterns to the server for precisely predicting the future position of the moving objects. This makes the SpADE system very flexible in terms of customizing the most suitable motion pattern for a particular moving object.

- *Index Service Module* is responsible for constructing and maintaining the indices used for processing spatial-temporal queries. Similar to the MOD service module, a user can add a particular index structure into the index service module for the motion pattern customized by the user. In our implementation, we adopt the B^x-tree.

- *MySQL Database* is used to store index and query the velocity and position data of all moving objects. It contains three types of tables, namely *user table*, *moving object table* and *static object table*. The user table stores all users registered into the SpADE system, which is used to evaluate the validation of moving clients; the moving object table stores moving object information (e.g. positions and velocities); and the static object table stores the road network information as well as buildings in the city.

6.1.3 Client/Server Protocols in SpADE

In our system, each SpADE client accesses the SpADE server via the standard socket protocol. Note that the usual java package (e.g. java.sql) cannot be used to develop the client program for the purpose of interacting with the server in a wireless and mobile environment.

Information from the clients such as register request, login request, update and query request, is represented as a simple byte stream and parsed by the SpADE server.

6.2 System Implementation

In this section, we describe the implementation of the B^x-tree on top of the MySQL. We first briefly review the B^x-tree (details can be found in Chapter 3), and then discuss the implementation issues occurring in the indexing and query phases respectively.

6.2.1 Data Modelling and the B^x-tree

We model a moving object by $O = (\overrightarrow{x}, \overrightarrow{v})$, a position and a velocity respectively. Every moving object updates its information to the data server when it feels necessary at time t_u.

To harness the widely available and efficient index B^+-tree, the first task is to linearize the representation of the locations of the moving objects. This is done by means of a space-filling curve, which enumerates every point in the discrete, multi-dimensional space. Attractive space-filling curves such as the Peano curve (or Z-curve), which we use in our system, preserve proximity, meaning that points close in the multidimensional space tend to be close in the one-dimensional space obtained by the curve [58].

Since the B^+-tree with space-filling curves only works well in static databases, in order to maintain the proximity preserving property in moving object environments, the moving objects need to be differentiated based on their timestamps. Therefore, the index is effectively partitioned by placing entries in partitions based on their update time. Specifically, the time axis is partitioned into intervals of duration Δt_{mu}, and then each such interval is further partitioned into n equal-length subintervals, termed *phases*. Here Δt_{mu} is the maximum duration in-between two updates of any object location.

Then, updates in the same phase are mapped to the same so-called *label timestamp*. For an object with label timestamp t_{lab}, its position at t_{lab} is computed according to its position and velocity at t_u. This future position is then transformed to a 1-dimensional value by applying the space-filling curve.

Finally, an object O updated at t_u is represented by a value $B^x value(O, t_u)$:

$$B^x value(O, t_u) = [index_partition]_2 \oplus [x_rep]_2 \tag{6.1}$$

where *index_partition* is an index partition determined by the update time, *x_rep* is obtained using a space-filling curve, $[x]_2$ denotes the binary value of x, and $\oplus$ denotes concatenation. And the two binary value are computed as follows:

$index_partition = (t_{lab}/(\Delta t_{mu}/n) - 1) \bmod (n+1)$

$x_rep = x_value(\overrightarrow{x} + \overrightarrow{v} \cdot (t_{lab} - t_u))$

6.2.2 Implementation Issues

In the implementation, we have the following technical issues to resolve.

Relational Table Definition

First, we need to define for moving objects a proper relational table that can be indexed by the B^+-tree so that we can exploit its power when manipulating moving object information. For this purpose, we add a field B^x_Value (computed by equation 6.1) into the table of moving object and make it as the primary key indexed by the B^+-tree. A simple relation scheme is demonstrated in Table 6.2.2 with concise explanations. Extra fields can also be defined according to the practical needs in a specific application.

