ON OPTIMIZING MOVING OBJECT DATABASES

SU CHEN

(Bachelor of Science)

Fudan University, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Supervisor: **BENG CHIN OOI**

School of Computing

Department of Computer Science

National University of Singapore

2012

Abstract

Recent advances in positioning technologies and wireless communications lead to a proliferation of location-based services. The moving-object database is a specialized database system for efficiently storing and processing the location data in location-based services. The dynamic nature of objects introduces new challenges to existing database techniques, especially dealing with the frequent location updates. Given the massive number of GPS-equipped mobile devices and the spectacular growth rate today, it is of vital importance to consistently improve the performance of moving-object databases.

In this dissertation, we exploit the possibility of enhancing the performance of moving-object databases from various aspects. As a preliminary, we propose a benchmark for evaluating moving-object indexes and conduct a comprehensive study on state-of-the-art moving-object indexes. Based on the strengths and drawbacks of existing indexes revealed by the study, we design the ST^2B-tree—an index for moving objects that can automatically adjust itself to adapt to workload changes in moving-object databases. We also present an adaptive updating mechanism to minimize the updating workload in moving-object databases, without affecting the query accuracy. The results of extensive performance study show that the proposed techniques take one step further towards optimizing the performance of moving-object databases.

Acknowledgements

First, I would like to express my deepest thanks to my supervisor Prof. Beng Chin Ooi. I sincerely appreciate his guidance, patience and encouragement, which helped me survive all challenges, pains and even desperation during the period of my candidature. I would say that I was a kind of person who gives up easily when I feel something that is beyond my ability. Without Prof. Ooi urging me forward, I could have quitted for a couple of times. Sometimes, the pressure was hard to bear, but the result turns out to be good. Although very serious and strict in research, Prof. Ooi is easy to get along with in life. He likes to dine, play sports with us and cares about the lives of his students. He is not only a supervisor, but also an elder or I would say even a friend of me.

I also want to say thank you to our professors of SoC. I am sincerely grateful to Prof. Chee Yong Chan and Prof. Mong Li Lee for their advices on my thesis. As my thesis advisory committee members, they both give me valuable guidance from the very beginning of my PhD to the composition of my thesis. I am deeply appreciative of Prof. Kian-Lee Tan and Prof. Anthony K. H. Tung for their help and suggestions on my research.

I would like to thank Divesh Srivastava, Luna Dong Xin and Laks V.S. Lakshmanan. While working with them, I learnt a lot from them, which I found invaluable in my subsequent research. I want to express my special

thanks to Dr. Divesh for taking me as an intern in AT&T labs. The six months memorable days in New Jersey was a great experience to me.

I am sincerely indebted to Prof. Christian S. Jensen and Prof. Mario. A. Nascimento. It was my great honor to work with them as a junior PhD candidate. I am always thankful that they lead me to walk the first step of my research, which is also the hardest step.

My senior fellows Xiaoyan Yang, Zhenjie Zhang, Yuan Ni, Linhao Xu, I am so appreciative to the help and care they give to me on both my research and on my life. They always take care of me like my elder sisters and brothers. My colleagues, Sai Wu, Yu Cao, Dongxiang Zhang, and lovely junior fellows, Shanshan Ying, Yanyan Shen, Meiyu Lu, Meihui Zhang, Xiaoli Wang, Peng Lu, Feng Li, Xuan Liu, Jingbo Zhang and everyone else in my lab, there are so many of you that I cannot name you all. Thank you all for your accompany. I will always remember the joyful and bitter days we spent together. Without you all, the PhD life would be quite boring!

I am always grateful to my long-term house-mate and best friends, Xianjun Wang, Yingyi Qi, Shaojie Zhuo and Dong Guo. My dear Xianjun and Yingyi, we have known each other for more than 11 years. You are my true sisters for life. Although I have never spoken out, I am always appreciative for your tolerance on my bad temper and innocent behavior. Without your accompany, I would not have the courage to come to Singapore alone. Shaojie and Dong, although we did not so familiar before we came to Singapore, we became a family since the first day we came here. I would always remember the day when we came to Singapore and started our PhD study together. We all overcame the hardness of PhD study and now I am so happy that finally all of us get the degree together.

Last but not least, I would like to express my gratitude and love to my dear parents. My dear mum and dad, without your consistent support and love, I would definitely not be able to make it. I know that I am easily losing my tamper when I feel stressed. So every time when I got into any difficulties, I took it out on you, as I know you would never get angry with me. When I felt upset or depressed, your voice was the most effective cure, which brought me inspiration and courage. I always felt too embarrassed to speak out. Here, I want to say, I love you and thank you, my dear parents.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

With recent breakthroughs in positioning technologies and wireless communications, Location-Based Service (LBS) becomes an emerging industry. A LBS makes use of the geographical positions of mobile users and provides the users with useful information regarding their positions. Most of the present LBSs, such as navigation and mobile advertising, are closely related to our daily lives. LBSs hence gain popularity quickly and proliferate at an amazing rate. Naturally, how to manage location data of mobile clients and provide better support to LBSs soon becomes a hot topic in database community and has received special attention in database research. The mobile users, as the core of LBSs, are abstracted as moving objects in database terminology. In this chapter, we first give an overview of moving-object management. Then, we describe the challenges in moving-object databases comparing to traditional database systems, followed by a brief review of state-of-the-art techniques in moving-object databases.

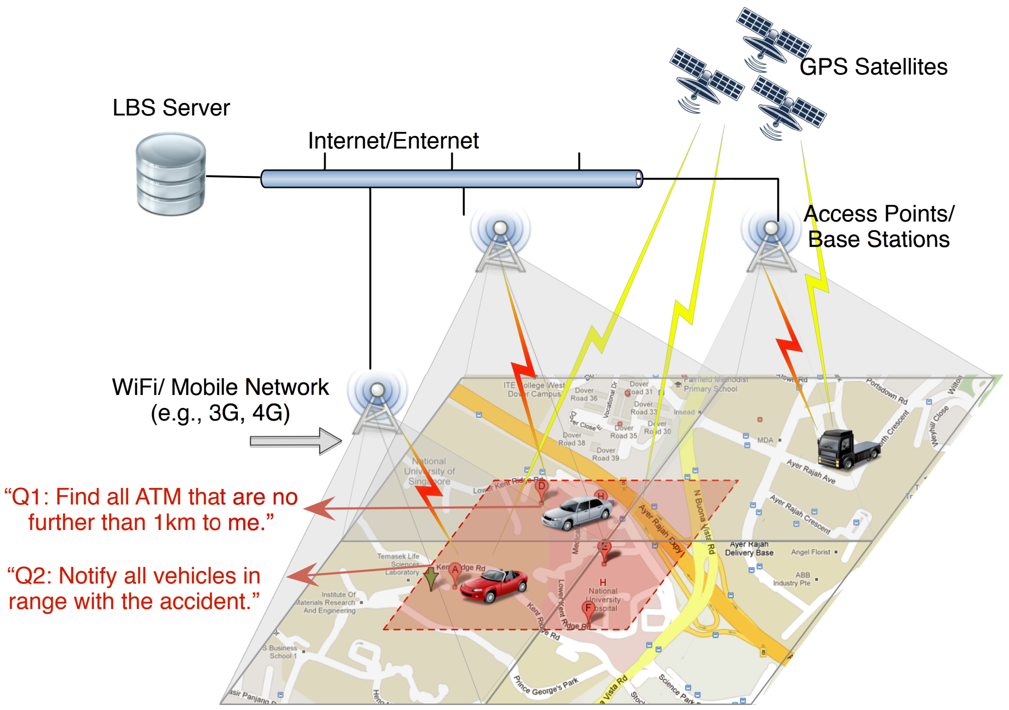

GPS Satellites
LBS Server
Internet/Enternet
Access Points/ Base Stations
WiFi/ Mobile Network (e.g., 3G, 4G)
"Q1: Find all ATM that are no further than 1km to me."
"Q2: Notify all vehicles in range with the accident."

Figure 1.1: A general framework of a location-based service

In recent decades, we have witnessed the rapid proliferation of *Location-Based Services* (LBS) [85, 90]. In the 1990s, location-based services, such as traffic control and management systems, are mainly restricted for government usage. In the 2000s, thanks to the advances in technologies, location-based services have branched out into personal and everyday usages, ranging from navigation, taxi calling, and mobile advertising to logistics management. Modern positioning techniques, such as GPS (*Global Positioning System*) and RFID (*Radio-Frequency Identification*), make it possible to sense the current location of an object. Nowadays, more smart devices, e.g., the iPhone, car navigators and even digital cameras, are equipped with GPS receivers, making their locations self-perceivable. The ubiquitous wireless communications, including the cellular network, 3G and WiFi, build up the communication channels between location-based service providers and these mobile devices.

Figure 1.1 illustrates a general framework of systems that provide location-based services. The mobile clients, e.g., automobiles, get their current locations from the GPS satellites and send their locations to the LBS server by WiFi or 3G network via wireless access points or base stations in cellular network respectively. To answer location-based queries such as "find all ATM that are no further than 1km to me" and "notify all vehicles that are within 1km to the place where an accident happened" efficiently, a database is used for storing and retrieving the location data.

Although there are a number of general-purpose database management systems, such as Oracle, IBM DB2 and Microsoft SQL Server, they are designed for serving as many, diverse applications as possible. However, they may not be the optimal solution for special applications, such as the LBSs. A wide variety of specialized databases are proposed to meet the requirements of different applications, e.g., the spatial database for *Geographical Information System* (GIS). Considering the geometric characteristics of moving objects, spatial databases were used for managing location data in moving-object applications in the early 1990s. However, since spatial databases are designed for managing static geographical data, e.g., lines and polygons, it is hard for them to capture the mobility of moving objects efficiently, i.e., the continuous change of objects' locations in LBSs. As a result, *Moving-Object Database* (MOD) [114, 40] was introduced in the mid-1990s, exclusively designed for managing moving objects in LBSs.

In the rest of this chapter, we describe the challenges in moving-object management, followed by a brief review of state-of-the-art technologies and research topics in moving-object databases. At the end of this chapter, we summarize the contributions and present an outline of the thesis.

1.1 Challenges in Moving Object Management

The problem of moving-object management has attracted great enthusiasm from the database researchers. For decades, researchers have worked consistently on enhancing the scalability of the database system, i.e., the amount of workload the system can handle. The workload of a database system consists of two parts, the updates and queries. Traditional databases are designed and optimized for relatively static data, for which updates are infrequent compared to the queries.

Moving-object databases, on the other hand, are proposed specially for managing moving objects, whose locations change continuously over time. To track the objects precisely, objects are required to inform the system about any change of their locations, introducing heavy updating workload to the database system. The high frequency of updates on the data is a distinguishing feature that differentiates the moving-object database from traditional databases, where data are assumed to be constant and are updated very occasionally.

With the ubiquitous GPS-equipped devices, the number of traceable mobile clients of LBSs increased rapidly in the 2000s. There were roughly 175 million handsets using the GPS worldwide in 2007, and the number would increase to 560 million in 2012. Analysts have predicted the number of GPS-enabled handsets will be set to more than triple during the next five years. Given the incredibly large number of mobile users and the sustainable growth rate today, the traditional databases cannot scale up with the increasing number of moving objects. Designed for static data, traditional databases concern more on the query processing than the updates. Traditional databases show their inadequacy of dealing with frequent updates from a large number of moving objects. The capability of dealing with such frequent, enormous updates is the primary consideration in the design of moving-object databases.

Besides the scalability of the system, the quality of service (QoS) is another major concern in moving-object databases design. Compared with traditional databases, minimizing the query response time is even more important in moving-object databases. Since the locations of objects change continuously, the answer to a query can be transient and become out-of-date easily, especially when objects move at a high speed. Therefore, providing immediate query response is another requirement of utmost importance in moving-object databases.

1.2 Research in Moving-Object Databases

In this section, we briefly examine state-of-the-art technologies in moving-objects databases and describe some popular research topics.

1.2.1 Updates in Moving-Object Databases

As objects move, their locations change continuously over time. One fundamental problem in moving-object databases is how to track these dynamic locations. Intuitively, an update is required whenever an object changes its location. Regardless of the positioning error or the transmission time between the object and the database, this simple but strict updating protocol leads to a 100% precision. However, this simple updating protocol will produce unacceptably heavy updating workload to the database system.

By contrast, the number of updates can be reduced by easing the trigger condition of an update. For example, updates are required every 30 seconds or every 100 meters. With less frequent updates, the error between the exact location and the location known by the database increases. A lower tracking precision results in a higher inaccuracy in the answers to queries of the moving-object databases, and affects the quality of location-based service eventually.

Therefore, an efficient updating protocol is essential to achieve a trade-off between the tracking precision and the amount of updates. Considering that minimizing the updating workload and maximizing the tracking precision are two opposite goals, the current design of updating protocol follows the principle that the number of updates should be minimized on condition that the tracking error (or the query inaccuracy) does not exceed a specific threshold.

In moving-object research, there are in general three types of updating protocols, namely the time-based updating protocol (e.g., update every 30 seconds), the distance-based updating protocol (e.g., update every 100meters) and the deviation-based updating protocol. The deviation-based updating protocol makes predication on object's location, an update is required when the distance between the predicted location and the exact location exceeds a given threshold ϵ (e.g., ϵ=100m). With the same threshold on the tracking error, the deviation-based updating protocol usually leads to the lightest updating workload, making it the most-adopted protocol in the literature. A detailed review of these updating protocols is presented in Section 2.2.

1.2.2 Indexes in Moving-Object Databases

The index is the key component in any database system for speeding up the retrieval of a large amount of data. It is even crucial in moving-object databases. A great number of indexing techniques have been proposed for moving-object databases exclusively. A short survey on existing moving-objects is provided in Section 2.3.

Considering the peculiarities of moving-object data, there are generally two major concerns in the design of moving-object indexes: (1) how to extend existing indexing structures to deal with dynamic location data; (2) how to improve the update efficiency of the index.

First, consider that data stored in moving-object databases are spatio-temporal data, i.e., continuous changing locations. Because the locations are spatial data points in nature, it is intuitive to use existing spatial indexes such as the R-tree [41] and Quadtree [88] for indexing moving-object data directly. Then, the problem is how to combine the spatial location and the corresponding time information into one single index. A representative solution to this problem is the TPR-tree [87], which is one of the most noted moving-object indexes. The TPR-tree introduces the *Time-Parameterized Bounding Rectangle*, consisted of bounding rectangles on objects' locations and velocities respectively. A *Minimum Bounding Rectangle* (MBR) is a rectangle that encloses all possible locations of objects at any given point of time, derived by linear interpolation on the location and velocity bounding rectangles. As a result, the R-tree's update and query algorithms can be applied directly to moving objects, by using the MBR at corresponding time as the *Bounding Rectangle* (BR) in the original R-tree.

On the other hand, despite of the remarkable improvement in the speed of data retrieval, indexes also require additional storage space and introduce additional overhead on database updates. Traditional databases are optimized for relatively static data, and improving the efficiency of query processing is the primary concern in the design of the indexes. Reducing the overhead on the storage and updates is the secondary consideration.

However, due to the dynamic nature of moving objects, updates are more frequent in moving-object databases, comparing to they are in traditional databases. The additional cost on updates cannot be ignored. As a result, traditional indexes show deficiencies in such update-intensive applications. It turns out that updates in the R-trees are not efficient enough to catch up with the frequent updates in moving-object applications [50, 20]. Consistent efforts have been made to improve the update efficiency of the R-tree, resulting in indexes such as the LUR-tree [57], the RUM-tree [119] and the bottom-up R-tree [57]. Other works, e.g., the B^x-tree [50] turn to other simple but mature indexing structures, such as the B$^+$-tree and the grid file, for better update performance. For the purpose of minimizing the updating workload, there are also another branch of works [82, 54] that build indexes of the queries instead of the objects. These indexes help retrieve queries that are affected by an object update and the queries are updated accordingly.

1.2.3 Other Research Topics in Moving-Object Databases

Besides the numerous indexing techniques for supporting efficient updates and queries, existing research covers a variety of aspects in moving-object databases.

Modeling and storage: As a fundamental problem, different models of representing moving objects in the database and the storage paradigm in the moving-object database have been proposed in [92, 114, 33, 82, 5, 45, 24, 39].

Updating protocol: The adoption of updating protocol is essential to the moving-object database. The tracking precision is higher when the updates are more frequent. Various updating protocols have been developed in [111, 113, 112, 58], with the purpose of minimizing the number of updates while keeping a high tracking precision.

In-network moving objects: While most of the existing works assume that objects move freely in Euclidean space, a number of other works [34, 27, 39] consider objects whose movements are constrained by a fixed road-network. The road network is utilized to improve the storage of objects and query processing on them.

Distributed query processing: Most of the existing works assume that there is only one single database server. Considering the limited computational capacity and communication bandwidth of a single server, works in [14, 35, 108, 116] exploit the possibility of processing queries distributedly. By making use of the computational capability of the mobile devices; objects not only report their locations to the database server but also collaborate with the server on processing the queries. Other works [6, 44] study moving-object management in a P2P (Peer-to-Peer) network, where a set of servers are used to manage all objects together.

Uncertainty: Due to the imprecision in positioning techniques and the latency in wireless communications, imprecision is an inherent characteristics of moving-object databases. In addition, the updating protocol adopted in moving-object databases can reduce the amount of updates, at the cost of increased imprecision of location tracking. A bunch of works [81, 105, 24, 68, 67] are dedicated to the study of the imprecision (i.e., uncertainty) of objects' locations and query answers, using probabilistic approaches.

Privacy: Protecting the privacy of mobile clients becomes a hot issue in location-based services recently. The k-anonymous and data cloaking techniques are incorporated into moving-object databases to prevent the service provider from inferring the identity of mobile client or the real location of the client [36, 65, 66].

Data mining: Works in [63, 43, 64] perform data mining tasks in moving-object databases, designing indexing structures and algorithms for discovering interesting patterns on objects movements or aggregation information on objects' locations, which are helpful to location-based services such as the navigation system and the traffic management system.

1.3 Contributions of the Thesis

As mentioned, a large number of indexes have been proposed in literature to enhance the performance of moving-object database systems. However, each of these indexes claimed in its original proposal that it was capable of outperforming the others (i.e., its predecessors). This makes it difficult to know the advantages and disadvantages of these indexes; and it is even harder for the potential users of the indexes to make a decision on which index is the best suited for a specific application.

In addition, existing works on moving-object indexes focus either on the design of indexing structures or on the development of efficient algorithms for various kinds of queries. Variability in data workload, i.e., cardinality and distribution of objects, has so far been overlooked in the design of moving-object indexes. Such data variability has a significant impact on the performance of the indexes. It is important for moving-object indexes to be adaptive to such variability.

On the other hand, comparing to the numerous indexes having been proposed in the literature, little attention has been paid to other techniques such as modeling the objects and the updating protocols. While the index plays a crucial role in enhancing the efficiency of database operations, the object model and the updating protocol have a joint effect on the updating workload of the system, i.e., the number of updates. With the purpose of improving the performance of moving-object databases, such techniques are of equal importance to or even more important than the indexes.

This thesis investigates the problem of performance optimization in moving-object databases from various aspects. Considering the indexing techniques, the thesis aims to establish a comprehensive study on existing moving-object indexes and design an index that is adaptive to the data variability in moving-object applications. The thesis also explores the possibility of performance optimization from other fundamental aspects of moving-object databases.

Specifically, the contributions of this thesis are as follows.

- First, we propose a benchmark for evaluating moving-object indexes. The proposed benchmark covers a series of carefully generated datasets, a broad variety of workloads, and a standard evaluation procedure. It covers important aspects of moving-object indexes that have not previously been covered by any benchmark. The results of the benchmark study can elicit the characteristics of existing indexes and offer input to future index development.

- We present a new moving-object index, called ST^{2}B-tree. The design of the ST^{2}B-tree considers two aspects: (1) the feasibility of tuning the index; (2) the advantages and drawbacks of existing indexes, as revealed in our benchmark study. We also introduce an online tuning framework for the ST^{2}B-tree. Although specially designed for the ST^{2}B-tree, the tuning framework is applicable to existing and future indexes in the same category as the ST^{2}B-tree. With the online tuning framework, the resulting indexes can adapt to the variability in the moving-object environment.

- We design an STSR-based updating protocol for moving-object databases. The new updating protocol can largely reduce the updating workload of the database, by relaxing the tracking accuracy with care. The STSR-based updating protocol guarantees the quality of query answers by pulling passive updates from objects whenever necessary.

1.4 Outline of the Thesis

The remainder of this thesis is organized as follows. First, the next chapter presents an exhaustive review on existing techniques in moving-object databases, focusing on those that are closely related to this thesis. Chapter 3 presents a benchmark for evaluating the performances of moving-object indexes and provides a thorough study on several representative indexes. Based on the findings of the comparative study of existing indexes, Chapter 4 introduces the ST^{2}B-tree as well as the framework for tuning the ST^{2}B-tree online. Next, Chapter 5 introduces the STSR-based updating protocol on reducing the workload of moving-object databases. Finally, Chapter 6 concludes the thesis and discusses some topics for further research.

Chapter 2

Literature Review

Moving-object management is a well-studied topic in database community. Tremendous research efforts have been put into this area in last decades, involving almost every aspect of moving-object databases. In this chapter, we review several essential techniques in moving-object databases, especially those related to this thesis. In particular, we first introduce the models for preparing the objects for database storage and processing. Next, we review existing updating protocols for tracking objects efficiently. Finally, we survey state-of-the-art indexing and query processing techniques in moving-objects databases.

2.1 Modeling Moving Objects

In moving-object management, the first issue to solve is how to represent location data of objects in the database. While few works exist that study this topic exclusively [92, 114, 33, 5, 45, 39], there are mainly two approaches of modeling moving objects in the literature.

2.1.1 Objects as Static Spatial Points

The first model simply treats moving objects as other general types of data in traditional databases. By neglecting the kinetic feature, moving objects are simply represented as spatial points, i.e., multi-dimensional data points in their space of movement, and the database system stores the last-known locations of objects. In particular, an object moving in the x-y plane is represented as a tuple $\langle oid, \overrightarrow{p} \rangle$, where oid is the identifier of the object, $\overrightarrow{p} = \langle p_x, p_y \rangle$ are the coordinates of the object's location.

As in any traditional database, the data stored remains constant unless explicitly updated. In the context of moving-object databases, this means that an object is assumed to stay at the stored location unless it reports a location update to the database. With this simple model, objects are handled in an ad-hoc mode. From the database's perspective, the movement of an object is not continuous. Instead, the object jumps from one location to another at discrete points of time.

This model is quite naive and straightforward. With this simple model, existing DBMSs can be used directly for managing moving objects without much effort. It gains benefit by re-using the mature and comprehensive techniques on traditional database, ranging from indexing, query processing to transaction management and etc. For this reason, this model gains popularity in a bunch of works [72, 120, 125], on processing queries on current locations of objects.

On the other hand, since the movements of objects are continuous, their locations keep changing all the time. The locations stored in the database are very likely to be obsolete after a short duration of time. To keep the data stored in the database up-to-date, objects are required to report their "current" locations at a high enough frequency, especially when they are moving fast. As a result, in order to retain a given degree of quality of service, this simple mode introduces high updating workload as well as communication overhead between objects and the database server. Although traditional DBMS performs well in managing static data that are not updated often, it cannot handle well highly dynamic data from moving objects.

2.1.2 Objects as Time-Parameterized Functions

Unlike the first model that does not distinguish moving objects from other traditional data, the second approach models moving objects as functions of time, making use of the patterns beneath objects' movements. In general, the location of an object at any time t is abstracted as a function $\overrightarrow{p}_t = f(t)$, and the value f changes with the time t. For each object, the database keeps the coefficients of its motion function. Given a point of time t', the database can always derive the location of the object at t'.

Take the linear function as an example. A moving object in the x-y plane is represented by a tuple $\langle oid, \overrightarrow{p}, \overrightarrow{v}, t_{up}\rangle$. As in the first model, oid and $\overrightarrow{p}$ are the identifier of the object and tis location at time t_{up}. $\overrightarrow{v} = \langle v_x, v_y\rangle$ denotes the velocity vector at t_{up}, containing the coordinates of the object's speed in corresponding dimension . Then, the location of the object at any point of time t' can be derived from the linear function:

$\overrightarrow{p}_{t'} = f(t') = \overrightarrow{p} + (t' - t_{up}) \cdot \overrightarrow{v}$.

By modeling objects as time-parameterized functions, updates are required only when the motion functions change, e.g., the velocity changes (either the direction or the speed) in the above linear function. While the location of a moving object changes all the time, the corresponding motion function may remain constant for longer time duration. If an object travels with a constant velocity, updates can be eliminated.

Compared with the location-only model, this approach reduces the number of updates significantly. In addition, this model preserves the continuity of object's movements: the trajectory of an object consists of conjunctive piece-wise motion functions in the database system.

Due to its simplicity, the linear function, as introduced in the example, is undoubtedly the most popular mathematical function adopted by most existing works [87, 80, 78]. However, considering the complexity and randomness of object's movement in practice, linear function may not perform well in capturing objects' motion in some circumstances. More complex and accurate functions such as the recursive motion function [95] and Chebyshev polynomials [15] are also introduced for modeling objects' movements.

Beside the significant reduction in the amount of updates, we gain other benefits from modeling objects as time-parameterized functions. By representing the location of an object as a time function, it becomes possible to predict the near-future location of the object, as long as the motion function remains un-changed in the near-future. This characteristic enables a group of future queries in addition to historical and present queries. Because of the importance of predictive queries in today's location-based services, this model is adopted by most of the works in the literature.

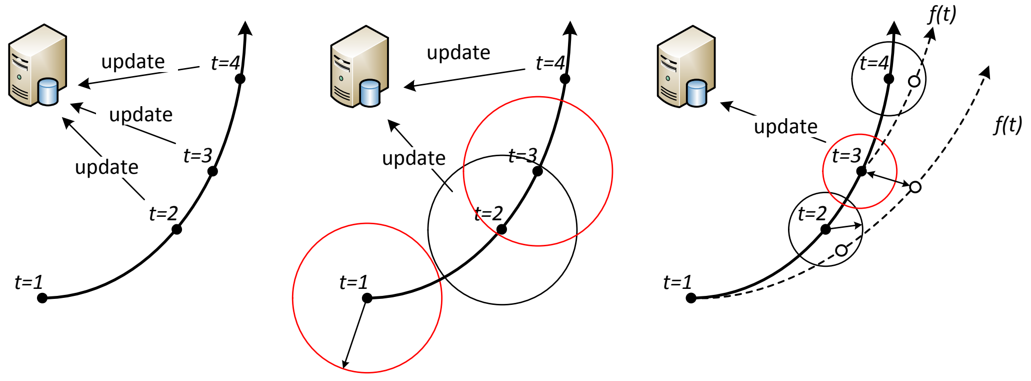
2.2 Tracking Moving Objects

Given the two object models, the next problem in moving-object databases is how to keep the locations of the objects up-to-date. On the one hand, for the purpose of maximizing the tracking precision, updates are required whenever object's location changes or motion function changes. On the other hand, in order to reduce the frequency of updates, the trigger condition of an update can be relaxed, however, at the expenses of tracking precision. Therefore, the design of a "good" updating protocol is important for maintaining the tracking precision while minimizing the communication costs between the objects and the database server [114, 115]. In this section, we review the existing updating protocols in moving-object databases [111, 113, 112].

2.2.1 Time-Bounded Updating Protocol

The first and the most simple updating protocol is called *Time-Bounded Updating Protocol*, where updates are issued periodically [59]. For example, updates are required every 30 seconds. With this method, the physical time is discretized into a serial of logical timestamps; an object sends an update to the database server every timestamp. This updating protocol is always used together with the location-only object model introduced in previous section. Figure 2.1(a) shows an example of the time-based updating protocol. In the figure, the solid line represents the real trajectory of an object, and the solid points on the line show the positions of the object at corresponding timestamps.

With the time-bounded updating protocol, updates must be frequent enough to keep a reasonable tracking precision. However, thanks to its



(a) Time-bounded (b) Distance-bounded (c) Deviation- bounded

Figure 2.1: Examples of updating protocols for tracking moving objects

simplicity, the time-bounded updating protocol is widely adopted in dealing with continuous queries on current object locations [120, 72, 121], where frequent query evaluation or maintenance is required.

2.2.2 Distance-Bounded Updating Protocol

Another traditional updating protocol that is typically accompanied by the location-only object model is the *Distance-Bounded Updating Protocol*. As the name suggests, this updating protocol requires an update from an object once the distance between its current location and the last-known location stored in the database exceeds a specified threshold. For example, updates are required every kilometer. As shown in Figure 2.1(b), from $t = 1$–4, two updates are required at $t = 2, 4$, since $dist(\overrightarrow{p_1}, \overrightarrow{p_2})$ and $dist(\overrightarrow{p_3}, \overrightarrow{p_4})$ are greater than the distance threshold, shown as the radius of the circles.

With the distance-bounded updating protocol, the frequency of updates depends on the velocity of object's movement and the distance threshold. The database system can achieve a trade-off between the tracking precision and updating frequency by tuning the distance threshold. The distance-bounded updating protocol is as simple as the time-bounded updating protocol, but provides an error bound on the tracking precision.

2.2.3 Deviation-Bounded Updating Protocol

Different from the previous two updating protocols, the *Deviation-Bounded Updating Protocol* is applicable only to the second object model where objects are represented as functions of time. With the time-parameterized motion function, it is possible to derive the location of an object at a given timestamp from its last update. In particular, an update is required only when the distance between its current location and the derived location exceeds a specific error bound. Figure 2.1(c) depicts an example of this updating protocol, where the dashed line represents the trajectory captured by the motion function (previously updated at $t = 1$). At $t = 3$, the distance between the exact location (the solid point) and the derived location (the hollow point) exceeds the spatial tolerance. With the deviation-bounded updating protocol, an update should be issued at $t = 3$.

The time-bounded and distance-bounded updating protocols are neither effective nor efficient if the objects are modeled as time-parameterized functions. For the time-bounded updating protocol, an update is meaningless if the function does not change at the scheduled timestamp; on the other hand, the tracking error cannot be bounded: for example, since the database system derives the location of the object at any timestamp before the next scheduled update from the last updated motion function, the distance between the derived location and the real location can be very large, if the motion function of the object changes dramatically right after the last update.

The deviation-bounded updating protocol decides the next updating time adaptively, depending on how significant the motion change is. The deviation-bounded updating protocol brings about significant decrease in the update frequency, while keeps the tracking error under control. Therefore, it is now the most widely adopted updating protocol in state-of-the-art moving-object databases [50, 87, 100, 123, 126].

2.2.4 Deviation-Based Updating Protocol for Predictive Queries

In [95], Tao et al. improve the basic deviation-bounded update protocol to support predicative queries better. Specifically, two motion functions are stored on both the object and the database server. The motion function at the object side is always in accordance with its current movement, which is supposed to be more accurate. At each timestamp, the object investigates the errors between the derived locations of the two functions at subsequent timestamps before a specific maximum predictive time. An update is issued if the error at any timestamp exceeds the deviation tolerance.

Continue with the example shown in Figure 2.1(c). The solid line (the dashed line resp.) can represent the motion model stored at the object side (the server side resp.). Suppose current time is $t = 1$ and the maximum predictive time is 1. No update is required as the error of two models at $t = 2$ is still tolerable. On the contrary, if the maximum predictive time is 2, an immediate update will be issued since the error of two models at $t = 3$ exceeds the threshold.

2.3 Indexing Moving Objects

Indexing is one of the most popular topics in database research for enhancing the performance over massive data. Given the large number of objects and updates, the index is undoubtedly the most essential component in moving-object databases and hence has attracted the most research efforts. In this section, we review existing indexes for moving objects.

2.3.1 A Taxonomy of Moving-Object Indexes

Mokbel et al. give a survey of indexing techniques introduced on or before 2003 [69], and follow up with a continuation containing an overview of indexes presented for the years 2003–2010 [76]. In [69, 76], moving-object indexes are classified into the following categories.

2.3.1.1 Indexing the Past

The first category of indexes deals with historical data, i.e., the locations and trajectories of moving objects in the past. Since objects are moving continuously, the historical data in a moving-object database keep increasing over time. Such indexes are necessary on answering queries, such as “where did Michael go yesterday?”, and applications that are interested in mining patterns from objects’ trajectories.

One intuitive way to index historical data is to group the data by time, i.e., to store object’s locations at the same time together in one spatial index, such as the R-tree [41] and the Quadtree [88]. In order to reduce the storage cost, indexes such as the MR-tree [122], the HR-tree [74] and the HR+-tree [96], overlapping quadtrees [106] incorporate the overlapping techniques in the B-tree[13] into the R-tree and the Quadtree, by sharing common nodes or entries among trees of consecutive timestamps.

While the temporal dimension takes priority in indexes such as the HR-tree, some other indexes consider the spatial dimension first. In particular, the space is first partitioned into small regions. Then, data inside each region are indexed in individual structure ordered by the time. For example, in the MTSB-tree [127], the space is partitioned into grid cells and a B-tree is maintained for indexing data inside each cell according to their update time. The FNR-tree [34] and the MON-tree [27] considers objects that travel inside a fixed road network. A 2D R-tree is used to index the line segments (roads or routes), and location or trajectory data are organized within each line segment in chronological order. The MV3R-tree [97] stores two R-trees on the same set of objects, one of which is optimized for queries on a given timestamp and the other is optimized for supporting queries over an interval of time.

Since spatial indexes such as the R-tree are designed for general multi-dimensional data, an alternative approach of indexing historical data is to take the time as another dimension(s) in additional to the spatial data of objects. For example, the 3D R-tree [104] stores objects in an R-tree with 3-dimensional key $\langle x, y, t \rangle$; in the RT-tree [122], objects are indexes by 6-dimensional key $\langle MBR, t_s, t_e \rangle$ in an R-tree, where MBR is the bounding rectangle of the objects' locations in time interval $[t_s, t_e]$; the STR-tree [80] extends the R-tree with insert/split algorithms optimized for clustering segments of the same trajectory together while preserving the spatial proximity of segments.

Another category of indexes [94, 109, 79] focus on supporting queries on objects' trajectories, such as "find all automobiles that traveled from the central park to the city hall during 10am to 11am yesterday?". Given the object's models, an object's trajectory is represented as a series of line segments, either by linear interpolation on consecutive locations or as a conjunction of its motion functions. The TB-tree [80] uses an R-tree to store all trajectory segments, but with the constraint that the segments inside a leaf node must belong to the same trajectory. SETI [16] partitions the space into regions and uses an R-tree to index segments inside each region, where cross-boundary segments are indexed in both R-trees.

2.3.1.2 Indexing the Current

Queries such as "notify all vehicles that are entering the neighborhood of a petrol station with the price drop" are the most common queries in location-based services. These queries require the database to keep track of the current location of objects. The second category of moving-object indexes are designed to answer these types of queries efficiently.

Hashing [93] is one of the earliest attempts to index current locations of objects. The space is partitioned into regions and objects are indexed with the hash value of the region it belongs to. Updates are issued only when objects move across regions. The hashing approach is extended by many other grid-based indexes [70, 72, 120], where the space partition is with the help of a uniform grid [77]. In order to reduce the I/O cost of updates, LUGrid (Lazy Update Grid) [121] maintains a memory buffer for incoming updates and flushes updates of the same disk page (i.e., the same grid cell) together.

Comparing to the grid, a larger number of indexes prefer the R-tree as the underlying structure. The R-tree can be used directly to index current locations of objects, as multi-dimensional data. However, it is well-known that updates in the R-tree are not efficient enough given the high frequency in MODs. Therefore, R-tree-based indexes try to improve the performance of updates. The LUR-tree (Lazy Update R-tree) [57] prevents the propagation of an update to upper level if the updated location is still covered by the MBR of the entry in the R-tree. [60] extends the LUR-tree and introduces several bottom-up update strategies in the R-tree to further reduce the update cost. In the RUM-tree [119, 91], an update inserts the new location into the R-tree, but does not delete the old record. Similar to the LUR-tree, the RUM-tree maintains an update memo in the memory of all obsolete entries. The old entries are purged periodically and in a batch mode.

2.3.1.3 Indexing the Current and Near-Future

The last category of indexes is designed for managing the near-future locations of moving objects. Since objects are moving continuously, a query asking about current locations of objects might become obsolete even before the results are sent back to the query issuer. Therefore, instead of the current locations of objects, queries in moving-object databases are more interested in objects' locations in the near future. In order to predict the future locations, objects are modeled as functions of time in all indexes in this category, typically the linear function.

Numerous indexes of this category have been developed in literature. Most of them are based on traditional spatial indexes such as the R-tree [41], the R*-tree [8] and the Quadtree [88].

The TPR-tree [87] first introduces the idea of time-parameterized bounding rectangles in the R-tree. The bounding rectangle of the R-tree can enlarge with time so that it can still enclose the objects' future locations. All the other R-tree-based indexes of this category are descendants of the TPR-tree, such as the TPR*-tree [100], the STAR-tree [83] and the R^{EXP}-tree [86]. These indexes try to improve the TPR-tree for better performance from different aspects in different circumstances.

The B^x-tree [50] is the first moving-object index using the B^+-tree [26] as the base structure. Although, B$^+$-tree is not designed for spatial or temporal data, a bunch of indexes including the B^x_r-tree [53], the B^y-tree and αB^y-tree [18], the B^{dual}-tree [123] are all built on top it, for its high updating efficiency. All of these indexes utilize the space filling curve to transform the multi-dimensional data of moving-objects into 1-dimensional value that can be indexed by the B$^+$-tree.

While the B^x-tree and TPR-tree families are the two prominent groups for indexing future locations of objects, there are other attempts on using other structures, such as the quadtree-based STRIPES [78], MOVIES [28] which is built on linearized kd-tree, and the Polar Tree [79] which takes the binary tree as the basic structure.

As predictive queries and indexing of near-future locations of moving objects are the focus of this thesis, we review indexes of this category in more detail in the following.

2.3.2 A Close Look at Indexes of Future Locations

In essence, a moving object is a multi-dimensional point in nature, whose coordinates keep changing over time. The design of a good index has two concerns: how to preserve an object's temporal information, i.e., the updating time, and how to cluster objects based on their location proximity as time elapses. In this section, we review existing structures of indexing objects' future locations from these two perspectives.

2.3.2.1 Preserve Temporal Information in the Indexes

To support predictive queries on future locations, objects are represented by time-parameterized functions. The database system stores the coefficients of the functions. However, most of the existing indexes do not store the last updated time for each object individually. As such, some mechanisms are required to get a synchronized view of object locations to enable predictive queries on the objects. There are in general three mechanisms.

Indexing with single reference time: In the first mechanism, the index has a global reference time t_{ref} and the motion functions of all objects stored in the index are mapped to the reference time. Take the linear function as an example. An object is represented as function $f(t') = \overrightarrow{p} + (t' - t_{up}) \cdot \overrightarrow{v}$, $\overrightarrow{p}$ and $\overrightarrow{v}$ are the location and velocity reported in the last update. Since the index does not store the updating time t_{up} for each object separately. The function is transformed to $f(t') = \overrightarrow{p} - (t_{up} - t_{ref}) \cdot \overrightarrow{v} + (t' - t_{ref}) \cdot \overrightarrow{v}$. Then, the index keeps the two coefficients $\overrightarrow{p} - (t_{up} - t_{ref})\overrightarrow{v}$ and $\overrightarrow{v}$ for the object instead of $\overrightarrow{p}$ and $\overrightarrow{v}$. The TPR-tree and its variants adopts this single reference time mechanism, where the reference time is set to the birth time of the index.

However, as shown in many recent works, e.g., [50, 20], indexes which organize objects with single reference time such as the TPR-tree suffer from severe performance degradation with time. Consider the case that two objects, which are stored in the same leaf node, have not been updated for quite a long time. The MBR of the leaf node could become quite large to enclose both objects. With increasing large MBRs, the overlap between upper level MBRs increases and the index performance deteriorates. Tao et al. have shown in [101] that such deterioration can be alleviated when the update frequency is high enough, i.e., 10% objects update in each timestamp. However, this frequency is too high that indexes such as the TPR-tree cannot afford. In addition, since the performance deterioration is mainly caused by those objects with low update frequencies, even a few such objects will degrade the performance.

Indexing with multiple reference times: On the other hand, indexes such as the B^x-tree, the B^{dual}-tree, the STRIPES have multiple reference times. The time axis is partitioned into intervals. Each interval is assigned a certain reference time, e.g., the staring time of the interval. Objects are distributed into different groups according to their last updating time and the transformation of the motion function is based on the corresponding reference time. An update moves the object from its old group into the group whose time interval covers the current updating time. Indexes of this category also use a single tree structure, e.g., B$^+$-tree and quadtree, to manage all objects. However, the underlying tree is divided into several logical sub-trees each of which manages objects of a group, i.e., whose updating time belongs to the same time interval. We call this method the "multi-tree" technique.

As a result of object updates, the "multi-tree" technique actually rebuilds the index periodically. All objects are required to update at least once within a given time interval T so that the sub-tree with the oldest time interval is empty, i.e., all objects have been moved to other sub-trees. As a result, the oldest sub-tree can be reused for subsequent updates. A range query searches all sub-trees with extended query regions. For each sub-tree, the query region is extended from the querying time to corresponding reference time using the maximum velocities of objects in the sub-tree.

Since T is the maximum update time interval of all objects, it could be quite large due to few infrequently updated objects. As a result, a query needs to search too many sub-trees with large extended query regions. In [18], the authors present a technique to improve the query performance at the expense of higher update cost, regarding the highly variable update frequencies. An object is inserted into multiple sub-trees with different reference times (rather than the one covering the updating time only). The number of sub-trees depends on the predicted time of the object's next update and a tunable parameter α, which is used to balance the update and query performance. Instead of searching all sub-trees, a query is performed on the α youngest sub-trees only. Query performance is improved as fewer sub-trees are searched with smaller extended query regions.

2.3.2.2 Preserve Location Proximity in the Indexes

Given the global reference time(s) and the coefficients of the motion function, we can predict locations of all objects at any point of time. Then, the second issue is how to cluster near-by objects together in the index to facilitate query processing. Considering this, existing indexes can be classified into two major categories: space partitioning and data partitioning indexes.

Space Partitioning Indexes: The space partitioning indexes adopt a prior and fixed partition on the physical space, i.e., the space where objects move. Such indexes typically partition the physical space in advance using a grid. An object is indexed by the cell it belongs to. Indexes in [70, 72, 120], utilize the grid index directly, while some other indexes such as those developed in [50, 123] use a B$^+$-tree on top of the grid. Each grid cell is assigned a unique id, and objects are indexed by the B$^+$-tree with the id of the cell they belong to. As space partitioning indexes split the space using a single uniform grid, the workload across different parts of the index may not be balanced. Such imbalance does impact the performance of the indexes, especially in the presence of skewed data.

Data Partitioning Indexes: On the other hand, the data partitioning indexes organize objects in dynamic partitions, based on the object distribution. Representative schemes in this category include the TPR-tree [87], the

TPR*-tree [100], the R^{EXP}-tree [86] and the STAR-tree [83]. All of these methods are based on traditional spatial indexes such as the R-tree [41] and the R*-tree [8]. In all R-tree-based indexes, at the bottom level, a leaf node accommodates up to a given number of objects that are close to one another. At higher levels of the hierarchical structure, each intermediate node contains up to a given number of entries, each of which contains a pointer to one of its children and the MBR (*Minimum Bounding Rectangle*) of the corresponding child. The MBR of a leaf is the smallest rectangle covering all the objects it contains. Similarly, the MBR of an intermediate node is the region which just covers the MBRs of all its children. A node split occurs when the number of objects/entries to be stored in a node exceeds its capacity. On the other hand, two neighboring nodes are merged into one if objects/entries of both nodes can be accommodated in only one node. As a result of splitting and merging, objects/entries are clustered into groups based on their proximity. Therefore, the regions in which objects are crowded always consist of many small (leaf) MBRs. Sparse regions, on the contrary, are typically covered by a few large MBRs. By comparison, data partitioning indexes are typically less susceptible to data diversities and changes as a result of MBR merging and splitting.

However, in [20], the authors compare several state-of-the-art indexes experimentally and the result shows that space partitioning indexes outperform data partitioning indexes in most cases, especially on the update performance. In general, space partitioning indexes surpass their counterparts that are based on data partitioning in two ways. First, both the grid index and the B^{+}-tree are well established indexing structures present in virtually every commercial DBMS. The index can be integrated into an existing DBMS easily. No fundamental (lower level) changes are needed for the underlying index structure, concurrency control or the query execution module of the DBMS. Second, in comparison with spatial indexes such as the R-tree, operations such as search, insertion and deletion on the grid index and the B$^+$-tree can be performed very efficiently. To keep the objects well organized, updates in the R-tree are quite complex. Guo et al. [37] have shown that the pre-processing and tree optimization strategies employed in the TPR*-tree [100] result in extra delay in locking, and hence reduce the performance gain in query processing due to the preprocessing during insertions. Other techniques, such as bottom-up update [57], lazy update [60] and update memo [119], have been developed in the literature to improve the update performance of the R-tree. However, although the update cost decreases with varying degrees, it is still higher than those space partitioning based indexes [20].

Another recent work, the STRIPES [78], is a hybrid of both space partitioning and data partitioning indexes. It utilizes the quadtree as the underlying index, so that the way of partitioning is fixed but guided by the data distribution. However, since the quadtree is an unbalanced structure, dense regions are partitioned into finer quads and stored deeply in the tree. Updates and queries on these objects always incur higher overhead.

2.3.2.3 Revisit of State-of-the-Art Indexes

We review six moving-object indexes to which we will later apply the proposed benchmark: the RUM-tree [119], the TPR-tree [87], the TPR*-tree [100], the B^x-tree [50], the B^{dual}-tree [101], and the STRIPES [78]. The TPR-tree is chosen since it is the predecessor of more than a dozen proposals for moving-object indexes. The other five indexes are representatives of indexes of different base structures.

2.3.2.4 R-Tree With Update Memo

Traditional spatial indexes such as the R-tree [41] and the Quadtree [32] were designed mainly with query efficiency in mind and implicitly assumed relatively static datasets.

However, in our problem setting, updates are very frequent due to the need of tracking continuous movements. To render indexing techniques more suitable for workloads with frequent updates, several techniques [57, 60, 11] have been proposed to improve their update performance. A recent representative of this line of work is the RUM-tree by Xiong et al. [119].