In our system, there are two such tables. One is to store the latest update

Field	**Type**	**Note**
id	integer	Object identifier
$B^x_Value^*$	long	Value of formula 6.1
pos_x	double	Longitude of last update
pos_y	double	Latitude of last update
vel_x	double	Longitude speed of last update
vel_y	double	Latitude speed of last update
time	long	Update time

Table 6.1: Moving Object Relation Scheme

information of the moving objects, called the *current information table.* Note that this table is indexed by the B^x_Value obtained according to the B^x-tree rationale. The other table simply keeps all the historical information, called the *historical information table.*

Time Synchronization and Representation

The next important issue is to synchronize time between clients and the server because the time displayed in users' mobile devices may be a little different from the server time. In our system, we always treat the server time as the standard time. All the messages received from clients are stamped with the current server time. For example, an incoming update is assigned with the current server time and then stored in the table. Moreover, the server will notify the clients to adjust their local time if the time difference between them is higher than a threshold.

For the representation, we use *long* type for time by converting the real time (date and time) into a long value. Besides, as the B^x-tree uses a periodic time representation for the index partition, we also need to "roll up" the real time periodically in our system. In other words, the server need to transfer the real time to the label timestamp t_{lab} periodically.

Space Extent Normalization

Another issue is the map extent and space-filling curve transformation. For brevity and computation simplicity, we usually assume a unit data space when using a space-filling curve to represent moving object positions in the B^x-tree. But this is not the case when we turn to real applications where geographical maps are used. To deal with this, we add a coordinate normalization module in front of the B^x-tree implementation. When geo-related information is received from any moving object, the relevant geographic coordinates are transformed into the ones acceptable by the space-filling curves, and then used to generate B^x values.

Processing Spatial-Temporal Queries

In the SpADE system, the typical process of executing a spatial-temporal query will take several steps involving different system components, as shown in Figure 6.2.

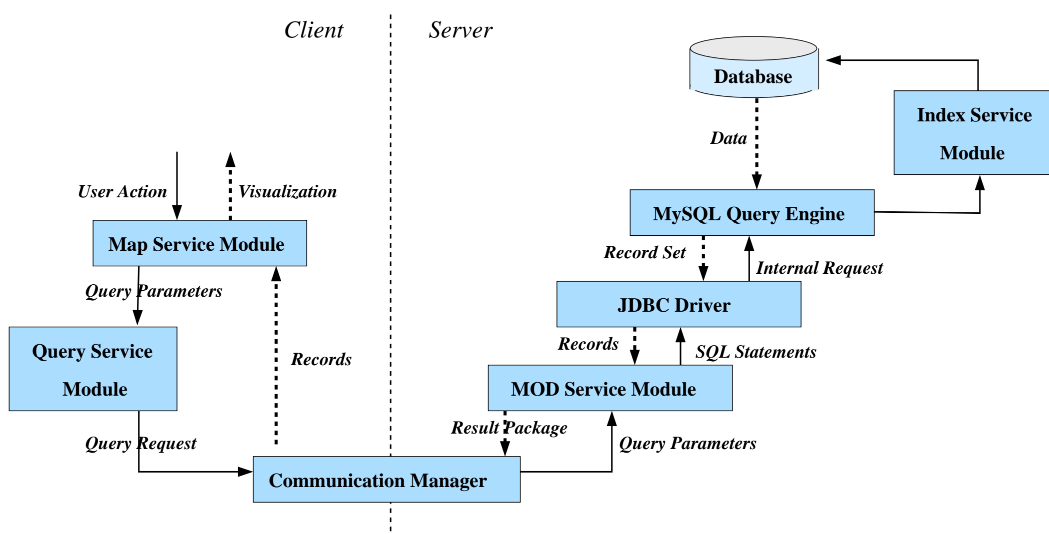


Figure 6.2: Execution of a Spatial-temporal Query

1. On the client side, a user first selects the query type (e.g. kNN or range query). The map service module then obtains the query parameters by lis-

tening to the user's pen event on the map.

2. Then, the query service module constructs the user query according to the query parameters and sends the query request to the server via the communication component.

3. Once receiving the query message, the server side communication part extracts query parameters and passes them to the MOD service module.

4. The MOD service module composes SQL statements according to the query parameters and the B^x-tree rationale, and passes the SQL statements to the MySQL query engine via the JDBC connection.

5. Inside the MySQL, the index established will be exploited to retrieve data requested. The data will be returned to MOD service module via the JDBC connection.

6. Along the reversed direction, the data will be sent back to the client with necessary packing/unpacking operations inserting/extracting corresponding controlling bytes.