The RUM-tree introduces a main-memory memo that makes it possible to avoid disk accesses for deleting the old entry during an update. Therefore, the cost of an update is reduced to the cost of an insert. In particular, object updates are ordered temporally according to the processing time. By maintaining the update memo, more than one entry for an object may coexist. Obsolete entries are deleted lazily in batch mode. Garbage collection is employed to limit the percentage of obsolete entries in the tree and to control the size of the update memo.

Because the RUM-tree extends the R-tree, it indexes only point locations rather than linear functions of time. For the benchmark, we will therefore apply the memo-based update technique of the RUM-tree to the TPR*-tree described next.

2.3.2.5 The TPR-tree and TPR*-tree

Saltenis et al. [87] proposed the TPR-tree (Time-Parameterized R-tree) that augments the R*-tree [8] (a variant of the R-tree) with velocities to index linear functions of time. Specifically, an object is represented by its location as of (global) a reference time and its velocity vector. The sides of the bounding rectangles (BRs) employed are also functions of time, and the BRs are chosen so that they bound all the contained moving objects or BRs at any time in the future (including the current time).

If no updates occur on a TPR-tree, its BRs will expand, and query and update performance will deteriorate. When updates occur, objects are placed in the BRs that they now fit into, and BRs that have grown too much are tightened.

Tao et al. [100] have proposed the so-called TPR*-tree, which is a variant of the TPR-tree. The TPR*-tree uses the same data structure as the TPR-tree, but applies different algorithms for maintaining the index. In particular, these algorithms aim to optimize time-range queries rather than time-slice queries, as done by the TPR-tree. And while the TPR-tree makes decisions on where to insert an object on a level-by-level basis, the TPR*-tree puts more work into insertions and makes more global decisions. In addition, the TPR*-tree is more aggressive than the TPR-tree when it comes to the tightening of BRs—while tightening costs I/O, it may save I/O subsequently.

2.3.2.6 B^x-tree and B^{dual}-tree

It is well-known that the R-tree, as well as structures based on the R-tree, is prone to low update efficiency when compared to structures such as the B$^+$-tree. The problem is that BRs tend to overlap, which results in multiple (partial) paths from the root to the leaf level being explored during the deletion that occurs in connection with an update. This, serves as motivation for exploring B$^+$-tree-based techniques for the indexing of moving objects. Another source of motivation is the fact that B$^+$-trees are already supported widely in existing DBMSs, promising easier integration into existing systems of B^+-tree-based techniques.

The first B^+-tree-based index, called the B^x-tree, was proposed by Jensen et al. [50]. The B^x-tree adopts the time-parameterized function to model objects. A space-filling curve (e.g., the Peano or Hilbert curve) is applied to obtain a one-dimensional point from the location of an object at corresponding reference time. A partitioned B^+-tree is used where updated locations that occur at approximately the same time go into the same partition.

To support queries, original queries are subjected to transformations that counter the data transformations. These transformations involve query window enlargements that depend on the velocities of the objects indexed, the number of partitions, and the query time; a transformed query is created for each partition in the B^+-tree.

To reduce the adverse effects on performance of velocity skew and outliers on performance, Jensen et al. [53] equip the B^x-tree with more careful query enlargement algorithms.

Yiu et al. [123] have recently proposed the B^{dual}-tree, which aims to utilizes velocity information to obtain better query performance. The B^{dual}-tree uses a four-dimensional Hilbert curve to map both location and velocity vectors to one-dimensional point. One B^{dual}-tree is composed of two B^+-trees, and these two trees swap states every so-called maximum update interval (the maximum time duration between any pair of updates issued by any object). Each internal entry in the B^{dual}-tree is associated with a set of moving-object rectangles (MORs). Each MOR is square in size and has continuous Hilbert values corresponding to the keys stored in the sub-trees of the entry. These MORs can be treated as BRs as used in the TPR-tree, and hence the query algorithms of the TPR-tree can be applied to the B^{dual}-tree with minor modifications. However, this query algorithm is not based on that of the B$^+$-tree and is thus more difficult to be integrated into existing database systems.

2.3.2.7 The STRIPES

Similar to the B^{dual}-tree, the STRIPES [78], is also a dual index that indexes both location and velocity data. In particular, the STRIPES maps two-dimensional moving objects to four-dimensional points and indexes them by a PR bucket quadtree [89]. The arrangement promises efficient updates and less efficient queries because a node in the STRIPES may contain an arbitrarily small number of entries, meaning that relatively many pages need to be accessed to obtain a query result. The low page utilization also leads to large space consumption. To alleviate these problems, the authors suggest storing a leaf node with occupancy at most 50% in half of a page and leaf nodes with over 50% occupancy in a full page. However, such an arrangement not only complicates the concurrency control mechanism, but also makes the integration into a DBMS more difficult.

2.4 Querying Moving Objects

A wide variety of queries have been introduced in moving-object databases. In the following, we examine existing moving-objects queries in four different taxonomies.

2.4.1 A Classical Taxonomy

In general, all moving-object queries are derived from corresponding spatial queries, equipped with additional query predicate on the time dimension. For example, the range query of spatial data retrieves all (static) spatial objects within the query region. Its counterpart in moving-object database finds all (moving) objects that are inside the query region at some point of time or during some time interval. The classical taxonomy classifies moving-objects queries based on the spatial component of the query predicate.

2.4.1.1 Range Query

The range query is the most fundamental type of queries in spatial databases and moving-object databases, which retrieves objects inside a spatial region given in the query predicate. Typically, the query region is a rectangle represented by the lower-left and upper-right corners.

Processing range query on moving objects are the same as on other general multi-dimension data. To avoid scanning all objects, indexes are created to accelerate the search. As reviewed in Section 2.3.1, existing moving-object indexes derive from traditional indexes such as the B$^+$-tree, and the R-tree. The range query processing is either an extension of the original query algorithm on the base structure or based on some pre- and (/or) post-processing of the queries. For example, the TPR-tree [87] extends the R-tree range query algorithm to search over the time-parameterized bounding rectangles; in B^x-tree [50], a range query on the objects is decomposed into a set of disjoint range queries on the B$^+$-tree and the answer of the query is a combination of the answers to all transformed queries.

2.4.1.2 *k*NN Query

The *k-Nearest Neighbor* (*k*NN) query is another well-studied type of query in database research. A *k*NN query retrieves k objects that are closest to the location indicated in the query predicate.

Evaluation of the *k*NN query is more complex than that of the range query. A number of works have been proposed for processing kNN queries efficiently [9]. As for indexes derived from some spatial index, e.g., the TPR-tree [87], kNN query processing adopts the well-known branch-and-bound methodology. On the other hand, indexes such as the B^x-tree [50] process a kNN query as incremental range queries. The query processing algorithm first estimates the distance of the k'th neighbor and issues a range query to find all objects within the range. If fewer than k objects are returned by the range query, another range query of larger radius is executed, so on and so forth.

2.4.1.3 Other Types of Queries

Besides the two basic types of queries, many other query types have been introduced and investigated in the literature.

Reverse k-Nearest Neighbor Query: Given a querying object o, the *Reverse k-Nearest Neighbor Query* (RkNN) [10, 118, 56, 117, 30] finds all objects in the database of whom o is the nearest neighbor. An application of RkNN query is in a digital battlefield game. A player issues an R1NN query on the location of his team members. The answer to such query is the group of players who might ask the query issuer for help, as he is the one who is closest to these players.

In [10], Benetis et al. use the TPR-tree index the objects and facilitate the processing of kNN and RkNN queries. The 60°-property is utilized to prune unnecessary objects in the query processing. In particular, the space around the query point q can be partitioned into six 60° fan-shaped regions centered on q, and in each region only the nearest neighbor of q can be the NN of q. In this way, an RkNN query is reduced to a 6kNN query.

While Benetis et al. in [10] examine the problem of predictive RkNN queries, the authors in [118, 56, 117] are more interested in monitoring the answers of RkNN queries continuously. Kang et al. [56] define two types of RkNN queries—the monochromatic and bichromatic RkNN queries, and propose algorithms for monitoring the answers to both queries. Wu et al. [117] present a CRange-k method that enhances the pruning power in the query processing by maintaining a continuous range query around each query point q. A recent work [30] introduces a client-server scheme for processing RkNN queries, where the communication cost is the primary concern.

Close Pair Query and Closest Pair Query: The *Close Pair Query* [127] aims to find out all pairs of objects the distance between who are smaller than a given threshold ε. A typical application is "find out the airplanes that were closer to each other than 10 miles in last month". Such queries are helpful for air-traffic monitoring systems to arrange flight routes. The close pair query is an extension of the spatial join in spatial databases. In particular, processing queries of this type usually considers a self-join on objects' locations with $dist \leq \varepsilon$ as join condition.

Similar to the close pair query, the *k-Closest Pair Query* [25, 128] also investigates the distances between objects and tries to find the k closest pairs between moving objects and points of interests (POI), i.e., static spatial points such as the ATMs. A real-life example of the k closest pair query in ship pilotage system is "Monitor 10 pairs of sonar tracking stations and ships that are the closest to each other". Chung et al. [25] use a time function to represent the distance between an object and a POI and present an event-based structure to detect and index the changes of the order of the distances efficiently. Zhu et al. [128] propose a system schema that shifts part of the query processing job to the objects, for the purpose of reducing the communication costs between objects and the database server.

Density Query: The *Density Query* [42, 51] tries to figure out all regions where the object density exceeds a given threshold, e.g., “find all regions that contain more than 500 vehicles in the next hour”. Queries of this type are useful in traffic management applications, which are interested in finding out all jamming areas that have too many vehicles inside.

The density query is first introduced by Hadjieleftheriou et al. in [42]. However, it turns out that finding dense regions of arbitrary size is inefficient as for moving-object applications. Therefore, the authors simplify the general density query by partitioning the space into disjoint cells and finding dense cells instead. Jensen et al. [51] improve the original query processing algorithm in [42] to avoid answer loss. While both [42] and [51] study density queries on Euclidean space, assuming that objects are moving freely. Lai et al. [58] define the density query over a constrained road-network and present an algorithm of discovering dense regions of arbitrary size and shape in the network.

Skyline Query: The *Skyline Query* retrieves a set of data points that are not “dominated” by any other points. While most existing works on the skyline query consider static data, Huang et al. [47] and Chen et al. [19] investigate the skyline query in the context of moving objects. The skyline query is valuable in location-based recommendation system. For example, a tourist may ask for a list of near-by restaurants that are cheap but highly rated while he is traveling along the beach. In such case, the skyline may change continuously as the objects move.

In [47], the skyline is computed at the first timestamp and incrementally maintained using a kinetic data structure. An effective query processing algorithm is introduced exploiting the correlations between the search bound and the change of skylines. In [19], the authors propose two algorithms of processing skyline queries, named RBBS and TPBBS. The RBBS builds an R-tree for each skyline query. The R-tree takes the distance of the query to a spatial data point (e.g., the location of the restaurant) as one dimension and other static attributes (e.g., the price) as other dimensions. Then, a standard branch-and-bound algorithm can be applied on the R-tree to find the skyline. The TPBBS augments the TPR-tree with static attributes and applies the branch-and-bound algorithm on the TPR-tree to find the skyline. The TPBBS is proved to outperform the RBBS.

2.4.2 A Taxonomy from Temporal Perspective

The classical taxonomy introduced in previous section classifies moving-object queries from the spatial perspective. In this section, we classify moving-object queries based on the temporal component of the query predicate.

2.4.2.1 Querying the Past, Current and Future

Based on the time in the query predicate, queries in moving-object databases can be classified to pass, current and future queries.

Historical Query: The historical query is interested in the past locations or trajectories of objects, e.g., "where was Michael at 3pm yesterday".

Current Query: The current query retrieves the current locations of objects, e.g., "where is Michael right now?".

Predictive Query: The predictive query searches the future locations of objects, e.g., "where will Michael be after 5 minutes?".

Existing works aim to enhance the efficiency of the query processing with well-designed indexing structures and search operators on the corresponding index, as we reviewed in Section 2.3.1.

2.4.2.2 Snapshot, Window and Continuous Queries

According to the continuity of moving-object queries, moving-object queries can be classified into three types: the snapshot query, window query and continuous query.

Snapshot Query: The snapshot query retrieves objects' locations at a given point of time. "Find all vehicles in the central park at 10pm yesterday" is an example of snapshot historical range query.

In essence, a snapshot query is a traditional spatial query of corresponding type, but over the objects' locations at the time indicated in the query. Then, the query processing algorithms of the corresponding spatial queries are applicable directly as long as the requested locations are known. Existing works [87, 50, 78, 22, 123, 91] put a lot of effort into the design of indexing structures for efficient retrieval of objects' locations at a specific point of time.

Window Query: The window query contains a temporal predicate regarding an interval of time. For example, a corresponding window version of the aforementioned example can be "find all vehicles that passed by the central park from 10am to 11am yesterday".

An intuitive approach of processing a window query is to decompose it into a set of snapshot queries on consecutive points of time. However, this approach is not efficient especially when the time window is large. Based on this consideration, in most of the existing works on the window query [87, 98, 50], objects are represented by time-parameterized functions, and the query is evaluated only once based on objects' motion functions.

Continuous Query: The continuous query monitors the answers until the query issuer cancels the query explicitly, e.g, "notify all vehicles that are entering the neighborhood of a petrol station with the price drop".

Hu et al. [46] present a generic client–server framework for processing continuous queries in moving-object database. Kalashnikov et al. [55] builds an in-memory spatial index for the queries. Queries are re-evaluated at every timestamp; in the query evaluation, each object identifies the set of queries it is involved in with the help of the query index. Mokbel et al. [70] introduces the SINA—an improved query evaluation method. Queries and objects are stored in separate tables; query evaluation is performed as a spatial join between the two tables. To avoid the query re-evaluation at each timestamp, SINA presents an incremental evaluation technique. In particular, it uses a uniform grid to index both queries and objects. When an object moves from one cell to another, the answers of the queries in the two cells are updated accordingly. The incremental evaluation based on the grid index brings remarkable improvement on the query efficiency and is adopted by a number of subsequent works [120, 72, 118, 56, 117].

In these grid-based proposals, the database server stores the current locations of objects, and updates are issued when objects change their cells. Other works [99, 48, 62] utilize the time-function model and assume that the trajectories of objects are known. Event-driven query processing approaches are introduced for processing continuous kNN queries. The initial query processing phase finds event that might affect the answer of a query, e.g., change of k'th NN and change of NN order; the database monitors motion changes of objects and updates the query answers accordingly.

2.4.2.3 Stationary and Moving Queries

As mentioned, moving-object queries are spatial queries with additional predicates on the time. The spatial component of the query predicate is a spatial point or a spatial region as for the range query or the kNN query. In the context of moving-object databases, the spatial component can be moving over time as well. Depending on the mobility of the spatial component, moving-object queries can be classified into stationary and moving queries as follows.

Stationary Query (on moving objects): The stationary query takes a constant spatial predicate, e.g., a fixed query region (location resp.) as for the range query (the kNN query resp.). To distinguish from traditional static query, stationary query in moving-object terminology means stationary query predicate over moving objects.

Moving Query: By contrast, the spatial predicate of a moving query changes over time. Suppose a query is issued by a moving object itself. The object is referred to as "focal" object of the query. An example of the moving range query is "find out all traffic lights within 1km to me". While the query issuer is traveling in his car, the predicate of this query is a moving circular region with 1km radius. This example shows a moving query over static spatial data points, i.e., locations of the traffic lights.

Apparently, a moving query is either a window query or a continuous query so that it can capture the location changes of the focal object. While most of the existing works on continuous queries investigate this type of moving queries, works, such as MobiEyes [35], SEA-CNN [120], CPM [72], study another type of moving queries, where both the query and the objects are moving continuously, e.g., "show me the locations of all my friends if they are no further than 1km to me".

In this thesis, we focus on optimizing moving-object databases for snapshot, range and kNN queries on future locations of objects.

2.5 Summary

In this chapter, we reviewed some important techniques in moving-object databases. We first introduced the most fundamental problem of modeling objects in the databases. Next, we examined the existing updating protocols for tracking objects and pursuing a trade-off between tracking precision and the number of updates. Then we presented a brief survey of the indexing techniques of moving-objects and reviewed several representative indexes in detail. Finally, we showed some of the existing query processing techniques in moving-object databases.

Chapter 3

A Benchmark for Evaluating Moving Object Indexes

Progress in science and engineering relies on the ability to measure, reliably and in detail, pertinent properties of artifacts under design. Progress in the area of database-index design thus relies on empirical studies based on prototype implementations of indexes. In this chapter, we propose a benchmark for measuring the performance of moving-object indexes. The benchmark evaluates notable aspects of moving-object indexes that have not previously been covered by any benchmark. We also demonstrate the viability of the benchmark by applying the benchmark to some representative moving-object indexes. The results of our benchmark study offer new insight into state-of-the-art moving-object indexes.

3.1 Introduction

Given the large population of moving objects and the mobility of such objects, indexing techniques are essential for supporting frequent updates and efficient query processing in moving-object databases. As reviewed in the previous chapter, numerous moving-object indexes have been proposed in the literature. However, such indexes often come with results of empirical studies suggesting that they are capable of beating its counterparts in the competition. This renders it difficult to obtain an overview of the advantages and disadvantages of the existing indexes. It is even harder for the potential users of the indexes to make a decision on which index is the best suited for a specific application. Therefore, there is a need for a benchmark that can offer important insight into the behavior of each indexing technique and provide guidance on index selection and improvement.

While a few benchmarks [73, 52] exist that address the problem of providing a standard way of evaluating moving-object indexes, they are quite limited with respect to the generation of datasets and workloads. Furthermore, none of the existing benchmarks consider a multi-user environment, which is the typical scenario in almost every moving-object application.

In this chapter, we aim to establish a more comprehensive benchmark than has been seen hitherto. In particular, we propose a benchmark that covers a series of carefully generated datasets, a broad variety of workloads, and a standard evaluation procedure. Different datasets are employed either to measure the overall index efficiency or to simulate certain real-world scenarios. The workloads generated mix updates and queries according to the settings of several parameters. The evaluation procedure exhaustively assesses the performance of a moving-object index regarding the update efficiency, query efficiency, concurrency control as well as storage requirements. We also report on the results of applying the benchmark to six recent indexes spanning all the categories aforementioned in Section 2.3. To the best of our knowledge, no previous studies have compared this many indexes under the same standard. The results of the benchmark study elicit the characteristics of each index and give directions on future index development.

The remainder of this chapter is organized as follows. First, Section 3.2 provides some backgrounds in benchmarks for moving-object indexes. Section 3.3 then presents the details of the proposed benchmark. Section 3.4 reviews our implementations of the indexes. Section 3.5 reports the results of the benchmark study on state-of-the-art indexes of moving objects. Finally, Section 3.6 presents a summary of this chapter.

3.2 Background

Before introducing our benchmark, we first provide some background information on database benchmarks, especially those for moving-object databases.

Several benchmarks have been proposed for traditional spatio-temporal databases that store static objects such as buildings and roads. In early work, Werstein [110] proposed a set of queries to test the temporal and three-dimensional capabilities as well as the spatial capabilities of a database. More recently, Tzouramanis et al. [107] proposed a benchmark for evaluating the performance of access methods for time-evolving regional data. They compared four types of quadtree-based spatio-temporal indexes using raster data. However, both benchmarks do not apply to moving-object databases; they target different types of data, update-operation loads, and types of queries than do by moving-object applications.

For moving-object databases, Theodoridis [103] proposed a benchmark that includes a database description and ten non-predictive SQL-based queries without any experimental results. Recently, Düntgen et al. [29] proposed a benchmark, BerlinMOD, that uses the Secondo DBMS [38] for generating moving-object data. A scenario is simulated where objects move within the road network of Berlin, sampled positions from such movements are used as data. A total of 17 carefully selected, SQL-based queries make up the workload. This benchmark concerns the past, historical positions of moving objects, and it targets the evaluation of complete spatio-temporal DBMSs.

Several benchmarks exist that specifically target techniques for the indexing of the current and near-future positions of moving objects. Myllymaki and Kaufman [73] proposed such a benchmark, called DynaMark. Query and update performance are measured in CPU time, as indexes are assumed to be main-memory resident. Queries on near-future positions of the moving objects are considered. More recently, Jensen et al. [52] proposed a benchmark, called COST. Their workload generation differs substantially from the benchmark we propose in this chapter. Notably, they assume that objects move in Euclidean space or in a complete spatial network, and that an update occurs when an object's actual location differs from that known by the index by a chosen threshold. Object locations are inaccurate, but are known with guaranteed accuracies. Our benchmark uses different actual road networks for dataset generation, and it assumes accurate positions. Finally, Tao et al. [101] conducted a careful study of the query performance of general primal and dual indexes, but with little focus on the update performance. None of the existing benchmarks take into account concurrency control issue, which is crucial to moving-object databases.

3.3 The Benchmark

The goal of the benchmark is to extensively evaluate important aspects of a moving-object index. To do so, we carefully design the datasets and workloads to be used, and propose a standard evaluation procedure. This section presents the details of these aspects.

3.3.1 Datasets and Workloads Generation

Our benchmark considers object locations in the space domain of $100,000 \times 100,000\text{m}^2$, which is sufficient to simulate objects moving in various real environments such as small and large cities. A dataset contains a sequence of updating records, each of which contains the time of the update t, the location and velocity of the corresponding object at t. The benchmark adopts the linear motion model as introduced in Section 2.1, meaning that any future location of an object is derived from its latest updated location and velocity by linear interpolation. Specifically, we generate three types of datasets: uniformly distributed, Gaussian distributed and road-network-based datasets.

Uniformly Distributed Datasets: The initial locations of objects are uniformly distributed in the physical space, and their speeds and directions are randomly chosen. Specifically, an object's speed is randomly selected from $N_{sp} + 1$ candidate speeds ranging from 0 to the maximum speed v_{max}, i.e., $\{0, v_{max}/N_{sp}, 2 \cdot v_{max}/N_{sp}, \ldots, v_{max}\}$. An object issues updates according to an update frequency parameter f_{up}, which indicates how many updates are issued for an object within the maximum update interval t_{mu}. The default value of f_{up} is 1, meaning that each object sends exactly one update every t_{mu} time units. These uniform datasets are used for investigating the overall performance of a moving-object index and the effect of various factors on the index efficiency.

Gaussian Distributed Datasets: The Gaussian distributed datasets aim to capture scenarios where objects (e.g., vehicles) cluster around certain locations of interest such as prominent landmarks and shopping centers. A large number of objects flock to these places, and hence their locations follow a Gaussian-like distribution. In addition, objects trend to slow down when approaching these places, i.e., when it becomes more crowded.

To generate such datasets, we first randomly select a set of (static) points as locations of interest, referred to as hotspots. Around each hotspot, we define multiple speed zones as rings. Each speed zone has a speed limit proportional to its distance to the corresponding hotspot, so that inner zones have lower speeds than outer ones. A speed zone also defines the update frequency accordingly, i.e., an object updates more frequently if it moves with higher speed.