7. Finally, when the map service module on client side receives the desired moving objects with the velocity and position information, it can visualize the results on the map according to the user's preference specified beforehand.

We now proceed to elaborate the query processing in steps 4 and 5 which involves the interaction with the MySQL database. We first address the current and near-future queries which are directly supported by the B^x-tree. As described in [36], in the B^x-tree, a range query in the 2-dimensional space is first enlarged and then transformed into a set of range queries in the 1-dimensional space. Each subquery corresponds to a sequence of B^x_Values, which are used to locate records in

the MySQL database. Specifically, our implementation expresses the sub-queries as the traditional SQL statements by using the start (end) point of the sequence of B^x_Values as the lower (upper) bound for the searching. It is worth noting that we do not need multiple SQL statements for sub-queries since too frequent interactions with the MySQL query engine may result in poor query performance. Instead, we combine all the query conditions into one SQL statement. For example, suppose m sub-queries are represented by start and end points of their corresponding sequences, i.e., $[\langle sub_0^{st}, sub_0^{ed}\rangle, \langle sub_1^{st}, sub_1^{ed}\rangle, ..., \langle sub_m^{st}, sub_m^{ed}\rangle]$ ("st" denotes the start point, "ed" denotes the end point), and the data table name is MO, the resultant SQL statement is the following:

SELECT $*$
FROM MO
WHERE ($B^x_Value \geq sub_0^{st}$ AND $B^x_Value \leq sub_0^{ed}$)
OR ($B^x_Value \geq sub_1^{st}$ AND $B^x_Value \leq sub_1^{ed}$)
OR ...
OR ($B^x_Value \geq sub_m^{st}$ AND $B^x_Value \leq sub_m^{ed}$);

To further improve the query efficiency, we propose to combine sub-queries with B^x_Values close to one another. The two sub-queries are considered to be close if the interval between their sequence is less than a given threshold. According to our experiments, this strategy can greatly reduce the total number of conditions in the SQL statements and achieve better query performance.

The retrieved objects are only candidates of the final results. They will be further examined before correct ones are returned to the user. Other types of queries like the k nearest neighbor queries and the continuous queries can be easily solved by the algorithms of the range queries with minor modifications.

From the above discussion, we can clearly see that our implementation of query processing is non-intrusive, and such implementation could easily be ported to other backends or platforms.

Updating Spatial-Temporal Data

To query moving objects, the velocity and position of all moving objects must be kept fresh. The SpADE system provides two ways to update a moving object's spatial-temporal information.

1. Users are able to set their preferred update frequency to determine how often to update their velocity and position data. When the setting time has expired, the latest GPS information will be automatically reported to the server.

2. The client queries the server at a regular time interval to check if the distance between the predicted location and the actual location exceeds the system threshold. If it is the case, the new GPS data will be updated to the server.

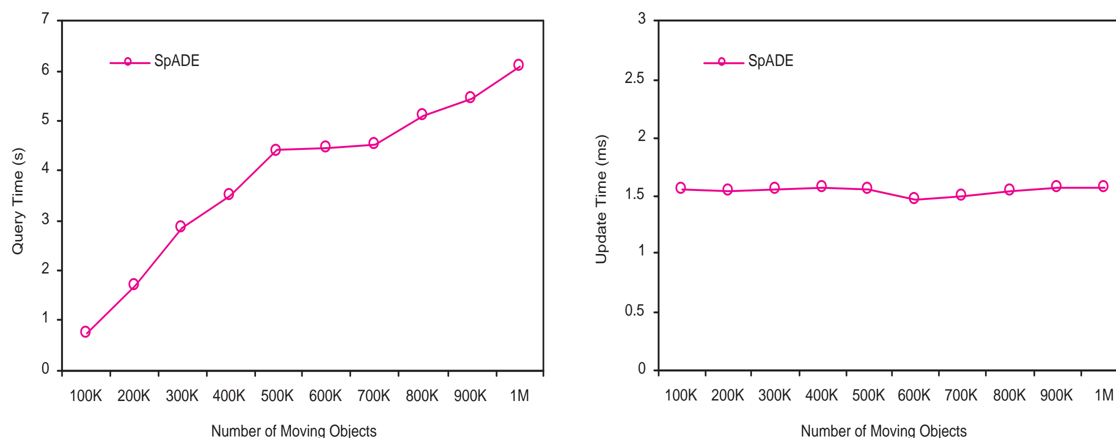
Generally, the SpADE system takes four steps to refresh the velocity and position information of all moving objects. In the first step, the client obtains the velocity and position information from the GPS service module and then transfers them to the server. In the second step, according to the identifier of the moving object, the old information is removed from the table and inserted into a history table. In the third step, the index service module calculates the index key of the B^x-tree in terms of the new velocity and position data. Finally, the new index key, velocity and position data are inserted into the MySQL database.

6.3 Performance Studies

The SpADE system resides at an IBM x255 server running Linux with four Intel Xeon MP 3.0GHz/400MHz processors and 18G DDR main memory. To evaluate the system performance, we simulate clients from 100K to 1M and measure the response time of each range query and update. The parameters used is the same as the default settings in Chapter 3. In the following, we present two representative results.

First, we show the average range query performance in Figure 6.3(a). We can observe that the average response time increases with the number of moving objects (i.e. clients). This is mainly due to the corresponding increase of the number of moving objects inside the query range which requires the underlying MySQL database in the SpADE system to retrieve more data. Note that the performance pattern is similar to that shown in Figure 3.21 of Chapter 3 which examines the query performance of the B^x-tree.

Figure 6.3(b) shows the average update response time of the SpADE system.



(a) Query Response Time

(b) Update Response Time

Figure 6.3: Query and Update Performance of SpADE System

As expected, the update cost keeps constant when the number of moving objects grows up. This is because in the SpADE system, each insertion or deletion only incurs one insertion or deletion in the MySQL database which has a B^+-tree on the primary key, i.e., the B^x_Value.

To sum up, our SpADE system preserves the properties of the B^x-tree and is ready to provide high quality service to a large number of clients.

6.4 Summary

In this chapter, we present our design on extending an existing DBMS to support spatio-temporal query processing. We present the architecture of SpADE. SpADE is based on a client/server architecture, within which the moving object data is stored and managed on the server on top of a relational database system called MySQL. On the server side, we made use of the the B^+-tree to implement the B^x-tree for indexing moving objects. We present implementation issues related to both updates and queries. Note that all these have been achieved cleanly without rewriting any main components of MySQL backend. It is obvious that our proposed design could be implemented on any proprietary commercial backends cost effectively.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

Moving object applications that entail the storage of samples of continuous, multi-dimensional variables pose new challenges to the traditional database technology. This thesis addresses the challenges of providing supports for indexing, querying and privacy protections in moving-object environments. Also, this thesis presents the system architecture of SpADE.

For the indexing, we proposed a new index structure, the B^x-tree, which indexes current and near-future positions of moving objects. The B^x-tree uses a new linearization technique to exploit the volatility of the data values being indexed (i.e. moving object positions) such that the moving points can be indexed using a classical B^+-tree. The B^x-tree is able to support various types of queries, such as predictive interval range queries and predictive k nearest neighbor queries. According to our extensive experimental studies, the B^x-tree is efficient and robust

regarding both update and query operations. In fact, the B^x-tree significantly outperforms the TPR*-tree, especially for the update operations. This result is not surprising since being a B^+-tree based index, the B^x-tree inherits good properties of the B^+-tree and avoids multiple-path searching during the update processing.

One possible limitation of the B^x-tree though is that it is a little sensitive to the parameters. However, compared to its competitors, the B^x-tree is still far more efficient. Unlike its competitors, the B^x-tree is elegant in design and scalable in terms of data sizes and page sizes, and can be incorporated into existing DBMS cost effectively.

Apart from the queries that are directly supported by the B^x-tree such as the range, KNN and respective continuous queries, we also studied an emerging type of query. The density query is to locate the regions with a density higher than a given threshold. In this thesis, we first presented the definition for the density query which eliminates the answer loss problem, and then proposed a two-phase filter-refinement framework which can be applied to most existing index structures. The experimental results show that our approach achieves efficient query performance and requires little storage space. The good query performance is possibly attributed to the filtering. The filter phase maintains a density histogram, which enables quick pruning and reduces the number of candidates to be further examined. The compact storage space is mainly due to the use of Discrete Cosine Transformation which compresses the density histogram to a great extent. To sum up, our framework provides a way to handle density queries efficiently. Such techniques may be very useful in a traffic control system to help predict possible traffic jams.