Given the hotspots, an object is placed initially at a position $\overrightarrow{p}$ around a randomly selected hotspot; the distance from the object to the hotspot follows a Gaussian distribution. To ensure that objects follow the same Gaussian distribution as time elapses, updates are generated as follows. First, we find the speed zone that the object is located in and obtain the speed range $[\overrightarrow{v_{\vdash}}, \overrightarrow{v_{\dashv}}]$ and the update frequency f_{up} accordingly. We then generate a new position $\overrightarrow{p_{up}}$ around the same hotspot according to the Gaussian distribution and check whether the distance between $\overrightarrow{p_{up}}$ and $\overrightarrow{p}$ is within range $[\overrightarrow{v_{\vdash}} \cdot t_{mu}/f_{up}, \overrightarrow{v_{\dashv}} \cdot t_{mu}/f_{up}]$, where t_{mu}/f_{up} is the estimated time between two updates. If $\overrightarrow{p_{up}}$ is not in the above range, we re-generate $\overrightarrow{p_{up}}$ until it satisfies the constraint. Finally, we compute the speed of the object as $\overrightarrow{v} = (\overrightarrow{p_{up}} - \overrightarrow{p}) \cdot f_{up}/t_{mu}$. The update is of the form $\langle \overrightarrow{p}, \overrightarrow{v}, t \rangle$, where t is the current time. The next update of this object will be issued when the object reaches $\overrightarrow{p_{up}}$ or enters a new speed zone. Subsequent updates are generated in the same way.

Road-Network-Based Datasets: This type of datasets is generated based on a digital representation of real road networks. We use the network-based moving-object generator of Brinkhoff [12] with some modifications to accommodate the needs of our benchmark.

The digital road network data used derive from the TIGER/Line files [4]. Specifically, a road is represented by a polyline, i.e., a sequence of connected line segments. An object is initially placed on a randomly selected road segment and then moves along this segment in a random direction. The speed is generated in the same way as for uniformly distributed datasets. When the object reaches the end of the road segment, an update is issued, and the object continues moving along another randomly selected connected segment. Each object is required to issue at least one update within t_{mu}.

Query Workloads: The query workloads consist of predictive queries with query times that range from the current time to t_{mu} time units into the future. Two fundamental types of queries are considered. One is the range query that retrieves all objects whose locations fall within certain rectangular region at the query time. The other is the kNN query that retrieves k objects for which no other objects are closer to the query object at the query time. For both types of queries, we first randomly choose a point in the space based on the same distribution of the objects. For a range query, the query range is a rectangle of specific size centered at this point; for a kNN query, the point chosen is used as the query object directly.

3.3.2 Performance Evaluation Procedure

Our benchmark evaluates both time and space efficiency of an index in single-user and multiple-user environments. Four metrics are used: (i) number of I/O accesses, (ii) CPU time, (iii) size of the index on disk, and (iv) throughput (response time). The last metric is only used in multiple-user environments.

For each dataset, an index is initialized during the first t_{mu} time units (e.g., $120s$) and then runs for another $2 \cdot t_{mu}$ time units. Considering that t_{mu} restricts the maximum time interval between two updates of an object, all objects must have been inserted into the index after the initialization phase, i.e., the first t_{mu} time units. Updates are issued according to the update frequency parameter f_{up}. A batch of 100 queries are issued every $t_{mu}/10$ time units. We record the average index size and average query and update costs during the subsequent $2 \cdot t_{mu}$ time units. For each index, we investigate the following aspects.

A1 Data size: The number of objects varies from 100,000 to 1,000,000 (100K and 1M for short).

A2 Time effect: Each index runs for another $5 \cdot t_{mu}$ after the initialization round. The update and query costs are collected and averaged for every $t_{mu}/10$ time units.

A3 Maximum object speed: The maximum speed varies from 10m/tu to 100m/tu ("tu" is short for time unit, which is set to 1 second in the benchmark).

A4 Update frequency: The update frequency f_{up} varies from 1 to 10, meaning that each object updates for 1 to 10 times during t_{mu} time units.

A5 Range query size: All range queries in the query workload are square-shaped with sizes varying from 1,000$\times$1,000m^2 to 10,000$\times$10,000m^2.

A6 Number of neighbors: The number of neighbors k of the kNN queries ranges from 10 to 100.

A7 Query predictive time: The predictive time of the queries varies from 0 to t_{mu}.

A8 Buffer size: An LRU buffer is used by the benchmark to simulate the memory of a database system. The size of the LRU buffer varies from 0 to 1024KB, i.e., from 0 to 256 pages when the page size is 4KB. By default, the buffer is set to 50$\times$4KB pages.

A9 Disk page size: The page size varies from 1KB to 8KB, covering the general cases of most existing operating systems.

A10 Number of hotspots: The number of hotspots varies from 1 to 10,000. When Gaussian distributed datasets are used, we aim to observe the effects of the number of hotspots, which determines the overall distribution of the dataset. The datasets are expected to be skewed when there are few hotspots and near-uniform when there are large numbers of hotspots.

A11 Road network size: For the road-network-based datasets, one additional experiment is introduced to investigate the effect of the network size on the indexes. The size of a road network is defined as the sum of its nodes and edges. Three real digital road networks are included in the benchmark: the Oldenburg (OL) city map has 6,105 nodes and 7,035 edges; the Singapore (SG) city map has 11,414 nodes and 15,641 edges; and the San Francisco (SA) city map has 175,343 nodes and 223,308 edges.

A12 Update/query ratio: In a multi-user environment, updates and queries are intermixed according to a proportion, which is different from the

Table 3.1: Parameters, their value ranges and default values (in bold)

Parameter	**Values**
Space domain	$\mathbf{100,000 \times 100,000m^2}$
Data size	$\mathbf{100K}, \ldots, 1M$
Maximum speed	$10m/tu, \ldots, \mathbf{100m/tu}$
Maximum update interval, t_{mu}	$\mathbf{120tu}$
Update frequency, f_{up}	$\mathbf{1}, \ldots, 10$
Range query size	$\mathbf{1,000 \times 1,000m^2}, \ldots, 10,000 \times 10,000m^2$
Number of neighbors, k	$\mathbf{10}, \ldots, 100$
Query predictive time	$0tu, 10tu, \ldots, \mathbf{60tu}, \ldots, 120tu$
Time duration	$\mathbf{240tu}, 600tu$
Buffer size (number of pages)	$\mathbf{50}, 0, 16, 32, \ldots, 256$
Disk page size (KB)	$1, 2, \mathbf{4}, 8$
Number of hotspots	$1, 10, 100, 1000, 10000$
Road network	$Oldenburg, Singapore, SanFrancisco$
Update/query ratio	$1:100, \ldots, \mathbf{100:1}, \ldots, 10,000:1$
Number of threads	$1, 2, \mathbf{4}, \ldots, 256$

single-user environment where queries are issued every $t_{mu}/10$ time units. In our benchmark, the update/query ratio varies from 1:100 to 10,000:1. This wide range of ratios covers many real scenarios, from query-intensive ones to update-intensive ones.

A13 Number of threads: In a multi-user environment, many users, i.e., objects, update and query the database simultaneously. In our benchmark, we use a multi-threaded program to simulate this scenario. Specifically, we maintain a thread pool that contains a certain number of worker threads. All tasks, mixed with updates and queries, are submitted to a job pool. Then, the tasks are dispatched to free worker threads in chronological order. The throughput of the thread pool and average task response time are recorded as the performance figures. The number of threads varies from 1 to 256.

In summary, Table 3.1 lists all parameters and their value ranges used in the evaluation; the values in bold denote the default values used.

3.4 Index Implementation

As a precursor to applying the benchmark to the six indexes reviewed in Section 2.3, namely the TPR-tree, the TPR*-tree [100], the RUM-tree [119], the B^x-tree [50], the B^{dual}-tree [123], and STRIPES [78], we cover pertinent details of the implementations of the indexes.

TPR-tree: In essence, the TPR-tree is a R-tree with customized querying and updating algorithms. Specifically, each leaf entry contains an object of the form $\langle oid, p_x, p_y, v_x, v_y \rangle$. oid is the identity of the object; p_x, p_y are the coordinates of the object at the reference time of the index (i.e., the time when the index was born); v_x, v_y are its velocities along the x- and y-axes. Each entry in an intermediate node consists of a child pointer pt and a bounding rectangle in the form of $\langle p_x^{\vdash}, p_x^{\dashv}, p_y^{\vdash}, p_y^{\dashv}, v_x^{\vdash}, v_x^{\dashv}, v_y^{\vdash}, v_y^{\dashv} \rangle$, where superscripts $^{\vdash}$ and $^{\dashv}$ indicate the lower and upper bound in corresponding dimension. The updating algorithm of the TPR-tree optimizes the tree structure for predictive queries in the next H time units. In our benchmark, $H = t_{mu}$, meaning that the TPR-tree is optimized for the upcoming t_{mu} time units.

TPR*-tree: The TPR*-tree has exactly the same leaf and internal node structure as the TPR-tree and inherits the above TPR-tree by overloading the functions of `choose_subtree`, `split`, `reinsertion`, and `deletion`. Through overloading, we also avoid any extra performance gap between the TPR-tree and the TPR*-tree introduced by differences implementations. Since the TPR*-tree requires a sample range query for the `split` and `reinsertion` operations, a query with default size (i.e., $1000 \times 1000\text{m}^2$) is used in the benchmark. The implementations of the TPR-tree and the TPR*-tree are based on those provided by the authors of the TPR*-tree [3].

RUM*-tree: It has been proved that all indexes belonging to the R-tree family show inefficiency on dealing with updates, compared with other basic indexing structures such as the B$^+$-tree. The RUM-tree [119] significantly improves the update performance of the R-tree by introducing an update memo. Updates are postponed and executed in bulk. Therefore, we apply its memo-based update approach to the TPR*-tree, and denote the resulting index as the RUM*-tree. Obsolete entries in a leaf node are cleared whenever the node is accessed or the node gets the token for garbage collection. There are 10 tokens in total, and after every 1000 updates, each token is passed to another leaf node.

B^x-tree: The B^x-tree has two partitions. The Hilbert-curve is utilized since it is proved to outperform the other choice (i.e., the Z-curve)[50]. The order of the Hilbert curve used for space partitioning is always optimized with respect to the number of objects in the dataset. The maximum update interval in the B^x-tree is the same as the optimizing time interval t_{mu} of the TPR-tree. The iterative expanding query algorithm in [53] is employed to avoid excessively large query region after expansion. The velocity histogram contains 1,000×1,000 cells.

B^{dual}-tree: The B^{dual}-tree uses the same B$^+$-tree as the B^x-tree and also has two partitions. The order of the Hilbert-curve is also optimized as that of the B^x-tree, but one degree smaller since the B^{dual}-tree partitions two more dimensions (the velocity dimension). The MORs are updated along with the internal nodes of the B$^+$-tree. In particular, the MORs of an internal entry are updated if the key range of its subtree is changed by the update. The MORs are kept in main memory while the index is active.

STRIPES: STRIPES also has two partitions similar to the other two dual-tree indexes. To improve the space utilization, the two optimization techniques provided by the authors in [78] are employed: each leaf node first acquires a half page and will acquire a full page when the data exceeds half of a page; non-leaf nodes are packed together to be stored in as few disk pages as possible.

For the throughput test in multi-thread environment, the B-link concurrency control mechanism as presented in [61] is implemented in the underlying B$^+$-treefor the B^x-tree and B^{dual}-tree. Similarly, the R-tree (for the TPR-tree, TPR*-tree and RUM-tree) employs the R-link technique [75]. As for the STRIPES, we implement a native 2-phase locking technique of the quad-tree. Specifically, a search operation holds a read lock on each node on its current searching path while an insertion/deletion holds a write lock on the current node. The write lock is released when locks on its children are granted and split/merge will not happen to the current node after the insertion/deletion.

3.5 Experimental Study

We now proceed to report on the results of the benchmark study on the six representative indexes. All the indexes were implemented in C++, in the way as introduced in the previous section. To be fair, they adopted the same type of block file, buffer manager, and lock manager. All experiments were conducted on a PC with Intel Core 2 Duo 2.66 GHz processor, 2 GB RAM, and a 200 GB SATA disk, running the Window XP Pro OS.

Table 3.2 shows some statistics collected in the experiments, including the node capacity, index height, leaf node fan-out and the page utilization factor of each index respectively. In addition, for the 100K uniform and gaussian datasets, the sizes of the in-memory memo of the RUM-tree are

∼191KB and ∼146KB respectively, which are 7% and 4% of the size of the corresponding index.

3.5.1 Uniformly Distributed Datasets

First, we examine the first nine aspects of index performance on uniformly distributed datasets. Unless specified, query performance concerns range queries only.

A1 Effect of data size: Figure 3.1 shows the disk space, average update and query performance of the indexes when the dataset size varies from 100K to 1M. First, we summarize our observations on the space utilization of the indexes as follows.

- STRIPES requires the most space due to its unbalanced quad-tree structure. Although STRIPES employs several space optimization techniques, its space utilization is still quite low in comparison with the other balanced index structures.

- Except for STRIPES, all the indexes are similar in size. The size of the RUM*-tree is the second largest one because it keeps a number of obsolete entries. The TPR-tree and the TPR*-tree require nearly the

Table 3.2: Statistics on the indexes

Index	**Capacity**		**Uniform Data (100K)**			**Gaussian Data (100K)**		
	leaf	non-leaf	height	fan-out	page utilization	height	fan-out	page utilization
B^x-tree	170	510	2	115	0.68	2	120	0.71
B^{dual}-tree	170	510	2	115	0.68	2	113	0.67
STRIPES	204	16	3	60	0.29	5	59	0.29
TPR-tree	204	113	2	136	0.66	2	132	0.65
TPR*-tree	204	113	2	139	0.68	2	134	0.66
RUM-tree	170	113	2	108	0.64	2	108	0.63

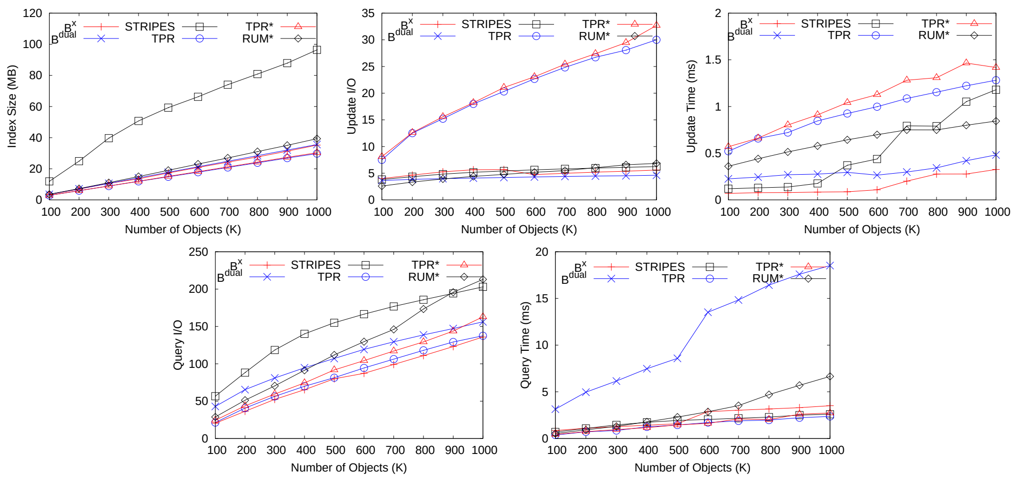


Figure 3.1: A1 Effect of data size

same disk space since they share the same node structure. So do the B^x-tree and the B^{dual}-tree.

- The sizes of the R-tree-based indexes are slightly smaller than those of the B^+-tree-based ones. This is because the B^x-tree and the B^{dual}-tree need to store one more field for each object: the *key* value, i.e., the Hilbert value, resulting in a smaller leaf fan-out.

On the other hand, the update and query costs of all indexes increase linearly with the increasing dataset size, since more objects have to be retrieved for answering a given query on a larger dataset. We also observe several differences in the update and query performance among the indexes, which are detailed as follows.

- The TPR-tree and the TPR*-tree have the best query performance, but also exhibit the worst update performance. The poor update performance is mainly due to the overlaps being followed in deletions among bounding rectangles, which result in multiple search paths during query processing.

- Compared with the TPR*-tree, the RUM*-tree improves the update performance significantly, although at the expense of higher query cost. By simplifying an update to an insertion and a delayed deletion, the RUM*-tree reduces the overall update cost. However, obsolete entries left in the tree degrade the query performance.

- The B^x-tree and the B^{dual}-tree both achieve the best update I/O performance. They benefit from their common base structure, the B^+-tree, in which only a single path needs to be searched during a deletion (and an insertion). Updates of the B^{dual}-tree are generally slower than those of the B^x-tree due to the time-consuming computation of MORs. While the B^{dual}-tree and the B^x-tree have similar query I/O performance, the query time cost of the latter is much higher for the same reason. The query performance will be investigated in detail when we study aspect A5.

- The query processing time of STRIPES is as low as those of the R-tree-based indexes. STRIPES also incurs low update I/O cost that is comparable to that of the B^+-tree-based indexes. However, in comparison with the other indexes, the updating time of STRIPES increases much faster, because the quad-tree becomes taller and taller with increasing numbers of objects. An update then has to follow a long path to reach a leaf node. Similarly, the query I/O of STRIPES is the highest among all indexes due to the low space utilization of the underlying quad-tree.

A2 Effect of time: Figure 3.2 shows the performance of the indexes as a function of time. Regarding the space utilization, STRIPES is significantly affected by the elapse of time. The size of STRIPES shows a periodical

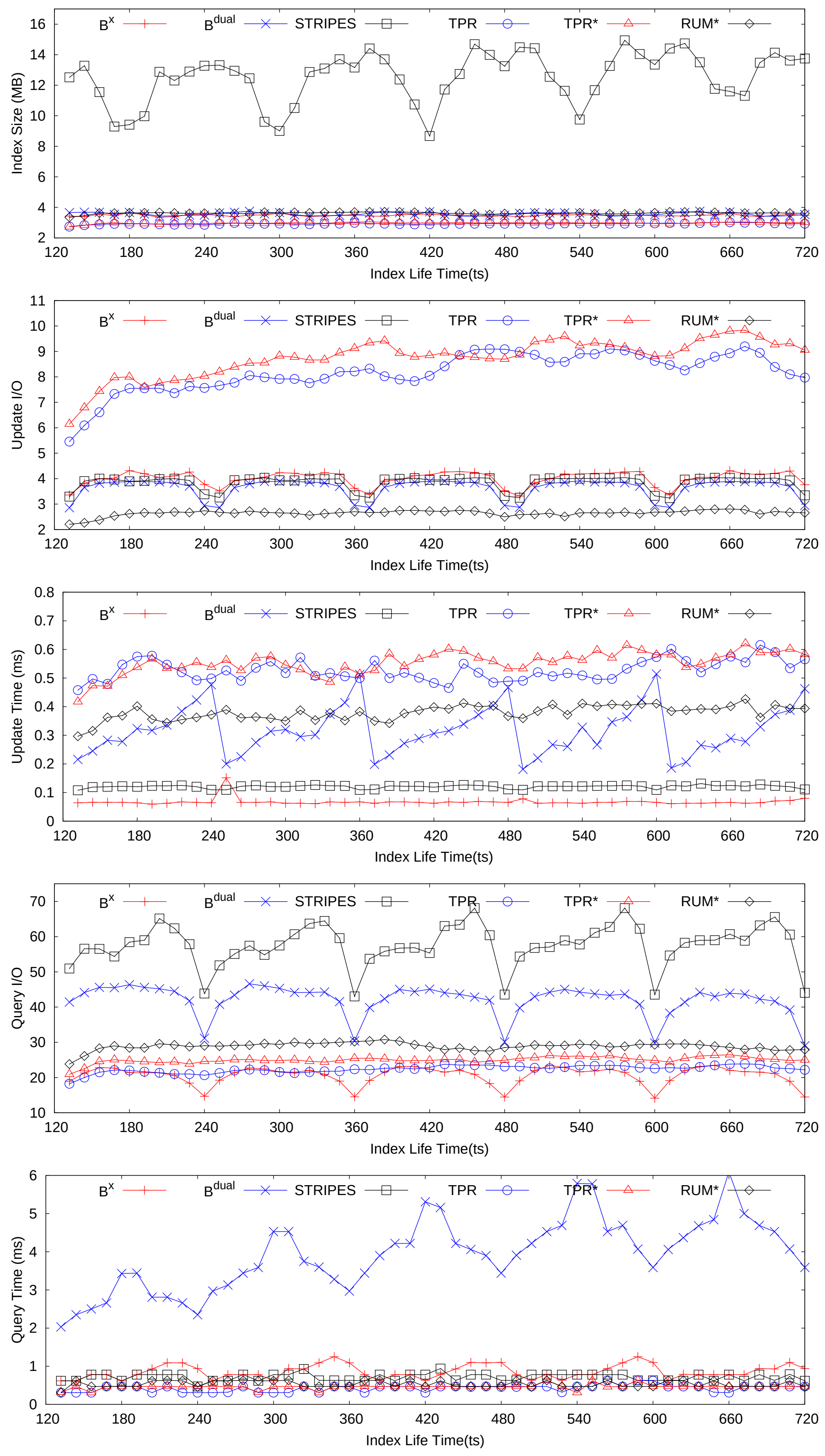


Figure 3.2: A2 Effect of time

change due to the dual-tree structure. Its space utilization is best when the two partitions are balanced. Actually, the B^x-tree and the B^{dual}-tree, both of which are also dual-tree structures, exhibit similar changes in size, but these are not as pronounced as that of STRIPES. In contrast, the sizes of the TPR-tree and the TPR*-tree are barely affected by the passing time. In what follows, we focus on the update and query performance of the indexes.

- The update costs (I/O and CPU time) of the TPR-tree and the TPR*-tree keep increasing with the life time of the indexes. This may be due to the increased overlaps among the BRs. Although the TPR*-tree aims to improve the TPR-tree's performance by shrinking the BRs more often to bound the objects more tightly, frequent BR adjustments introduce additional update I/O. However, as we can see, the overall gain in query performance is not obvious. The difference between the two indexes is not that significant.

- Updates in the RUM*-tree incur the least I/O cost among all the indexes. The reason is that the RUM*-tree eliminates the costly deletion from the update. However, the delayed deletions deteriorate the query performance.

- All dual-tree structures, i.e., the B^x-tree, the B^{dual}-tree, and STRIPES, demonstrate similar periodic patterns for both update and query performance. The performance is best when all objects are in one partition of the index while the other partition is empty. Regarding the update time, updates in both the B^x-tree and STRIPES are too fast to make the periodic changes visible.

A3 Effect of maximum object speed: Figure 3.3 shows the results of varying the maximum object speed from 10m/tu to 100m/tu. Increasing

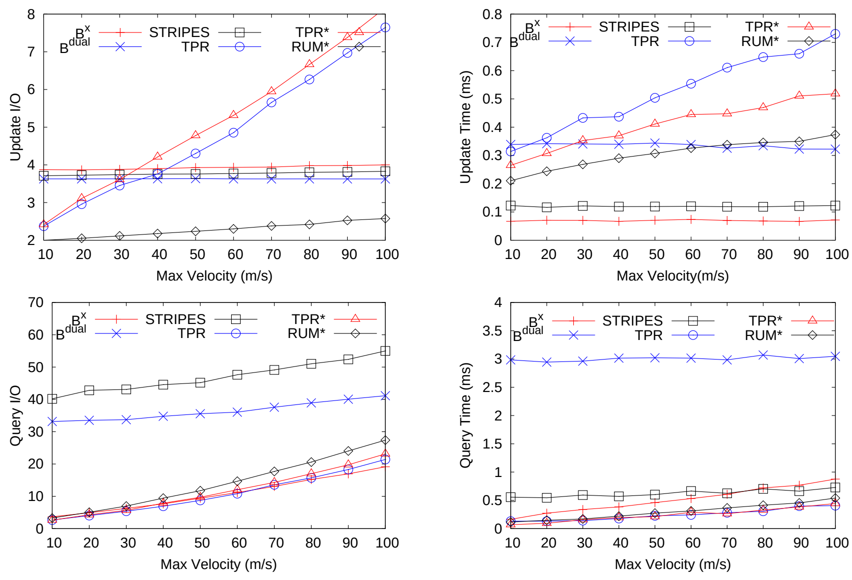


Figure 3.3: A3 Effect of maximum object speed

object speed leads to increasing query costs for all indexes but does not affect the update performance of the B^x-tree, the B^{dual}-tree, and STRIPES. The update costs of the R-tree based indexes increase linearly with the maximum object speed. This is because higher speeds lead to faster expansions of bounding rectangles. Consequently, more overlaps occur and more (partial) paths need to be accessed during updates. We also observe that the RUM*-tree incurs the least update I/O and that the growth rate is also smaller than that of the other two due to the reasons mentioned in A1. In comparison to the others, STRIPES and the B^{dual}-tree are less affected by the objects' speed, because both of them take advantage of object velocity while indexing.

A4 Effect of update frequency: In Figure 3.4, we show the results of varing the update frequency f_{up} from 1 to 10, meaning that the number of updates issued by an object during t_{mu} time units increases from 1 to 10. When t_{mu} is 120tu, for a 100K dataset, the number of updates during each time unit varies from $\frac{100K}{120}$ to $\frac{100K}{12}$, in increment of $\frac{100K}{120}$. In the figure,

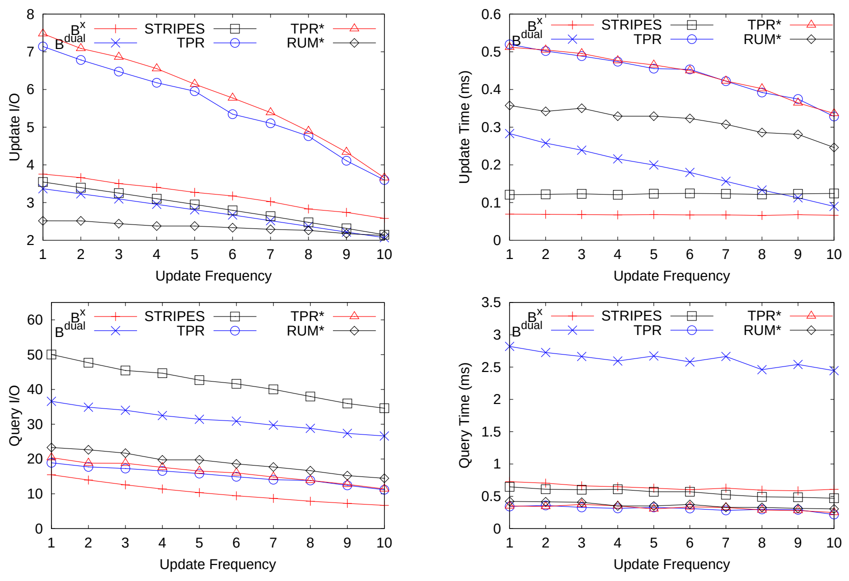


Figure 3.4: A4 Effect of update frequency

we can observe that all indexes, to varying degrees, exhibit better overall performance when updates are more frequent. The possible reasons are the following.

- When updates are frequent, the BRs in the TPR-tree, TPR*-tree, and RUM*-tree are tightened frequently, and hence they bound objects more closely, which leads to better update and query performance.

- The B^x-tree, the B^{dual}-tree, and STRIPES all exhibit minor improvements in update I/O with more frequent updates. These improvements are mainly caused by the presence of the LRU buffer. When an object issues updates frequently, changes in its positions are likely small, and the updated record will be stored in the same leaf node with relatively high probability. If the leaf node is already cached in the buffer, the I/O cost is reduced.

- Updates in the B^x-tree and STRIPES are too fast to make the changes in the updating time perceptible. The B^{dual}-tree benefits more from

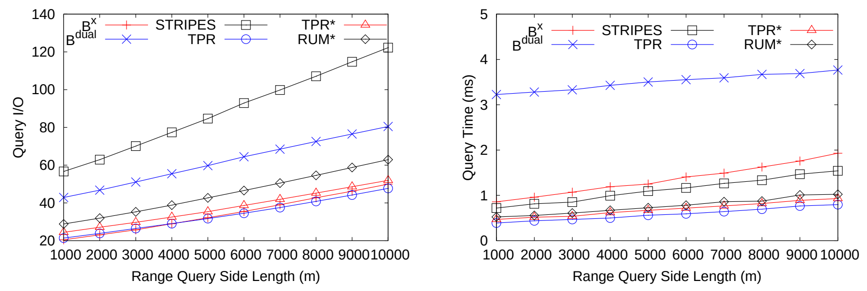


Figure 3.5: A5 Effect of range query size

a high update frequency. Each update on the internal node of the B$^+$-tree requires re-computation of the corresponding MORs. When objects are updated more frequently, the amortized update I/O decreases, and the average time of a single update decreases more significant.

A5 Effect of range query size: In this experiment, we investigate the effect of query sizes by varying the query window from 1,000×1,000m^2 to 10,000×10,000m^2. The results in Figure 3.5 show that the query costs of all indexes increase linearly with a larger query window. A larger window covers more objects and therefore leads to more node accesses. In the following, we elaborate the performance differences among the indexes.

- The TPR-tree and the TPR*-tree perform similarly and incur the lowest I/O cost.
- The query cost of the RUM*-tree is a little higher than that of the TPR*-tree mainly because of the existence of obsolete entries.
- The B^x-tree has similar query I/O to the TPR-tree and the TPR*-tree. This is because the B^x-tree uses the iterative query enlargement algorithm, which reduces the number of false hits as well as the query I/O. However, the query time of the B^x-tree is slightly higher compared to

other indexes, with the exception of the B^{dual}-tree. This is possibly caused by the complex computation of converting a 2-dimensional query range to a 1-dimensional interval and iterative query enlargements.

- Surprisingly, the query I/O of the B^{dual}-tree is much higher than that of the B^{x}-tree. A possible reason could be that the partitioning in the velocity dimensions results in some nearby objects with different velocities being distributed among different leaf nodes, while in the B^{x}-tree, nearby objects are clustered together. The query processing time of the B^{dual}-tree is also higher, but is affected less than the others. This is not only because it has higher query I/O, but also because checking the intersections between queries and large number of MORs is time-consuming.

- Among all the indexes, STRIPES has the highest I/O cost. The main reason is that STRIPES always needs to access more nodes to find the same number of objects due to its low space utilization. In addition, the low space utilization weakens buffering effect. However, thanks to its simple structure and query processing algorithm, query processing of STRIPES is even faster than that of the B^{x}-tree.

A6 Effect of number of neighbors: Considering kNN queries, all indexes exhibit slight increases in I/O costs and query processing time with larger k. In Figure 3.6, all indexes show similar trends as they do in Figure 3.5 with range queries. In the case of the B^{x}-tree, a kNN query is processed using incremental range queries. Due to the inaccuracy in estimating the initial search radius, the query range may be extended several times in order to find the k neighbors. Such repeated range queries result in an increased

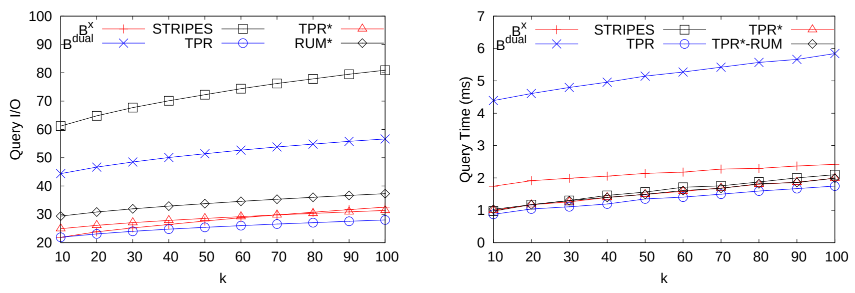


Figure 3.6: A6 Effect of number of neighbors

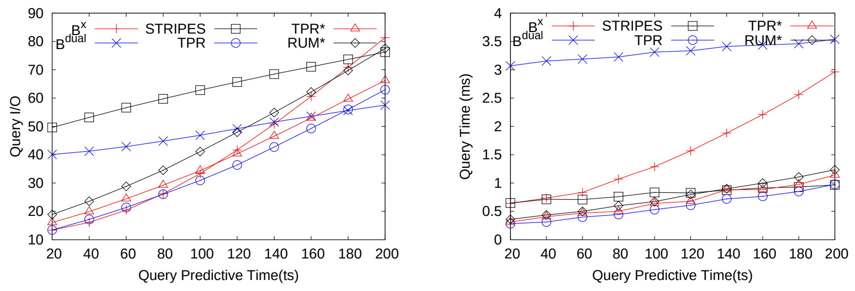


Figure 3.7: A7 Effect of query predictive time

query processing time while the I/O cost remains unchanged because of the bufferring effect.

A7 Effect of query predictive time: Figure 3.7 shows the effect of the query predictive time interval, varied from 0 to 120tu. It is not surprising that the query costs of all the indexes increase with longer the query predictive time, since either BRs or query ranges need to be enlarged to larger sizes. Among all indexes, the effect on the B^x-tree is most the significant. This is because only the B^x-tree processes a predictive query through query enlargement. A longer predictive time leads to a larger query region and hence a larger number of false positives, which introduces more I/O and longer processing time. The B^{dual}-tree and STRIPES, which index objects in the dual space, are less affected by the query predictive time, since the MOR and quadrant enlargements, supervised by the velocity constraints, are relatively small.

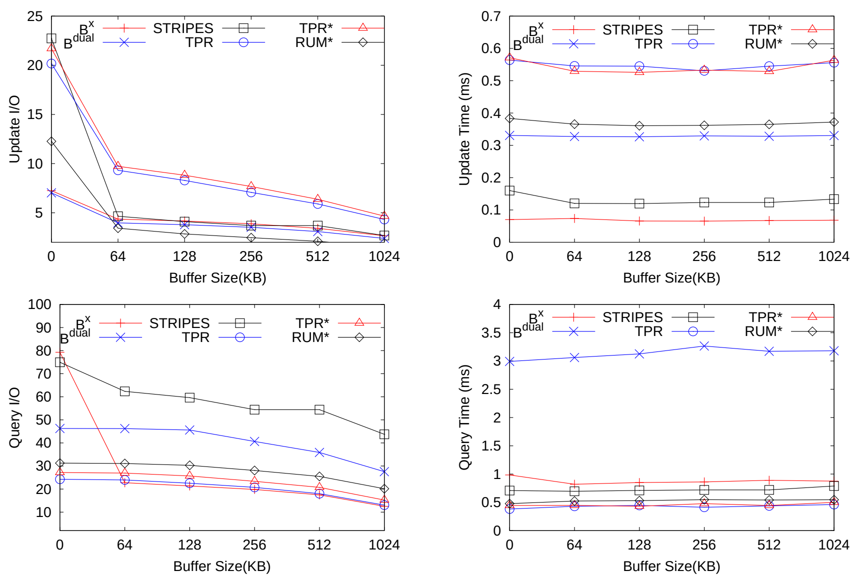


Figure 3.8: A8 Effect of buffer size

A8 Effect of buffer size: To investigate the effect of the buffer, we vary the number of buffer pages from 0 to 256. As shown in Figure 3.8, the query time is relatively stable since it is mainly determined by the query algorithms used by the indexes; the I/O costs of some indexes can be reduced substantially by using a small buffer (e.g., 32 pages); after certain point, the benefit of increasing the buffer size is negligible. In what follows, we discuss why some indexes benefit more from the use of the buffer.

- The three R-tree based indexes save quite a number of update I/O by using the buffer. This is because they need to read and write several internal nodes for each update—the costs of accessing such nodes are saved if the nodes are already in the buffer. Moreover, an update in the TPR*-tree is always slower than that in the TPR-tree, which conforms to the fact that an update in the TPR*-tree is more complex than the TPR-tree. We also observe that the buffering has less of an effect on the query performance.

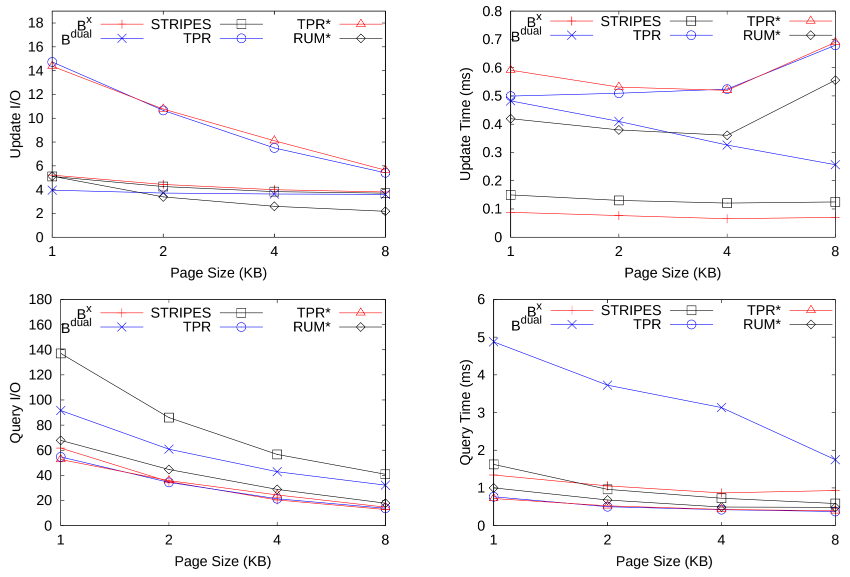


Figure 3.9: A9 Effect of disk page size

- The B^x-tree saves much query I/O with buffering, since some of the nodes are searched repeatedly during a query. Its update performance is almost independent of the buffer size because an insertion or a deletion only needs to access a single path in the index.

A9 Effect of disk page size: Figure 3.9 shows the effect of the disk page size which varies from 1KB to 8KB. With larger pages, fewer number of disk I/O is required for the same operation while more time is needed to check the larger number of objects in a node.

- For all indexes, both the update I/O and the query processing time decrease with larger pages. The reason is straightforward. With larger pages, the number of I/O required for a query decreases and so does the processing time.

- In terms of the update time, the TPR-tree and the TPR*-tree both benefit from a small page size, i.e., 1KB or 2KB. A larger page contains more objects that need to be accessed for an update. There is a higher probability of having to update the BRs all the way back to the root during an update. Therefore, it takes more time. However, with the help of an LRU buffer, most internal nodes can be kept in main memory, and the update I/O does not increases.

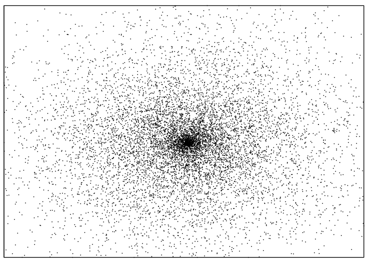
- It seems that the RUM*-tree performs best when the disk page size is 4KB. This may be due to the settings of the RUM*-tree in the experiments. As 10 tokens are used for garbage collection, the cleaning frequency may be too low for the 1KB and 2KB pages where there are many leaf nodes; and it may occur too often for the 8KB pages (few leaf nodes). Keeping too many obsolete entries will decrease the efficiency of the index, while performing the cleaning too often will slow down the updates. Tuning the parameters of the RUM*-tree's garbage collection may help to relieve the deterioration.

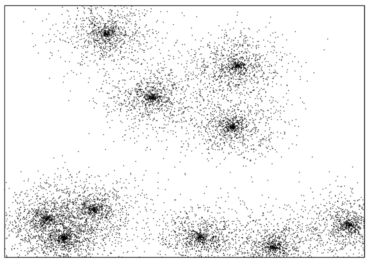
3.5.2 Gaussian Distributed and Road-Network-Based Datasets

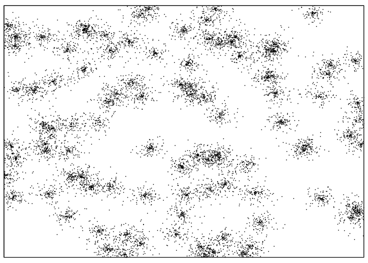
A10 Effect of number of hotspots: We now investigate index performance while varying the number of hotspots from 1 to 10,000. Figure 3.10 visualizes some of the datasets used in this set of experiments, and Figure 3.11 shows the results on these datasets. Note that in the dataset with 10,000 hotspots, objects are nearly uniformly distributed.

The performance of all the indexes demonstrate similar trends as those for the uniform datasets.

- The update costs of the TPR-tree and the TPR*-tree decrease with the increasing numbers of hotspots resulting in datasets that become more uniform. In comparison to the TPR*-tree, the RUM*-tree has a steadier update performance.


(a) 1 hotspots


(b) 10 hotspots


(c) 100 hotspots

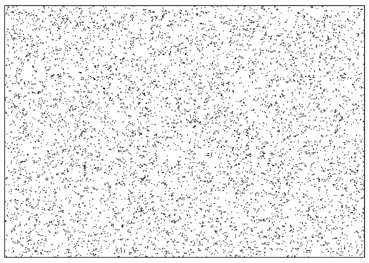

(d) 10000 hotspots

Figure 3.10: Gaussian datasets

- The B^x-tree incurs higher update I/O when the data are more skewed. As mentioned in Section 2.3, the granularity of space partitioning for the B^x-tree is optimized for the number of objects in the dataset. When data become more skewed, the object density around the hotspots becomes higher. Many objects are indexed in the same cell with the same indexing key, leading to overflow pages that adversely affect the performance.

- In comparison with the B^x-tree, the update costs of the B^{dual}-tree and STRIPES are affected less by the skewed data distribution. This may be because both of them consider the object velocity in partitioning, which weakens the impact of space distribution on the updates.

- In terms of queries, all indexes require higher query I/O when there are less hotspots, i.e., the data becomes less uniformed. Since queries

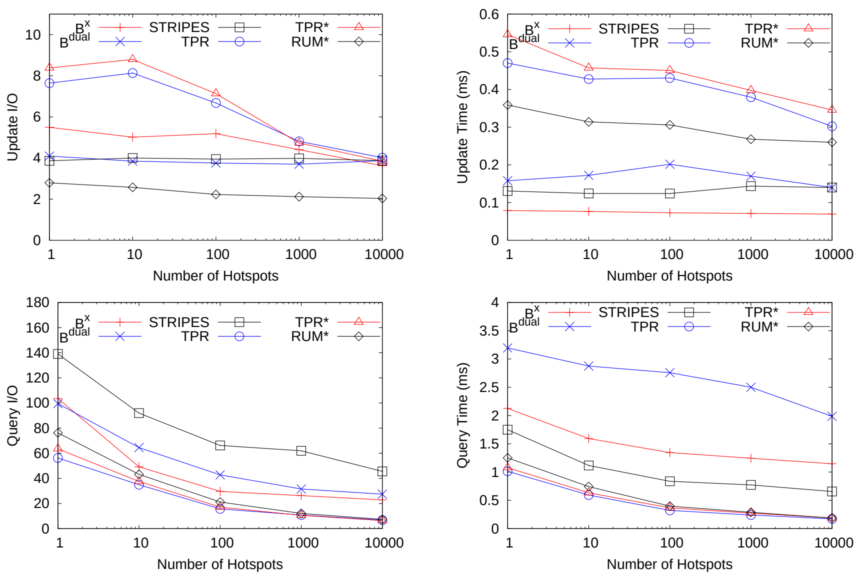
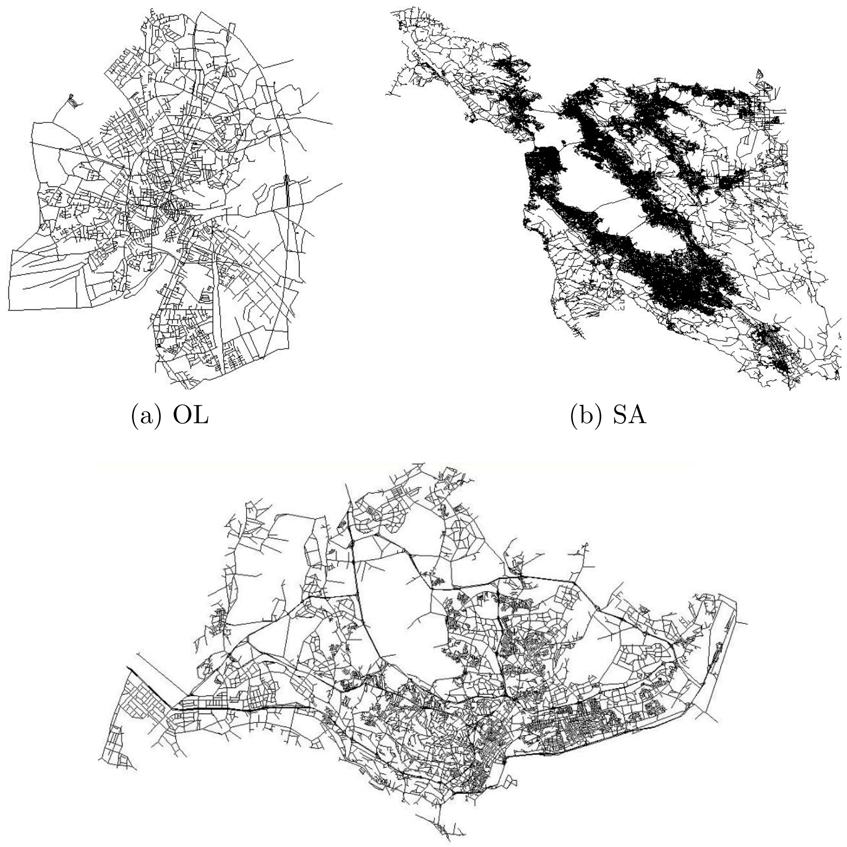


Figure 3.11: A10 Effect of number of hotspots

are generated according to the same Gaussian distribution around hotspots, a query needs to retrieve more objects in a more skewed dataset where objects are crowded around few hotspots.

- The query I/O of the B^x-tree increases fastest among all indexes when data becomes more skewed. This is due to the fixed space partitioning of the B^x-tree. The B^{dual}-tree and STRIPES, both of which partition the dual space (i.e., location and velocity), exhibit slower increases in query I/O in comparison to that of the B^x-tree. As for the R-tree-based indexes, the query I/O increases at the similar, slow rate. Since the R-tree-based indexes dynamically adjust BRs based on data distribution, the data skew has the relatively small impact on them.

A11 Effect of network size: This experiment examines the effect of network size by using three real maps of different sizes (OL < SG < SA), as shown in Figure 3.12. The results are shown in Figure 3.13, where we can observe that the performance is not affected by the size of the network. To


(a) OL (b) SA

(c) SG

Figure 3.12: Road network

obtain a better understanding of this behavior, we first analyze the features of each dataset.

- The OL network is the smallest. The nodes are distributed more evenly in space and hence the average length of each edge is the longest. This in turn results in less updates for each object.