With the expanding use of the Location-Based Services, such as the traffic control system just mentioned, users are becoming more sensitive towards the privacy

issues when they want to subscribe to such services. Usually, users may not be willing to disclose their personal information to the service providers. Therefore, our study attempted to reduce the chance of information leak while still providing the Location-Based Services. Our proposed system framework achieves high assurance privacy without sacrificing the service quality. Specifically, the framework not only prevents the service provider from knowing the exact locations of users, but also prevents users' locations from being disclosed to other users that are not authorized. The main idea is to employ agents in-between servers and users, which only serve as information passages and do not store any user data. These agents transform the user IDs and locations before sending them to the server. Therefore, the server only knows and handles the transformed data. To obtain higher privacy protection, we may need to pay more communication costs. A trade-off exists when taking into account the specific system configurations. Extensive experimental studies were conducted and the results indicate that our technique effective and feasible.

Besides the theoretical studies on various aspects of the Moving Objects Databases, we proposed a system design by extending an existing DBMS. We implemented the B^x-tree on the top of a popular relational database system MySQL. The strength of our design and the proposal is that we do not have to touch MySQL core extensively in order to manage moving objects.

7.2 Future work

There are several promising directions for future work in the research area presented in this thesis. The directions range from direct extensions of the research to applying the fundamental ideas to other applications.

For the indexing structure, one possible direction is to improve the range query performance in the B^x-tree since the current range query algorithm uses the strategy of enlarging query windows which may incur some redundant search. Another direction is to apply the linearization technique to other index structures.

For the density query, we can further consider the algorithms that support continuous queries and dense regions of different sizes or shapes, e.g., convex regions.

For the privacy issues, further study is needed to refine the proposed privacy protection metrics by assuming more prior knowledge that the adversary may possess, so as to better assess privacy risks. And another challenge is to support continuous query.

In terms of practice, we have implemented the B^x-tree index structure in our proposed real database system SpADE. We are considering deploying all the other techniques in the real DBMS, such as supporting density queries and providing privacy protections. These raises important research problems at the system level, in terms of the interaction of such algorithms with the actual systems, and issues related to adaptiveness and frequency of executions, etc.

BIBLIOGRAPHY

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. PODS*, pages 175–186, 2000.

[2] P. K. Agarwal and C. M. Procopiuc. Advances in indexing for mobile objects. *IEEE Data Eng. Bull.*, 25(2):25–34, 2002.

[3] M. Ankerst, M. Breunig, H. P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *Proc. ACM SIGMOD*, pages 49–60, 1999.

[4] D. H. Ballard. Strip trees: A hierarchical representation for curves. *Commun. of the ACM*, 24(5):310–321, 1981.

[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB Journal*, 5(4):264–275, 1996.

[6] N. Beckmann, H. P. Kriegel, R.Schneider, and B.Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 322–331, 1990.

[7] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proc. IDEAS*, pages 44–53, 2002.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[9] S. Berchtold, B. Ertl, D. A. Keim, H. P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensiona space. In *Proc. ICDE*, pages 209–218, 1998.

[10] S. Berchtold, D. A. Keim, and H. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. VLDB*, pages 28–39, 1996.

[11] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[12] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Comm. of ACM*, 24(2):84–88, 1981.

[13] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar. Preserving user location privacy in mobile data management infrastructures. In *Proc. Workshop on Privacy Enhancing Technologies*, pages 393–412, 2006.

[14] K. L. Cheung and A. W c. Fu. Enhanced nearest neighbor search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[15] H. D. Chon, D. Agrawal, and A. E. Abbadi. Using space-time grid for efficient management of moving objects. In *Proc. MobiDE*, pages 59–65, 2001.

[16] A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. *DB Technical Report TR-5*, 2004.

[17] A. Civilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. In *Proc. Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services*, pages 164–173, 2004.

[18] B. Cui, D. Lin, and K. L. Tan. Towards optimal utilization of main memory for moving object indexing. In *Proc. DASFAA*, pages 600–611, 2005.