- The SA network contains the largest number of nodes and edges, and hence it has relatively short edges. Also, the distribution of the nodes is more skewed than for the OL and SG networks where we can see that a large number of objects concentrate in the central part of the network. Therefore, objects in the SA dataset have the highest update

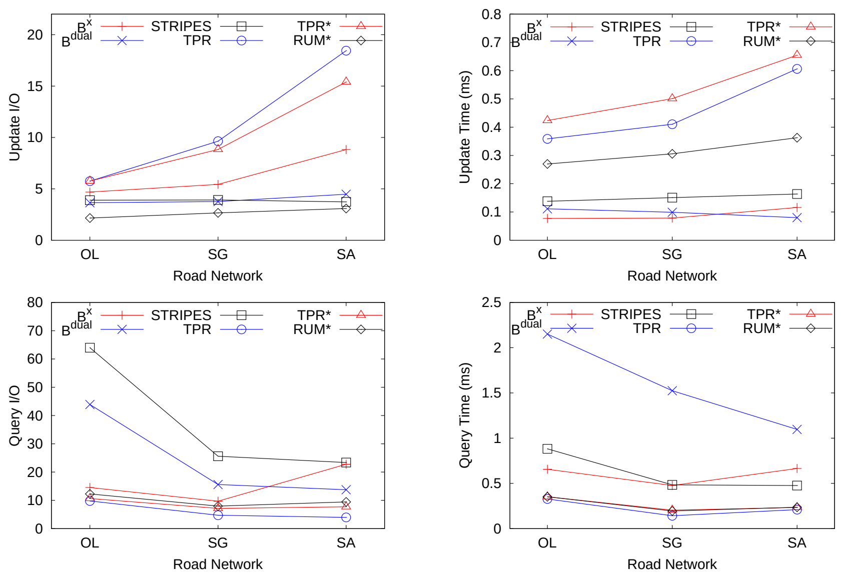


Figure 3.13: A11 Effect of road network

frequency and are much skewed.

- The SG dataset has an in-between size and hence a moderate update frequency. The object distribution is close to uniform in most parts of the network.

Figure 3.13 suggests a complex effect of the road network on the index performance. The size of the network affects the update frequency while the shape of the network determines the distribution of the objects. The index performance is affected by both factors, each of which was discussed in A4 and A10.

- Considering updates, the TPR-tree and the TPR*-tree both perform the best with the OL dataset and the worst with the SA dataset. This is because of the joint effect of the update frequency and the data skew. The update cost of all the other indexes remains nearly the same for the three networks, since the object distribution barely affects the their update costs (Figure 3.11 in A10) and since the impact

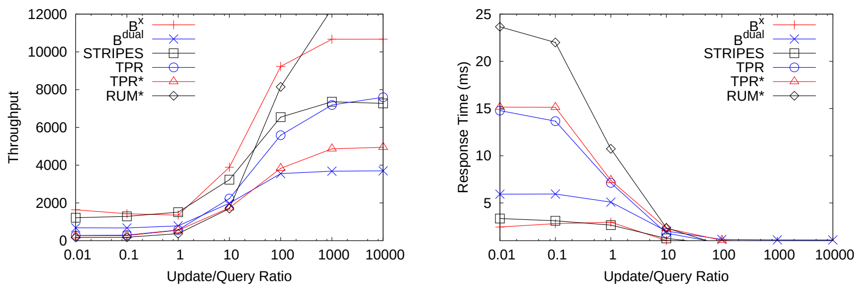


Figure 3.14: A12 Effect of update/query ratio

of the update frequency on their update costs is also small, i.e., less than one I/O, as shown in Figure 3.4.

- In terms of the query cost, all indexes perform the best in the SG dataset and worst in the OL dataset, which is consistent with that the SG datasets has a balance between the update frequency and the object distribution.

3.5.3 Concurrency Control

Finally, we evaluate all indexes in a multi-user environment. The performance is measured by two metrics: throughput and response time. Throughput is defined as the average number of tasks finished in one unit time (1tu); response time is the average time between the submission and completion of a task, i.e., either an update or an query.

A12 Effect of update/query ratio: Figure 3.14 shows the effect of update/query ratio. We can observe that the throughput of all indexes increase when there are more updates and fewer queries. The reason is that an update usually can be executed more quickly than a query. Hence, during the same duration of time, more tasks can be finished if a larger portion of them are updates. The response time decreases naturally with increasing throughput. Next, we focus our discussion on the performance differences between indexes.

- The B^x-tree exhibits the highest throughput. This is because it performs very fast for updates and relatively fast for queries. More importantly, the concurrency control mechanism of the B^+-tree is more efficient than those of the R-tree and the quad-tree.
- STRIPES achieves the second highest throughput since it has comparable update and query performances as the B^x-tree. Moreover, the unbalanced quad-tree in STRIPES has more levels and hence concurrent operations are more likely to be performed at different levels of the tree.
- The TPR-tree and TPR*-tree yield lower throughput mainly due to their relatively low update performance.
- The B^{dual}-tree has the lowest throughput, although the concurrency control in the B^+-tree is efficient. This is because the B^{dual}-tree holds the nodes for a longer time to handle each update (especially for MOR computations).
- The RUM*-tree has a relatively low throughput when queries constitute the majority of the workload. However, as the proportion of updates in the workload increases, the throughput of the RUM*-tree increases at the highest rate, just because updates in the RUM*-tree are quite fast.

A13 Effect of number of threads: Figure 3.15 shows the performance of indexes with various degree of concurrency when the update/query ratio is fixed at 100:1. Observe that the throughput of all the indexes peak at

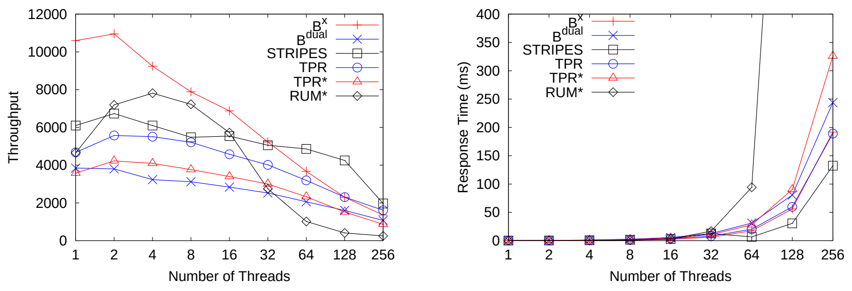


Figure 3.15: A13 Effect of number of threads

about 1 or 2 threads and then decrease when there are more threads. The main reason is that an update exclusively locks the node being accessed. When the update frequency is high, multiple threads will compete for the right to access the same node, meaning that it takes more time for each thread to execute an operation successfully. In addition, extra time is spent on frequent thread switching. As a result, the increase in the number of threads will not help with the throughput, but may even degrade the overall performance. This is also the reason why the response time increases nearly exponentially with the number of threads.

- The B^x-tree achieves the highest throughput and shortest response time among the indexes for the same reason as explained in A12.

- STRIPES has a moderate throughput, which is higher than that of the TPR-tree, the TPR*-tree, and the B^{dual}-tree. The impact of increasing number of threads on STRIPES is less pronounced compared to the others. Because of the quad-tree structure, STRIPES is unbalanced. Locks are distributed among different levels of the tree.

- Apparently, the TPR-tree outperforms the TPR*-tree in concurrent operations. The TPR*-tree accesses more internal nodes than the TPR-tree for each insertion and deletion. Although such additional I/O cost is relieved by the buffering in a single thread environment, the problem is hard to solve in a multi-threaded environment, where the TPR*-tree locks the internal nodes more often than the TPR-tree does.

- The performance of the RUM*-tree surpasses all the other indexes except the B^x-tree when there is no more than 32 threads. In addition to the internal nodes of the tree, the threads have to compete for the update memo as well. Therefore, the throughput of the RUM*-tree degrades fast with more worker threads.

3.5.4 Result Summary

Table 3.3 presents an overall comparison of the six indexes with respect to a total of eight different performance aspects, as listed in the leftmost column. For each row in the table, an entry with value "1" indicates that the corresponding index usually performs the best regarding the specific aspect, while an interval "*a*-*b*" in other entries indicates that the corresponding index is a to b times more expensive than the best index. For example, the first row summarizes the update I/O performance across all experiments. We can see that the B^x-tree and B^{dual}-tree both have value "1", which indicates that they generally outperform the other indexes. As for the TPR-tree, an interval "0.7-7" means the update cost of the TPR-tree is 70% of that of the B^x-tree in the best case and is about 7 times higher than that of the B^x-tree in the worst case. Generally speaking, in each entry, a smaller number and a smaller range of the interval indicates a better and more stable performance.

To summarize, we have the following interesting findings from Table 3.3.

- With some restriction on the storage, if an application needs to deal

Table 3.3: Performance summary

	RUM*-tree	TPR-tree	TPR*-tree	B^x-tree	B^{dual}-tree	STRIPES
Update I/O	0.9-2.3	0.7-7	0.7-7.7	1	1	1.1-1.3
Update time	6-16	8-17	8-18	1	1.2-8	0.9-1.2
Range query I/O	2-5	1	1	0.8-1.8	2-9	2.5-10
Range query time	1.1-2.4	1	1	1.1-1.6	1.2-3	0.9-1.5
kNN query I/O	2.5-2.6	1	1	1.1-1.3	2.3-2.5	3.3-3.5
kNN query time	2-2.2	1	1	1.5-2	3.7-5.1	1.4-1.5
Storage space	1.1-1.2	1	1	1.1-1.2	1.1-1.2	3.2-4
Response time	2-30	3.5-6.5	5-10	0.4-1.6	1.5-2.2	1

with a large number of updates, but relatively few queries, the B^x-tree is a good choice since it performs the best among all indexes in terms of both update I/O and update time and it requires relatively little storage space. This is also true in the multi-user environments where we can see that the B^x-tree exhibits almost the least response time.

- If an application has to handle more queries than updates, the TPR-tree is the proper one since it beats all the others with a stably efficient query performance regardless the query types.
- If the composition of the workload is unclear and the storage space is not a concern, the B^x-tree and STRIPES are good choices since they have relatively good, if not optimal, update and query performance in most situations.

3.6 Summary

In this chapter, we propose a benchmark for studying important properties of indexing techniques and evaluating the performance of moving-object indexes. The benchmark includes a dataset and workload generator, and a standard evaluating procedure. The parameters that control the datasets and workloads are varied considerably with the goal of covering a wide range of settings, the idea being to obtain a better understanding of the strengths and the weaknesses of an index under varying circumstances. We also report on the application of the benchmark to six recent moving-object indexes, thus offering findings that can serve as guidelines for choosing a proper index for a specific application. x

Chapter 4

ST^{2}B-tree: a Self-Tunable Spatio-Temporal B$^+$-tree Index

Due to the mobility of objects, the workload of a moving-object database, including updates and queries, changes all the time. Traditional static indexes are not able to cope well with such changes, i.e., their effectiveness and efficiency are seriously affected. This calls for the development of novel indexes that can be reconfigured automatically based on recent state of the system. In this chapter, we present the ST^{2}B-tree, a self-tunable spatio-temporal B$^+$-tree index for moving-object databases. The design of the ST^{2}B-tree allows it to change its configurations online, and hence makes it adaptive to the data variability and workload changes. To bring the tuning of the ST^{2}B-tree into use, an online tuning framework is also proposed in this chapter.

4.1 Introduction

Database tuning is crucial to the efficient operation of a database management system. In fact, most commercial database management systems provide some tuning tools. The goal of tuning is to ensure the database system always operates in an "optimal" state. Variations in the workload, including both queries and updates, can significantly impact the performance of the database. Usually, some components of the database, such as indexes and the query optimizer, can be adjusted to adapt to these workload changes.

Compared with conventional databases, data managed by moving-object databases are much more dynamic. In moving-object applications, such as the traffic management system, vehicles move continuously, and their locations change frequently. More importantly, the distribution of objects varies over time and space. For example, in a traffic management system, some places are likely to be more crowded and populated than others. The number of vehicles at certain locations may be larger during the day and relatively smaller at night. This means that the numbers of updates and queries vary in different places and at different times. Even for the same query, the number of objects involved is quite different at different times, resulting in different amount of work on the system. Given these dynamics in moving-object applications, database tuning is even more crucial than in conventional databases.

Traditionally, tuning is the responsibility of a database administrator, who needs to monitor the state of the database and makes the tuning whenever it is necessary. However, it is impractical for the administrator to keep an eye on the system all the time. In practice, the administrator tunes the database regularly by checking the recent state of the system in the system log. A better solution is to make a database system self-tunable so that tuning proceeds automatically with minimal human intervention.

In moving-object databases, for a tuning mechanism to be useful, it must be not only automated, but also light-weight and fully online. A moving-object database should be operational at all times (24 by 7). Updates and queries arrive at the system continuously, making it impractical and uneconomical to hold the system and postpone all regular operations (i.e., updates and queries) until the tuning procedure completes. The tuning should be performed online and in real time.

While some works have been done to develop self-tuning technologies in database systems, these are largely restricted to traditional static databases. A representative example of this line of work can be found in [17]. However, no existing work has exclusively studied the tuning problem on moving-object databases. Although there are many studies on moving-object indexes, they mostly focused either on designing indexing structures or developing efficient algorithms for various kinds of queries. Variability in the workload has so far been overlooked in the design of moving-object indexes. In this chapter, we study this problem, focusing on index tuning in moving-object databases, and make the following contributions.

- We identify the necessity of index-tuning in moving-object databases by specifying three types of data diversity in moving-object applications and their impacts on the performance of moving-object indexes.

- We present the ST^{2}B-tree—an *S*elf-*T*unable *S*patio-*T*emporal B^+-tree index for moving objects. The ST^{2}B-tree employs the "multi-tree" technique, which facilitates online tuning by maintaining multiple sub-trees. The index dynamically adapts the granularity of space partitioning based on the object density within regions around a set of reference points.

- We discuss the potential problem of the "multi-tree" technique and propose an eager update mechanism to cut down the overhead of migrating objects in the ST^{2}B-tree during rollover from one sub-tree to another.

- We introduce a framework for online tuning of moving-object indexes. Although specially designed for the ST^{2}B-tree, the framework is generally applicable to all "multi-tree" indexes. We also give guidelines on the selection of the best values for all tunable parameters for the ST^{2}B-tree, based on rigorous theoretical analysis on the workload with respect to each of these tunable parameters.

- An extensive experimental study was conducted to evaluate the performance of the ST^{2}B-tree and the tuning process. The results show that the tuning process lessens the degradation in the effectiveness of the index with virtually no overhead.

The remainder of the chapter is organized as follows. The next section discusses some background information. Section 4.3 presents the ST^{2}B-tree and provides insights on the tuning capability of the ST^{2}B-tree. Section 4.4 studies the constraints of "multi-tree" technique and introduces the eager update mechanism to alleviate it. Sections 4.5–4.6 provide the theoretical analysis on the tuning parameters. Section 4.7 presents the online tuning framework and describes how it works. Section 4.8 reports the results of an exhaustive performance study. Finally, Section 4.9 summarizes this chapter.

4.2 Background

In this section, we provide some background information on the design of the ST^{2}B-tree. We first describe three types of data diversity in moving-object applications. Then, we discuss how these data diversity may hinder an index from being "optimal".

4.2.1 Types of Diversity in Moving Object Applications

Intrinsically, moving objects are spatial objects whose locations change with time. The diversity (i.e., variability) of data fall into the following three categories, as illustrated in Figure 4.1.

Spatial Diversity: In a real moving-object environment, objects are rarely uniformly distributed in space, i.e., the density of objects varies in different areas in space. For example, some places of interest, such as commercial centers and major road junctions, always have higher object density than other places. Such a situation is portrayed in Figure 4.1(a). At time t_1, two regions (enclosed by solid circles) are highly populated with objects, while the other regions, e.g., the two enclosed by dashed circles, are of relatively lower object density.

Temporal Diversity: Besides the non-uniformity of object locations at any instance of time, non-uniformity also exists along the dimension of time, meaning that the total number of active objects tracked by the system changes with time. For example, it is common to have more vehicles during the daytime, causing heavy traffic on the roads. In contrast, traffic is much lighter at night, with fewer vehicles roaming on the roads. Referring to our running example, the distribution of objects at t_2 (Fig-

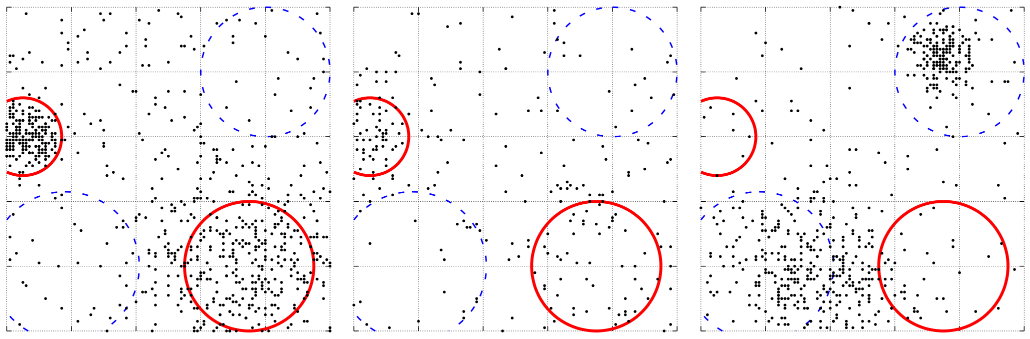

(a) Case 1: at t$_1$ (b) Case 2: at t$_2$ (c) Case 3: at t$_3$

Figure 4.1: Examples of data diversity in moving-object applications

ure 4.1(b)) is still the same as at t_1 (Figure 4.1(a)); however, the average density of the entire space decreases significantly.

Spatio-temporal Diversity: It is not uncommon to have a combination of the above types of diversity, i.e., both the number and the distribution of active objects change with time. Continuing with the example in Figure 4.1, from t_2 to t_3, besides an increase in the total number of objects, the places of interest move as well. Sparse areas may become dense while dense areas may become sparse due to some external factors such as peak hours, road work and accidents. As a practical scenario in real-life, between 8am to 9am, people drive from the residential suburbs to downtown; during office hours, most vehicles move in and around the downtown area; after 5pm, people start trickling home. The residential suburbs and downtown behave as places of interest alternately for different periods of time of a day.

4.2.2 Impact of Data Diversity on Index Performance

Given the aforementioned three types of diversity in moving-object applications, we now discuss their effects on the index performance. As indicated in

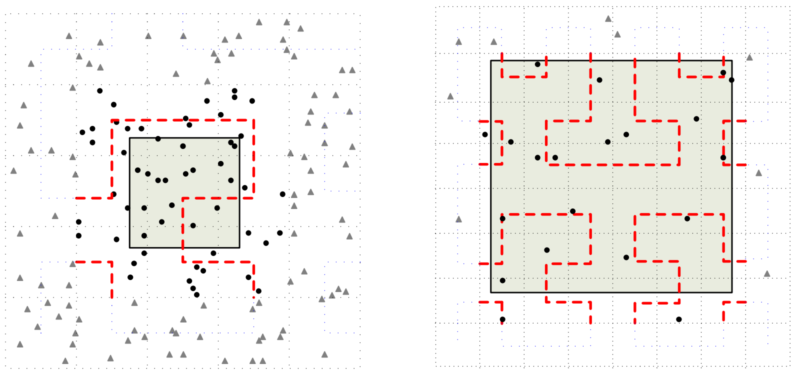

(a) Coarse partitioning (b) Fine partitioning

Figure 4.2: Effect of data diversity and space partitioning

Section 2.3, data partitioning indexes such as the TPR-tree [87] can dynamically cluster objects according to their distribution. Each cluster always contains a specific number of objects. Therefore, these indexes are less affected by the data diversity. On the contrary, space-partitioning indexes such as the B^x-tree [50] adopt a fixed way of partitioning the space. While the size and position of a partition are fixed, the number of objects remain in each partition varies in different areas and at different times. As a result, the data diversity of moving-object applications has a significant effect on the performance of this category of indexes. We hereby explain this effect in more detail.

Coarse Space Partitioning: When the space partitioning is too coarse with respect to the data density, each partition will contain many objects. A range query processing procedure has to check all objects in the partitions that overlap with the query region. This may lead to a large number of false positives for those partitions that only partially overlap with the query region (i.e., cover the boundary of the query region). For example, in Figure 4.2(a), all objects in the 3×3 cells (solid dots) that intersect with the query region (the dark square) must be examined, and 8 of these are boundary partitions where most of objects are negative answers to the query. On the other hand, recall that objects in one partition are treated indistinguishably by the index, e.g., indexed with the same key in the B^x-tree. Having too many objects in one partition leads to an increase in the number of overflow pages in the index. The overflow pages are organized as a chain, meaning that the update cost will be higher as the overflow pages have to be read and searched linearly. In addition, the existence of overflow pages compromises the balance property of indexes, which bounds the search cost.

Fine Space Partitioning: At the opposite end, if the space is partitioned in finer granularity, each partition contains fewer objects. For example, Figure 4.2(b) zooms in on the query region in Figure 4.2(a). With a finer partitioning, most partitions contain no more than one object. Obviously, no additional update overhead is incurred. The number of false positives in query processing also decreases because the smaller boundary partitions contain fewer objects. However, for query processing, instead of 6 partitions in Figure 4.2(a), 36 partitions have to be checked in Figure 4.2(b). Although the overhead of pruning false positives is reduced, the increase in the number of partitions to search deteriorates query performance. Take the B^x-tree as an example. A range query is transformed into several one-dimensional range queries that can be evaluated on the underlying B$^+$-tree. The number of one-dimensional range queries needed increases from 2 to 9 as shown in Figure 4.2 (Each one-dimensional range query is shown as dotted, thick line in Figure 4.2).

In summary, given the object density in some area and at some instance of time, the granularity of space partitioning affect the efficiency of an index significantly. Considering the three types of diversity in moving-object applications, an index suffers from performance degradation when the (fixed) space partitioning becomes unsuited for the data. In order to minimize the performance degradation caused by the data diversity, a moving-object index should: (1) discriminate between regions of different densities, and (2) adapt to density and distribution changes with time.

4.3 ST^{2}B-tree: a Self-Tunable Index for Moving Objects

In this section, we introduce the structure and basic algorithms of the ST^{2}B-tree. We also explain why the ST^{2}B-tree is capable of tuning itself to fit the workload changes in moving-object databases.

4.3.1 ST^{2}B-tree Structure

Similar to the B^x-tree as introduced in Section 2.3, the ST^{2}B-tree is built on the B$^+$-tree without any changes to the underlying tree structure and basic operations, such as insertion and deletion. Since the B$^+$-tree is a one-dimensional ($1d$ for short) index, to enable the adoption of the B$^+$-tree, we need to construct such $1d$ keys for the objects first. Given that a moving object is a spatio-temporal point in its natural space, the $1d$ key must capture both spatial and temporal characteristics of the object. Specifically, in the ST^{2}B-tree, the $1d$ key is composed of two components: KEY_{time} and KEY_{space}, as introduced in the following.

4.3.1.1 Index with Time

To deal with the time dimension, the ST^{2}B-tree adopts the "multi-tree" technique, which is adopted by many indexes such as the B^x-tree, B^{dual}-tree and STRIPES as reviewed in Section 2.3. Specifically, the "multi-tree" technique works as follows. Assume T_{up} is the maximal time interval between contiguous updates of an object, which means that an object is updated at least once in time interval T_{up}. The B$^+$-tree is logically split into two sub-trees, BT_0 and BT_1. Each sub-tree is assigned with a range of T consecutive time points, where $T = T_{up}$. Specifically, the time ranges covering BT_0 and BT_1 are $[2iT, 2iT+T)$ and $[2iT+T, 2iT+2T)$ respectively; as time elapses, the value of i increases from zero and the time ranges of the two sub-trees roll over alternately. The index therefore rolls over with time. This behavior is illustrated in Figure 4.3.

Without loss of generality, suppose that current time is $t \in [2iT, 2iT+T)$. As shown in Figure 4.3, BT_1' is the older sub-tree, i.e., the one covering the earlier time interval $[2iT - T, 2iT)$, while BT_0 is the younger sub-tree, i.e., the one covering the current time interval $[2iT, 2iT + T)$. BT_1' is now in a monotonic shrinking phase — only deletions affect BT_1', and all insertions are conducted in the other sub-tree BT_0. Since $T = T_{up}$, all objects would be updated during $[2iT, 2iT + T)$. As a result, at the next transition time $2iT + T$, all objects are stored in the younger sub-tree BT_0, while the older sub-tree BT_1' becomes empty. Subsequently, a new time interval $[2iT + T, 2iT+2T)$ is assigned to the empty sub-tree BT_1', which is now represented as BT_1 in Figure 4.3. This change of time interval for the sub-tree BT_1 does not interfere with any objects which are currently indexed in BT_0. The older sub-tree BT_1' is refreshed while the original younger sub-tree BT_0 now becomes the "older" one and enters the shrinking phase. In the next

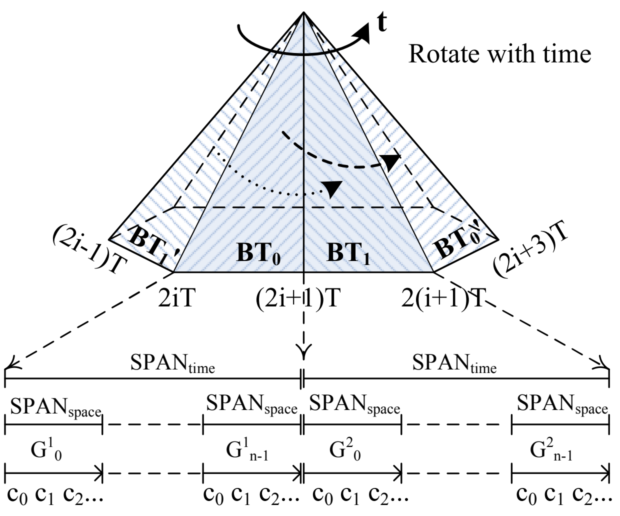


Figure 4.3: The essence of the ST^{2}B-tree

transition time $2iT + 2T$, BT_0 will become empty and be assigned with the newer time interval $[2iT + 2T, 2iT + 3T)$, and so on and so forth.

Suppose an object o issues an update $\langle \overrightarrow{p}, \overrightarrow{v} \rangle$ at t_{up}, where $\overrightarrow{p}$ and $\overrightarrow{v}$ represent the location and velocity of the object at t_{up}. The object will be indexed in the sub-tree whose time range covers t_{up}. For instance, updates issued in $[0, T)$ are indexed in the first sub-tree BT_0 while those in $[T, 2T)$ fall into the right sub-tree BT_1. Subsequent updates in $[2T, 3T)$ go back to BT_0, and so on and so forth.

Each sub-tree has a unique reference time T_{ref} and o is indexed with its location at T_{ref}, $\overrightarrow{p}' = \overrightarrow{p} + \overrightarrow{v} \cdot (T_{ref} - t_{up})$. Herein, we set T_{ref} to the upper boundary of the time range, which is:

$$T_{ref} = (i + 1)T, \text{if } t_{up} \in [iT, iT + T). \tag{4.1}$$

Basically, T_{ref} can be any instance of time. The value set in Equation 4.1 is the best choice that optimizes the performance of the index. We will discuss the selection of T_{ref} later in Section 4.6.1.

Finally, the temporal component KEY_{time}, which is used to identify the sub-tree that the object belongs to, is obtained as follows:

$$KEY_{time} = \begin{cases} 0, & \text{if } t_{up} \in [2iT, 2iT + T), \\ 1, & \text{if } t_{up} \in [2iT + T, 2iT + 2T). \end{cases} \tag{4.2}$$

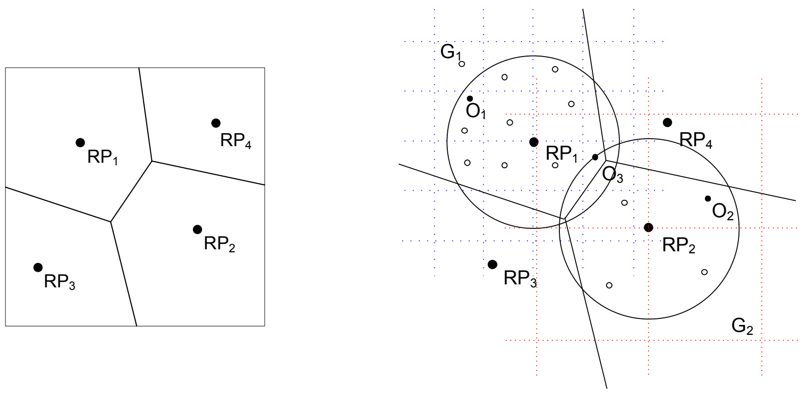
4.3.1.2 Index in Space

According to the taxonomy in Section 2.3, the ST^{2}B-tree is a data-supervised space-partitioning index. The ST^{2}B-tree adopts the same technique used in [124, 49]. The space is first partitioned into Voronoi cells of a set of reference points, and objects inside a Voronoi cell are clustered into a group, which occupies a contiguous segment of the entire key space. In [124, 49], objects inside each Voronoi cell are sorted by their distances to the corresponding reference point. The ST^{2}B-tree, however, further partitions the space of each sub-space with a uniform grid. An object is indexed by the id of the grid cell it belongs to.

As mentioned, in moving-object applications, objects are unevenly distributed in different areas. In the ST^{2}B-tree, these areas are distinguished with the help of reference points. Specifically, given a set of n reference points $\{RP_0, RP_1, \ldots, RP_{n-1}\}$, the data space is then partitioned into n disjoint regions $\{VC_0, VC_1, \ldots, VC_{n-1}\}$ in terms of the distance to the reference points, that is, the partitioning forms a Voronoi Diagram of the n reference points as illustrated in Figure 4.4(a). Along with each reference point RP_i, the ST^{2}B-tree maintains a grid G_i, which is centered at RP_i and covers its Voronoi Cell VC_i.

Given an update $\langle \overrightarrow{p}, \overrightarrow{v} \rangle$ of object o and its nearest reference point RP_i, the spatial component KEY_{space} is:

$$KEY_{space}(o) = i \times SPAN_{space} + cid(\overrightarrow{p}', G_i), \tag{4.3}$$



(a) Space partition (b) Grouping objects

Figure 4.4: Spatial key generation in the ST^{2}B-tree

where i is the grid id, i.e., the id of the reference point closest to o. A portion of $SPAN_{space}$ continuous keys in the B$^+$-tree are reserved for each grid. $i \times SPAN_{space}$ helps to locate the portion of key values reserved for grid G_i. $cid(\overrightarrow{p}', G_i)$ is the id of the cell in G_i that $\overrightarrow{p}'$ belongs to. The cell id is assigned with the help of a space filling curve. To preserve locality well, the ST^{2}B-tree utilizes the Hilbert curve as shown in Figure 4.2. In order to guarantee that the keys of adjacent grids do not intersect with each other, $SPAN_{space}$ must be an upper bound of *cid* in all grids.

Figure 4.4(b) illustrates how objects are indexed around two reference points RP_1 and RP_2. o_1, whose nearest reference point is RP_1, is indexed in G_1 and o_2 in G_2 likewise. Another object o_3, although covered by G_2 as well, is indexed in G_1 because o_3 is closer to RP_1 than RP_2, i.e., in RP_1's Voronoi cell VC_1. Although overlap may exist between adjacent grids, the Voronoi cells of reference points are disjoint. Therefore, it is clear for an object which grid it belongs to.

In summary, in the ST^{2}B-tree, object o is indexed with KEY_{ST^2}:

$$KEY_{ST^2} = KEY_{time} \times SPAN_{time} + KEY_{space}, \tag{4.4}$$

Algorithm 4.1: Compute Key

```
Input  : location $\overrightarrow{p}$, current time $t$
Output: $1d$ key
1 $KEY_{time} = \lfloor t/T \rfloor \mod 2$;
2 $KEY_{space} = KEY_{space}(\overrightarrow{p})$;
3 $key = KEY_{time} * SPAN_{time} + KEY_{space}$;
4 return $key$;
```

Algorithm 4.2: Update

```
Input  : object update $\langle oid, \overrightarrow{p}, \overrightarrow{v} \rangle$, current time $t$
1 Delete $(oid)$;
2 $T_{ref} = \lceil t/T \rceil * T$;
3 $\overrightarrow{p}' = \overrightarrow{p} + (T_{ref} - t) * \overrightarrow{v}$;
4 $key$=ComputeKey $(\overrightarrow{p}')$;
5 Insert $(key, \langle oid, \overrightarrow{p}', \overrightarrow{v} \rangle)$;
```

where $SPAN_{time}$, similar to $SPAN_{space}$, is the size of the key range reserved for each sub-tree. $SPAN_{time}$ must be an upper bound of KEY_{space} to avoid overlap between keys in two sub-trees. KEY_{time} and KEY_{space} are derived as described.

Algorithm 4.1 computes KEY_{ST^2} as in Equation 4.4. Algorithm 4.2 shows the steps of an update, where t is the current time. T_{ref}, KEY_{time} and KEY_{space} are derived based on Equation 4.1, Equation 4.2 and Equation 4.3 respectively. To process an update, the old entry of the object is first deleted from the index (line 1 in Algorithm 4.2). Procedure Delete(oid) deletes an object by its identifier. An auxiliary hash table is used to keep the identifiers of all objects and their current indexing keys. When the old record is deleted, Procedure Insert$(key, record)$ inserts the $record$ of the object into the B$^+$-tree using the new indexing key.

Figure 4.3 shows the essence of the ST^{2}B-tree. The current time is in $[2iT, 2iT+T)$. The two logical sub-trees are BT_0 and BT_1. At time $2iT+T$, the time range of BT_1' has changed to $[2iT+T, 2iT+T)$, shown as BT_1. The next rotation will happen at $2iT + T$. BT_0 will be assigned a newer time range, shown as BT'_0. For key allocation, the key space is halved according to the update time. At the second level, each half is further partitioned for the n grids. Finally, at the bottom level, within the key space of each grid, objects are sorted in ascending order of the id of the cells they belong to.

4.3.2 Snapshot Query Algorithms

Algorithm 4.3 depicts the evaluation procedure of a simple range query in the ST^{2}B-tree. Both sub-trees are searched. Since all objects in sub-tree BT_i are indexed with positions at time T_{refi}, the algorithm first enlarges the query region R_q from query time t_q to T_{refi} using the global maximum velocity max_v (line 3) (the same way as in the B^x-tree). Then, the grids of the reference points whose Voronoi cell intersects with the enlarged query region need to be further searched (lines 4-5). The cells that intersect with the enlarged query region are retrieved (line 6-7), in ascending order of cell id assigned by using the space filling curve. Finally, an object is added to the result set if its position at time t_q is contained in the query region (lines 8-9). When the query q is a current query, $t_q = t_{now}$ (t_{now} is the current time when the query is issued). If q is a predictive query, $t_q = t_{now} + h$, where h denotes the prediction interval.

In the ST^{2}B-tree, a kNN query is conducted as incremental range queries until exact k neighbors are found. We omit the detailed algorithm here (a similar procedure can be found in [50]). The query processing starts with an initial search radius r. If k neighbors are not found in the initial search region, it extends the search radius by *increment*. Both r and *increment*

Algorithm 4.3: Range Query

```
Input  : Query region $R_q$, query time $t_q$
Output: Query results $R$
1  $R = \emptyset$;
2  foreach sub-tree $BT_i$ do
3      $R'_q$ = EnlargeRegion($R_q, max_v, t_q, T_{refi}$);
4      foreach reference point $RP_j$ do
5          if $R'_q$ overlaps with $VC_j$ then
               // $VC_j$ stands for the Voronoi cell of $RP_j$
6              foreach cell $c$ of $G_j$ that overlaps with $R'_q$ do
7                  foreach object $o$ in $c$ do
8                      $p_{t_q}$ = Position($o, t_q, T_{refi}$); if $p_{t_q} \in R_q$ then
9                          Add $o$ into $R$;
10 return $R$;
```

are set to D_k/k as in [50], where

$$D_k = \frac{2}{\sqrt{\pi}}[1 - \sqrt{1 - \sqrt{\frac{k}{N}}}]. \tag{4.5}$$

D_k is the estimated distance to the k'th nearest neighbor [102] and N is the number of objects in a unit space.

4.3.3 Why is the ST^{2}B-tree Tunable?

We now explain why the ST^{2}B-tree can be easily tuned to adapt to the three kinds of data diversity discussed in Section 4.2.1.

Diversity in Space: The ST^{2}B-tree partitions space using n reference points. Each reference point has its own grid and the cell sizes are not necessarily identical for all the grids. In fact, grid granularity can be determined by object density in the Voronoi cell of the reference point. As shown in Figure 4.4(b), objects are relatively dense around RP_1, therefore G_1 is of finer granularity. For RP_2, objects are relatively sparse, so G_2 uses larger cells and partitions space at a coarser level. By using different grids in different areas (for different reference points), the ST^{2}B-tree can discriminate between regions of different density.

Diversity with Time: Based on the constraint that T is the maximum update time interval, at each transition time iT, the sub-tree in ST^{2}B-tree to be refreshed with a new time interval is always empty. At this point, the ST^{2}B-tree can use a different granularity of grids for this empty sub-tree, according to the object density investigated in the previous T time. The grids of the other non-empty sub-tree are kept unchanged and all the objects currently in the index are unaffected. Therefore, the two sub-trees in the ST^{2}B-tree can have their own set of grids. While searching/updating in a sub-tree, the corresponding set of grids is used. No collision happens. Once the granularity are determined, they will not change during the next T time before the next transition time.

Diversity in Space with Time: In a similar way as the granularity of grids can be tuned to capture the change of object density with time, the number and positions of reference points can also be tuned to capture the change of object distribution with time. For example, in Figure 4.1, at t_0 and t_1, the centers of the four circles are used as reference points. Later, at t_2, the centers of the three circles are used as reference points instead. Further, the two sub-trees use their own set of reference points and corresponding grids.

In short, thanks to the "multi-tree" structure of the ST^{2}B-tree, the two sub-trees work independently without interference. Any settings about the indexing, including both reference points and grid granularity, are tunable in the ST^{2}B-tree. As a result, the ST^{2}B-tree can meet the two requirements mentioned in Section 4.2. Later in Section 4.7, we will discuss the details about the self-tuning strategy of the ST^{2}B-tree, which helps the ST^{2}B-tree perform better with time-dependent workload changes.

4.4 Eager Update: Minimizing Object Migration during Rollover

As discussed in the previous section, the “multi-tree” indexing technique opens the door for tuning the index online. In addition, with the “multi-tree” technique, an object is indexed with a reference time T_{ref}, growing in step with the current time. The overall query performance of a “multi-tree” index will not deteriorate as time elapses, in contrast with the single-tree indexes such as the TPR-tree [87].

4.4.1 Effect of T: the length of the time interval covered by a sub-tree

The effectiveness and efficiency of the “multi-tree” structure depend largely on the value of T, i.e., the length of the time interval covered by a sub-tree.

4.4.1.1 T vs. Query Cost

The sub-tree rollover at the transition time iT is feasible only if no existing objects are affected, i.e., when the old sub-tree contains no objects. This condition is guaranteed by constraining the length of the time interval covered by a sub-tree (T) to be at least the maximum time interval between contiguous updates of an object (T_{up}), as mentioned in Section 4.3.1.1. In practice, some objects update very infrequently, resulting in a large T_{up} and a large T as well. As aforementioned, the ST^{2}B-tree avoids false negatives by query enlargement in query processing. The query performance largely depends on the size of the enlarged query region, while the query enlargement is proportional to the value of T. With a larger T, a query is enlarged to cover a larger region, incurring higher query processing costs. In the extreme case when $T \to \infty$, the "multi-tree" index degrades to a single tree index. The query enlargement keeps increasing as time elapses and the query performance deteriorates significantly with a large T.

4.4.1.2 T vs. Object Migration

Since the value of T affects the query performance significantly, we consider setting T as small as possible. However, if T is set smaller than the maximum update interval T_{up}, at the point of transition, there may be some objects remaining in the older sub-tree. For the "multi-tree" technique to work correctly, we have to migrate all these un-updated objects to the younger sub-tree manually before assigning a new time interval to the sub-tree[*]. In particular, objects are inserted into the younger sub-tree with their locations at the corresponding reference time. This location is estimated from the location and velocity stored in the older sub-tree. The migration of un-updated objects incurs extra update workload on the index. Furthermore, since the migration happens at the transition time, it causes a burst of updates. All the upcoming regular updates are postponed until the migration finishes and the older sub-tree is empty. Although, a query can be processed as usual, the response time may be long due to lock contention

[*]An alternative way of dealing with these un-updated objects is to discard them from the database, simply assuming that those objects have left the system (e.g., parking). No migration is required in this case. The older sub-tree is treated as empty. All the active objects are indexed by the other sub-tree and kept unaffected by the tuning. We do not consider this situation here.

caused by those updates. From this point of view, T should be as large as possible so as to minimize the migration cost. The smaller T is, the larger portion of objects need to be migrated, and the more frequent migration happens.

In brief, the value of T affects the query cost and the migration cost in opposite ways. We will discuss the effect of T on both the query and update theoretically in Section 4.6.2.

4.4.2 Eager Update

As shown above, a larger T leads to a higher query cost, while a smaller T may introduce extra migration cost. In order to avoid migration as much as possible while keeping T relatively small, we now introduce an eager update technique.

The principle of eager update is to update as many objects as possible without increasing the number of I/O accesses. Algorithm 4.4 shows the steps of an eager update. Consider an update in the form of $(oid, \overrightarrow{p}, \overrightarrow{v})$, where *oid* is the identity of the object, $\overrightarrow{p}, \overrightarrow{v}$ are the current location and velocity of the object.

The update procedure first finds the record of object *oid*. *oldkey* is the current indexing key of *oid* and L is the leaf node that contains the record (lines 1-2). If the object is already indexed in the younger sub-tree (line 4), the update continues in a normal way as in Algorithm 4.2 (line 5). Otherwise, if the update needs to move the object from the older sub-tree to the younger one, the eager update checks all the other objects in the same leaf node L (line 11), and computes their estimated locations in the reference time of the younger sub-tree (line 12). If the object will be indexed by the same key as the object *oid* (line 14), this record is inserted

Algorithm 4.4: Eager Update

```
Input : object update $\langle oid, \overrightarrow{p}, \overrightarrow{v} \rangle$
1  $L = \mathsf{Search}(oid)$;
2  $oldkey$=key of the existing record of $oid$ in $L$;
3  $KEY_{time} = \lfloor oldkey/SPAN_{time} \rfloor$;
4  if $KEY_{time} = t/T \mod 2$ then
5      Update $(oid, \overrightarrow{p}, \overrightarrow{v})$;
6  else
7      $T_{ref} = \lceil t/T \rceil * T$;
8      $\overrightarrow{p}' = \overrightarrow{p} + \overrightarrow{v} * (T_{ref} - t)$;
9      $key = \mathsf{ComputeKey}(\overrightarrow{p}')$;
10     $Q = \emptyset$;
11     foreach record $r_i = \langle key_i, oid_i, \overrightarrow{p}_i, \overrightarrow{v}_i \rangle$ in the leaf node $L$ do
12         $\overrightarrow{p}' = \overrightarrow{p}_i + \overrightarrow{v}_i * T$;
13         $key' = \mathsf{ComputeKey}(\overrightarrow{p}')$;
14         if $key == key'$ then
15             Q=Q + $\langle key, oid_i, \overrightarrow{p}', \overrightarrow{v}_i \rangle$;
16             Delete $r_i$ from L;
17     $L' = \mathsf{Search}(key)$;
18     foreach $r_i \in Q$ do
19         Insert $r_i$ into L';
```

into a queue Q (line 15) and then deleted from the current leaf node L (line 16). Q maintains objects to be updated in an eager manner. Subsequently, the update procedure finds the leaf node L' in the younger sub-tree which object *oid* should be inserted into (line 17). Finally, all objects in Q are inserted into L', since they have the same indexing key as object *oid* (lines 18-19).

4.4.2.1 Benefits of Eager Update

As we can see, an eager update accesses the same tree nodes as a regular update. Therefore, additional I/Os are incurred only when the eager update causes node splitting or merging while the regular update does not. With eager update, a regular update moves more than one object to the younger tree. As a result, in T time (supposing T is smaller than the maximal update time interval), more objects are updated to the younger tree than usual. Fewer objects are left in the older sub-tree and the migration cost decreases. Another benefit of eager update is that it saves the cost of regular update as well. If an object has been updated into the younger sub-tree eagerly, the real update of the object affects the younger sub-tree only, i.e., both deletion and insertion happen in the younger sub-tree. Considering that objects move continuously in the space, it is likely that the deletion and insertion access the same leaf node. The insertion and deletion of an update work along the same path. The number of I/O accesses could decrease, with an LRU buffer, even if it is small in volume. On the contrary, if eager update is not supported, the deletion operation would delete an entry in the older sub-tree and the insertion operation would insert an entry in the younger sub-tree. In this case, even if the object does not move at all, different set of nodes will be visited during these two steps of the update. We shall see the effect of migration and cost of different updates in Section 4.8.3.

4.4.2.2 Degree of Eagerness

In Algorithm 4.4, all objects, which have the same key as the core updating object in the younger sub-tree, are updated eagerly. Here, we introduce a tuning knob, denoted as $\mathcal{D}_e$, to migrate objects from the older sub-tree to the younger sub-tree more aggressively. Essentially, all objects whose indexing keys in the younger sub-tree differ no more than $\mathcal{D}_e$ from that of the core updating object are migrated. The intuition here is that objects with similar indexing key values are likely to be geographically close together. However, since objects move continuously in space, there is no guarantee on where those objects will be (in the younger sub-tree) after T time. As such, as $\mathcal{D}_e$

increases, the I/O overhead will also increase. Moreover, since objects with key values smaller than $\mathcal{D}_e$ may be found in other nodes, accessing these nodes will further increase the I/O cost. To handle the second problem, we restrict the migration to only those objects that are in the same leaf node as the core updating objects. In this way, *EagerUpdate* shown in Algorithm 4.4 corresponds to the case when $\mathcal{D}_e$=0. On the other extreme, with $\mathcal{D}_e$=∞, all objects within the leaf node will be migrated. A moderate value of $\mathcal{D}_e$ will result in migrating only subset of the objects within the node.

4.5 Grid Granularity

As discussed in Section 4.2.2, data density and grid granularity are important factors that affect the performance of any index based on grid partitioning. Thus, grid granularity is a core parameter to be tuned for the ST^{2}B-tree and any other space partitioning indexes. To find the optimal granularity, we now analyze the effects of different grid granularity on the overall performance of an index.

For ease of analysis, we assume that objects are uniformly distributed in the entire space and the space is partitioned using a single grid. Without ambiguity, the result is directly applicable to each grid in the ST^{2}B-tree with local uniform assumption around each reference point.