[19] D. L. Donoho and X. Huo. Beamlet pyramids: a new form of multiresolution analysis suited for extracting lines, curves, and objects from very noisy image data. In *Proc. SPIE*, pages 434–444, 2000.

[20] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. PODS*, pages 247–252, 1989.

[21] R. A. Finkel and J. L. Bentley. Quadtrees: A data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, 1974.

[22] E. Frentzos. Indexing objects moving on fixed networks. In *Proc. SSTD*, pages 289–305, 2003.

[23] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Comp. Surv.*, 30(2):170–231, 1998.

[24] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *Proc. IEEE ICDCS*, pages 620–629, 2005.

[25] G. Ghinita, P. Kalnis, and S. Skiadopoulos. Prive: Anonymous location-based queries in distributed mobile systems. *Technical Report TRB7/06*, 2006.

[26] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proc. MobiSys*, pages 31–42, 2003.

[27] M. Gruteser and X. Liu. Protecting privacy in continuous location-tracking applications. *IEEE Security and Privacy*, 2(2):28–31, 2004.

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

[29] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databses. In *Proc. SSTD*, pages 306–324, 2003.

[30] U. Hengartner and P. Steenkiste. Protecting access to people location information. In *Proc. SPC*, pages 25–38, 2003.

[31] G. R. Hjaltason and H. Samet. Speeding up construction of pmr quadtree-based spatial indexes. *VLDB Journal*, 11(2):109–137, 2002.

[32] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proc. VLDB*, pages 720–731, 2004.

[33] X. Huo and D. L. Donoho. Recovering filamentary objects in severely degraded binary images using beamlet-decorated partitioning. In *Proc. ICASSP*, 2002.

[34] V. S. Iyengar. On detecting space-time clusters. In *Proc. KDD*, pages 587–592, 2004.

[35] C. Jackins and S. L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Comput. Gr. Image Process*, 14(3):249–270, 1980.

[36] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proc. VLDB*, pages 768–779, 2004.

[37] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective density queries on continuously moving objects. In *Proc. ICDE*, page 71, 2006.

[38] C. S. Jensen and S. Saltenis. Towards increasingly update efficient moving-object indexing. *IEEE Data Eng. Bull.*, 25(2):35–40, 2002.

[39] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preserving anonymity in location based services. *Technical Report TRB6/06*, 2006.

[40] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *Proc.SSTD*, pages 364–381, 2005.

[41] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 369–380, 1997.

[42] G. Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. In *Proc. Design Automation Conference*, pages 352–357, 1982.

[43] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Proc. PODS*, pages 261–272, 1999.

[44] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proc. VLDB*, pages 34–145, 1995.

[45] D. Kwon, Sa. Lee, and Su. Lee. Indexing the current positions of moving objects using the lazy update. In *Proc. MDM*, pages 113–120, 2002.

[46] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *Proc. VLDB*, pages 608–619, 2003.

[47] P. Lehman and S. Yao. Efficient locking for concurrent operations on b-trees. *TODS*, pages 6(4):650–670, 1981.

[48] S. T. Leutenegger and M. A. Lopez. The effect of buffering on the performance of r-trees. In *Proc. ICDE*, pages 164–171, 1998.

[49] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *Proc. KDD*, pages 617–622, 2004.

[50] D. Meagher. Geometric modeling using octree encoding. *Comput. Gr. Image Process*, 19(2):129–147, 1982.

[51] R. P. Minch. Privacy issues in location-aware mobile devices. In *Proc. HICSS*, page 50127.2, 2004.

[52] M. F. Mokbel, C. Y. Chow, and W. G. Aref. The new casper: Query processing for location services without compromising privacy. In *Proc. VLDB*, pages 763–774, 2006.

[53] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[54] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. ACM SIGMOD*, pages 623–634, 2004.

[55] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad. Place: A query processor for handling real-time spatio-temporal data streams. In *Proc. VLDB*, pages 1377–1380, 2004.

[56] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. In *Proc. STDBM*, pages 57–64, 2004.

[57] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. *Geoinformatica*, 9(4):343–365, 2005.

[58] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *TKDE*, 13(1):124–141, 2001.

[59] M. A. Nascimento and J. R. O. Silva. Towards historical r-trees. In *Proc. ACM Symposium on Applied Computing*, pages 235–240, 1998.

[60] S. Nassar, J. Sander, and C. Cheng. Incremental and effective data summarization for dynamic hierarchical clustering. In *Proc. ACM SIGMOD*, pages 467–478, 2004.

[61] B. C. Ooi, K. L. Tan, and C. Yu. Fast update and efficient retrieval: an oxymoron on moving object indexes. In *Proc. of Int. Web GIS Workshop, Keynote*, 2002.

[62] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proc. SSTD*, pages 443–459, 2001.

[63] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proc. ICDE*, pages 301–312, 2004.

[64] D. Papadias and Y. Tao. Range aggregate processing in spatial databases. *TKDE*, 16(12):1555–1570, 2004.

[65] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *Proc. ICDE*, pages 166–175, 2002.

[66] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proc. VLDB*, pages 802–813, 2003.

[67] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *Proc. ACM SIGMOD*, pages 637–646, 2004.

[68] D. Pfoser and C. S. Jensen. Querying the trajectories of on-line mobile objects. In *Proc. ACM Int. Workshop on Data Eng. for wireless and mobile access*, pages 66–73, 2001.

[69] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proc. VLDB*, pages 395–406, 2000.

[70] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. Star-tree: An efficient self-adjusting index for moving objects. In *Proc. ALENEX*, pages 178–193, 2002.

[71] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *Proc. VLDB*, pages 263–272, 2000.

[72] K. R. Rao and P. Yip. Discrete cosine transform: algorithms, advantages, applications. *Academic Press Professional*, page 490, 1990.

[73] M. Reiter and A. Rubin. Crowds:anonymity for web transactions. *ACM Trans. On Inform. and Sys. Security*, 1(1):66–92, 1998.

[74] T. Roos. Dynamic voronoi diagrams. *PH.D. Thesis, University of Wurzburg, Germany*, 1991.

[75] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 71–79, 1995.

[76] S. Saltenis, C. S.Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD*, pages 331–342, 2000.

[77] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, 1984.

[78] H. Samet. The design and analysis of spatial data structures. *Addison-Wesley, Reading*, pages 102–103, 1990.

[79] B. Seeger and H. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. VLDB*, pages 590–601, 1990.

[80] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r^+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, pages 507–518, 1987.

[81] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. ICDE*, pages 422–432, 1997.

[82] E. Snekkenes. Concepts for personal location privacy policies. In *Proc. ACM EC*, pages 48–57, 2001.

[83] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proc. MDM*, pages 161–172, 2001.

[84] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proc. SSTD*, pages 79–96, 2001.

[85] V. Srinivasan and M. J. Carey. Performance of b-tree concurrency control algorithms. In *Proc. ACM SIGMOD*, pages 416–425, 1991.

[86] M. Tamminen. The excell method for efficient geometric access to data. *Acta Polytech. Scand. Mathematics and Computer Science Series No. 34*, 1981.

[87] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proc. ACM SIGMOD*, pages 611–622, 2004.

[88] Y. Tao and D. Papadias. Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In *Proc. VLDB*, pages 431–440, 2001.

[89] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. VLDB*, pages 287–298, 2002.

[90] Y. Tao, D. Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pages 790–801, 2003.

[91] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *TKDE*, 16(10): 1169–1184, 2004.

[92] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[93] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proc. ICMCS*, pages 441–448, 1996.

[94] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proc. ICDE*, pages 516–523, 1996.

[95] O. Wolfson, B. Xu, S. Chamberiain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. SSDBM*, pages 154–165, 1998.

[96] Y. Xia and S. Prabhakar. Q+rtree: Efficient indexing for moving object data bases. In *Proc. DASFAA*, pages 175–182, 2003.

[97] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proc. ICDE*, pages 643–654, 2005.

[98] M. Yiu, Y. Tao, and N. Mamoulis. The b^{dual}-tree: Indexing moving objects by space-filling curves in the dual space. *VLDB Journal*, 2006.

[99] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *Proc. ACM SIGMOD*, pages 443–454, 2004.

[100] C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proc. VLDB*, pages 421–430, 2001.

[101] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Spatial queries in the presence of obstacles. In *Proc. EDBT*, pages 366–384, 2004.