The notations used are listed in Table 4.1. We start our analysis by giving a definition of the grid order λ.

Definition 4.1. Grid Order.

The grid order λ is defined as the resolution of the space filling curve used for mapping grid cells into $1d$ values. A grid of order λ partitions the data space

Table 4.1: Notations for analyzing grid granularity

Symbol	**Description**
N	number of objects
λ	resolution of space filling curve
IO_L	number of leaf node I/O
N_L	number of leaf nodes
n_o	average number of overflow nodes per leaf node
C_L	capacity of leaf nodes
C_O	capacity of overflow nodes
f	average fan-out of tree nodes
h	height of the tree
V	velocity used in query enlargement
L	side length of the square-sized region covered by a leaf node
L_q	side length of a square-sized range query
t_q	time of the query
T_{ref}	reference time of objects stored in the index
N_q	number of $1d$ range queries
N_I	number of internal node accesses

into $2^\lambda \times 2^\lambda$ cells.

Suppose that the entire space is a unit space which is partitioned by a grid of order λ. Then the side length of each cell is $2^{-\lambda}$. The following Lemma 4.2 estimates the number of leaf I/O accesses [123].

Lemma 4.2 The number of leaf node accesses of a square-sized range query is

$$IO_L = N_L[L + L_q + V \cdot |t_q - T_{ref}|]^2 \tag{4.6}$$

N_L is the number of leaf nodes. Let $N_L = 2^{2i}$, where i is an integer no larger than λ. Then, each leaf node covers $2^{2(\lambda-i)}$ cells on average, which forms a square with side length $L = 2^{-i} = (1/N_L)^{1/2}$.

Let $L_Q = L_q + V \cdot |t_q - T_{ref}|$. L_Q is the side length of the enlarged query region. $(L + L_Q)^2$ is the probability that the enlarged query range intersects with the spatial region covered by a leaf node. The number of leaf nodes to be accessed by a query is therefore $N_L[L + L_Q]^2$.

However, Equation 4.6 in Lemma 4.2 is valid only when there are no overflow pages in the tree, which means that there are very few objects having the same key. Considering the uniform distribution of objects, $\frac{N}{2^{2\lambda}} \leq 1$ ($\lambda \geq \frac{1}{2}\log_2 N$) is a necessary condition of Lemma 4.2.

Lemma 4.3 If $\lambda \geq \frac{1}{2}\log_2 N$, IO_L does not change when λ increases.

Proof. If $\lambda \geq \frac{1}{2}\log_2 N$, each cell contains at most 1 object and duplicate keys are rare. $N_L{=}\frac{N}{f}$ and $f{=}69\% \cdot C_L$, where 69% is a typical fill factor of the B$^+$-tree.

$$\begin{aligned} IO_L &= N_L(L + L_Q)^2 = N_L\left((1/N_L)^{\frac{1}{2}} + L_Q\right)^2 \\ &= \frac{N}{f}\left((f/N)^{\frac{1}{2}} + L_Q\right)^2. \end{aligned}$$

Thus, IO_L is independent of λ. □

Lemma 4.4 If $\lambda \leq \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$, IO_L increases when λ decreases.

Proof. If $\lambda \leq \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$ ($\frac{N}{2^{2\lambda}} \geq C_L$), the number of objects contained in a cell is larger than the capacity of a leaf node. Each leaf node has only one key; therefore $N_L = 2^{2i}, i = \lambda$. Suppose each leaf node has n_o overflow nodes, then:

$$IO_L = N_L(1 + n_o)\left((1/N_L)^{\frac{1}{2}} + L_Q\right)^2$$

Let $C_L = C_O$,

$$\begin{aligned} n_o &= \frac{\frac{N}{2^{2\lambda}} - C_L}{C_O} = \frac{N}{2^{2\lambda}C_O} - 1 \\ IO_L &= \left(\frac{N}{C_O}\right)(2^{-\lambda} + L_Q)^2. \end{aligned}$$

Clearly, IO_L increases as λ decreases. □

Lemma 4.4 can be explained as follows. All objects contained in the boundary cells, which partially intersect with the query range, need to be checked. As λ decreases, the extent of a grid cell grows exponentially, bringing in more false positives. Access to these false positives incurs additional I/Os.

Corollary 4.5 The grid order λ that minimizes IO_L is in range

$$[\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L, \frac{1}{2}\log_2 N]$$

Proof. This can be easily deduced from Lemma 4.3-Lemma 4.4, IO_L keeps unchanged when λ is larger than $\frac{1}{2}\log_2 N$ (Lemma 4.3), and increases when λ is smaller than $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$ (Lemma 4.4). □

According to Corollary 4.5, in order to minimize the number of leaf node accesses during a query, the space filling curve with resolution $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \leq \lambda \leq \frac{1}{2}\log_2 N$ should be used.

While Corollary 4.5 focuses on the I/O overhead of the leaf nodes, we now consider the overhead of internal node accesses. The problem with dimensionality reduction is that a multidimensional range query is split into several $1d$ range queries. The number of $1d$ range queries has a significant impact on internal node accesses.

Lemma 4.6 The number of $1d$ range queries N_q and internal node accesses N_I increases with λ.

Proof. As proven in [71], the number of $1d$ range queries is about half the perimeter of the query range. Therefore, we have $N_q = 2 \cdot L_Q/2^{-\lambda}$. Suppose we do not modify the query algorithm of the B$^+$-tree. Each $1d$ range query starts from the root and searches for the lower boundary of the range. Then, the number of internal node accesses is:

$$N_I = N_q \cdot h = N_q \cdot \log_f N_L = 2^{\lambda+1} \cdot \log_f N_L \cdot L_Q$$

The height of the tree h is relatively stable when N_L varies. N_I is mainly determined by N_q. As λ increases, N_q increases, and so does N_I. Therefore, a larger value of λ indicates a heavier overhead on internal nodes. □

To evaluate the query performance of a tree-index, the number of leaf I/O N_L is usually the main concern. However, as we shall see in Section 4.8.2.1, the number of I/O varies slightly with a wide range of grid order (in between $[\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L, \frac{1}{2}\log_2 N])$. Consequently, the number of accesses to internal nodes dominates query performance in terms of query time. In addition, the effect of internal node accesses becomes even more important in a concurrent environment. When queries and updates arrive simultaneously, each access to the internal node requires locking the node and postponing concurrent updates accessing the same node. Therefore, according to Lemma 4.6 and Corollary 4.5, $\lceil\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L\rceil$ is the best value for λ that minimizes the query costs.

We also need to consider the effect of grid granularity on update cost. An update consists of deleting the old record and inserting the new record. To find the old record, $1 + \frac{1}{2}n_o$ nodes are searched on average, apart from the cost for node underflow. The old record is deleted and the node is written back. Despite the sporadic node overflow, inserting the new record incurs $(1 + n_o) + 1$ leaf and overflow node I/O. The insertion follows the overflow chain to obtain the last overflow node into which the new record is to be inserted. Writing it back contributes another I/O. The update cost increases with the average number of overflow pages. The cell size increases with smaller λ and so does the number of overflow pages.

In summary, a smaller λ within the range indicated in Corollary 4.5 leads to better query performance; nevertheless it incurs higher update costs. As we shall see in Section 4.8.1, the costs of query and update achieve the best

Table 4.2: Notations for analyzing time-related parameters

Symbol	**Description**
N_i	number of objects in BT_i
T_{up}	maximum update time interval
T	length of the time interval covers a sub-tree
T_{refi}	reference time of sub-tree BT_i
T_{li}	lower boundary of BT_i's time range
T_{ui}	upper boundary of BT_i's time range
tr	offset between T_{refi} to T_{li} ($T_{refi} - T_{li}$)
CQ_{avg}	average cost of a query
CU_{avg}	average cost of an update
CU_1	average cost of an update involving only one sub-tree
CU_2	average cost of an update involving both sub-trees
CU_m	average cost of migrating an object

trade-off when

$$\lambda = \lceil \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \rceil \tag{4.7}$$

In the analysis in this section, we have an implicit uniform assumption on query distribution. The performance of an index is sensitive to query distributions as well. Here, we simplify our cost model to uniform queries for the ease of analysis. We will show the impact of query distribution experimentally in Section 4.8.

4.6 Time Related Parameters

In this section, we will discuss the selection of two parameters of the ST^{2}B-tree: the reference time T_{ref} and the length of the time interval T covered by a sub-tree in the ST^{2}B-tree currently. Both are critical parameters for the "multi-tree" structure, having significant impact on the overall index performance. Table 4.2 shows the notations we use in the following analysis in addition to the notations in Table 4.1.

4.6.1 Reference Time of a Sub-tree: T_{ref}

In Section 4.3, as shown in Equation 4.1, we simply set the reference time T_{ref} to be the upper boundary of the time interval of a sub-tree. In fact, the reference time of a sub-tree does affect query performance. Since a query is enlarged into T_{ref} using maximum object velocity max_v, T_{ref} and max_v have a joint effect on query enlargement. max_v is a data-related parameter, over which the system has no control, while T_{ref} is a system parameter. In the following, we try to find an "optimal" value of T_{ref} that minimizes the average query cost CQ_{avg}. Since T_{ref} only affects query performance of the index and has no effect on the update performance, update cost is not our concern here.

Since each sub-tree covers a temporal range of length T, the query cost exhibits a periodic variation. Therefore, we investigate the average query cost CQ_{avg} in $[T_l, T_u)$, where T_l and T_u are the lower and upper boundary of the time interval covered by the younger sub-tree. Assuming that queries arrive evenly in time,

$$CQ_{avg} = \frac{1}{T_u - T_l} \int_{T_l}^{T_u} CQ(t)dt, \quad i = 1, 2, \ldots \tag{4.8}$$

where $CQ(t)$ is the cost of a query at time t. Given T as the length of the time interval,

$$CQ_{avg} = \frac{1}{T} \int_{T_l}^{T_l+T} CQ(t)dt, \quad i = 1, 2, \ldots$$

In Section 4.5, the number of leaf I/O for a query is estimated as $IO_L = N_L[L+L_Q]^2 = \frac{N}{f}[(\frac{f}{N})^{\frac{1}{2}}+L_Q]^2 = 1+2(\frac{N}{f})^{\frac{1}{2}}L_Q+NL_Q^2$. The cost is dominated by NL_Q^2. To simplify the analysis, we now estimate the query cost using NL_Q^2, i.e., the number of objects contained in the enlarged query L_Q. The experimental results in Section 4.8 empirically confirm the fact that the

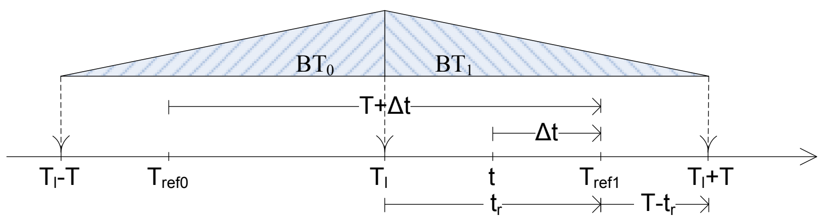


Figure 4.5: Graphic representations for time-related parameters

query cost is proportional to the number of objects in the enlarged query. Consider a square-sized range query with side length L_q enlarged with speed V along each side,

$$CQ(t) = N_0(L_q + V|t - T_{ref_0}|)^2 + N_1(L_q + V|t - T_{ref_1}|)^2, \tag{4.9}$$

where N_i represents the number of objects in the corresponding sub-tree BT_i at time t. $(L_q + V|t - T_{ref_i}|)^2$ is the enlarged query range of sub-tree BT_i. The cost of a query consists of two parts, i.e., the cost of processing the enlarged query over each sub-tree.

Combining Equations 4.8–4.9,

$$CQ_{avg} = \frac{1}{T}\int_{T_l}^{T_l+T} \left[(L_q + V|t - T_{ref_0}|)^2 N_0 + (L_q + V|t - T_{ref_1}|)^2 N_1\right] dt.$$

Without loss of generality, we assume that BT_0 is older than BT_1 in the current ST^{2}B-tree. Figure 4.5 illustrates the meaning of the notations. For BT_1, the query region is enlarged by Δt, where Δt is the time difference between T_{ref1} to the query time t, i.e., $\Delta t = t - T_{ref_1}$. Since $T_{ref1} = T_{ref0} + T$, the query region is enlarged by $T + \Delta t$ for BT_0. , then $\Delta t + T = t - T_{ref_0}$. Let $tr = T_{ref_1} - T_l$, which is the offset between the reference time to the lower boundary of BT_1's time range. When the query time t varies from T_l to $T_l + T$, Δt varies from $-tr$ to $T - tr$. Then, CQ_{avg} can be alternatively represented by the following equation:

$$CQ_{avg} = \frac{1}{T}\int_{-tr}^{T-tr} \left[N_0(L_q + V|\Delta t + T|)^2 + N_1(L_q + V|\Delta t|)^2\right] d\Delta t$$

1) The case when $\mathbf{T \leq T_{up}}$

Here, T_{up} is the maximum update time and N is the total number of objects. Suppose that object updates are uniformly distributed in T_{up} time. If $T \leq T_{up}$, the numbers of objects in BT_0 and BT_1 are:

$$\begin{cases} N_0 = N - \frac{t-T_l}{T_{up}} N \\ N_1 = \frac{t-T_l}{T_{up}} N \end{cases} \qquad T \leq T_{up}. \tag{4.10}$$

At the last transition time T_l, BT_1 is empty while BT_0 contains all N objects. Till timestamp t, $\frac{t-T_l}{T_{up}} N$ objects have been updated to BT_1 and $N - \frac{t-T_l}{T_{up}} N$ objects remain in the older sub-tree BT_0.

According to Equation 4.10, we have

$$CQ_{avg} = \frac{N}{TT_{up}} \int_{-tr}^{T-tr} \big[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) + (L_q + V|\Delta t|)^2 (\Delta t + tr)\big] \, d\Delta t$$

Lemma 4.7 When $T \leq T_{up}$, CQ_{avg} is minimized at

$$tr = T + \alpha T_{up} - \sqrt{\alpha^2 T_{up}^2 + \alpha T^2 - \alpha T_{up} T},$$

where $\alpha = 1 + VT/2L_q$ and $tr \in [T, 2T]$.

Proof.

$$\begin{aligned} CQ_{avg} &= \frac{N}{TT_{up}} \int_{-tr}^{T-tr} \big[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) \\ &\qquad + (L_q + V|\Delta t|)^2 (\Delta t + tr)\big] \, d\Delta t \\ &= \frac{N}{TT_{up}} E_1 + 2 L_q V \frac{N}{TT_{up}} E_2, \end{aligned}$$

where

$$E_1 = \int_{-tr}^{T-tr} \big[L_q^2 T_{up} + V^2 [T_{up} \Delta t^2 + (T^2 + 2T\Delta t)(T_{up} - \Delta t - tr)]\big] \, d\Delta t$$

$$E_2 = \int_{-tr}^{T-tr} (\Delta t + tr)|\Delta t|) d\Delta t + \int_{-tr}^{T-tr} (T_{up} - \Delta t - tr)|\Delta t + T| d\Delta t$$

Then, we consider the two parts of CQ_{avg} individually.

- As for $\mathbf{E_1}$:

$$E_1 = L_q^2 T_{up} T + V^2[T_{up} T tr^2 + T^3 tr - 3T_{up} T^2 tr + \frac{7}{3} T_{up} T^3 - \frac{7}{6} T^4]$$

It is easy to know that E_1 is minimized at $tr = T + \frac{T_{up}-T}{2T_{up}} T \in [T, 2T]$.

- As for $\mathbf{E_2}$:

$$E_2 = \begin{cases} -\frac{1}{2}(T - T_{up})T^2 + T_{up}(T - tr)T + \frac{1}{3}tr^3, & \text{if } tr \le T \\ \frac{1}{3}(T - tr)^3 + T_{up}(T - tr)^2 & \\ \qquad + T_{up}(T - tr)T + \frac{1}{2}(T_{up} - T + 2tr)T^2 - \frac{2}{3}T^3, & \text{if } T \le tr \le 2T \\ \frac{1}{2}(T - T_{up})T^2 - T_{up}(T - tr)T & \text{if } tr \ge 2T \end{cases}$$

Then, we know that

$$\min(E_2) = \begin{cases} \frac{1}{2}(T_{up} - \frac{1}{3}T)T^2, & \text{if } tr \le T, \text{when } tr = T \\ \frac{1}{2}(T_{up} + T)T^2, & \text{if } tr \ge 2T, \text{when } tr = 2T \end{cases}$$

Considering all the three cases, we can get the conclusion that E_2 should be minimized when $tr \in [T, 2T]$.

Finally, combining E_1 and E_2, if $T \le T_{up}$, CQ_{avg} should be minimized when $tr \in [T, 2T]$.

$$\begin{aligned} \min(CQ_{avg}) = & \\ & \frac{N}{TT_{up}} \left[L_q^2 T_{up} T + V^2 \left(T_{up} T tr^2 + T^3 tr - 3T_{up} T^2 tr + \frac{7}{3} T_{up} T^3 - \frac{7}{6} T^4 \right) \right] \\ & + 2L_q V \frac{N}{TT_{up}} \left[\frac{1}{3}(T - tr)^3 + T_{up}(T - tr)^2 + T_{up}(T - tr)T \right. \\ & \left. + \frac{1}{2}(T_{up} - T + 2tr)T^2 - \frac{2}{3}T^3 \right] \end{aligned}$$

Therefore, $\min(CQ_{avg})$ is minimized when $tr = T + \alpha T_{up} - \sqrt{\alpha^2 T_{up}^2 + \alpha T^2 - \alpha T_{up} T}$, where $\alpha = 1 + VT/2L_q$. $\square$

From Lemma 4.7, we see that CQ_{avg} is minimized when the reference time T_{ref_1} is sometime between $T_l + T = T_u$ and $T_l + 2T = T_u + T$. The "optimal" reference time varies with different queries, i.e., different L_q and V. To simplify the problem, we consider the case that T is equal to T_{up}.

Corollary 4.8 If $T = T_{up}$, all objects have been updated to the younger sub-tree in T time. CQ_{avg} is minimized when $tr = T$. T_{ref_1} is set to $T_l + T = T_u$, which is the upper boundary of its time range, the same as shown in Equation 4.1.

Proof. Derive from Lemma 4.7. □

Object migration can be reduced, or even avoided, by using the eager updates as introduced in Section 4.4. Since update cost is not our concern here, we can assume that all objects are eagerly updated to the younger sub-tree BT_1 within T time. As a result, although T is actually smaller than T_{up}, Corollary 4.8 also holds. The average query cost is minimized when T_{ref} is set to the upper boundary of the time range of a sub-tree, that is, when $tr = T$, we have:

$$CQ_{avg} = N\left[L_q^2 + \frac{V^2}{12}\left(4T^2 - 2T^2\frac{T}{T_{up}}\right) + \frac{L_qV}{6T}\left(6T^2 - 2T^2\frac{T}{T_{up}}\right)\right] \tag{4.11}$$

(2) The case when $\mathbf{T > T_{up}}$

If $T > T_{up}$, the numbers of objects in BT_0 and BT_1 are:

$$\begin{cases} N_0 = N - \frac{t-T_l}{T_{up}}N, N_1 = \frac{t-T_l}{T_{up}}N, & \text{if } T_{up} < t \le T_l + T_{up} \\ N_0 = 0, N_1 = N & \text{if } t \ge T_l + T_{up} \end{cases} \tag{4.12}$$

Before $T_l + T_{up}$, i.e., $t \le T_l + T_{up}$, N_0 and N_1 are the same with Equation 4.10. After $T_l + T_{up}$, i.e., $T_{up} < t \ge T_l + T_{up}$ all objects are moved to the younger sub-tree BT_1 and BT_0 is empty. According to Equation 4.12, we have

$$CQ_{avg} = \frac{N}{TT_{up}} \int_{-tr}^{T_{up}-tr} \left[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) \right.$$
$$\left. +(L_q + V|\Delta t|)^2 (\Delta t + tr) \right] d\Delta t + \frac{N}{T} \int_{T_{up}-tr}^{T-tr} \left[(L_q + V|\Delta t|)^2 \right] d\Delta t.$$

Lemma 4.9 If $T \geq T_{up}$, CQ_{avg} is minimized when $tr = \frac{1}{2}(T + T_{up})$. $T_{ref_1} = T_l + \frac{1}{2}(T + T_{up})$ and

$$CQ_{avg} = N \left[L_q^2 + \frac{V^2}{12} \left(T^2 + T_{up}^2 \right) + \frac{L_q V}{6T} \left(3T^2 + T_{up}^2 \right) \right]. \tag{4.13}$$

Proof.

$$CQ_{avg} = \frac{N}{TT_{up}} \int_{-tr}^{T_{up}-tr} \left[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) \right.$$
$$\left. +(L_q + V|\Delta t|)^2 (\Delta t + tr) \right] d\Delta t + \frac{N}{T} \int_{T_{up}-tr}^{T-tr} \left[(L_q + V|\Delta t|)^2 \right] d\Delta t$$
$$= \frac{N}{TT_{up}} E_1 + 2L_q V \frac{N}{TT_{up}} E_2 + \frac{N}{T} E_3,$$

where

$$E_1 = \int_{-tr}^{T_{up}-tr} \left[L_q^2 T_{up} + V^2 [T_{up} \Delta t^2 + (T^2 + 2T\Delta t)(T_{up} - \Delta t - tr)] \right] d\Delta t,$$
$$E_2 = \int_{-tr}^{T_{up}-tr} (\Delta t + tr)|\Delta t|) d\Delta t + \int_{-tr}^{T_{up}-tr} (T_{up} - \Delta t - tr)|\Delta t + T| d\Delta t,$$
$$E_3 = \int_{T_{up}-tr}^{T-tr} \left[(L_q + V|\Delta t|)^2 \right] d\Delta t.$$

The three components are minimized as follows.

$$\min(E_1) = L_q^2 T_{up}^2 + \frac{V^2}{12} T_{up}^2 [T_{up}^2 - 2TupT + 3T^2], \text{when } tr = \frac{1}{2}(T + T_{up})$$
$$\min(E_2) = \frac{1}{6} T_{up}^2 (3T - T_{up}), \text{for } \forall tr \in [T_{up} \leq T]$$
$$\min(E_3) = L_q^2 (T - T_{up}) + \frac{V^2}{12} (T - T_{up})^3 + 2L_q V \frac{1}{4} (T - T_{up})^2, \text{when } tr = \frac{1}{2}(T + T_{up})$$

Therefore, CQ_{avg} is minimized, when $tr = \frac{1}{2}(T + T_{up})$ and

$$\min(CQ_{avg}) = N[L_q^2 + \frac{V^2}{12}(T^2 + T_{up}^2) + \frac{L_q V}{6T}(3T^2 + T_{up}^2)].$$

$\square$

In summary, for the purpose of minimizing the average query cost, we set the reference time of a sub-tree as follows:

$$T_{ref} = \begin{cases} T_l + T & \text{if } T \leq T_{up}, \\ T_l + \frac{1}{2}(T + T_{up}) & \text{if } T > T_{up}. \end{cases} \tag{4.14}$$

Note that when $T \leq T_{up}$, Equation 4.14 is the same as Equation 4.1.

4.6.2 The Length of the Time Interval of a Sub-tree: T

While the reference time T_{ref} affects the query performance only, the length of the time interval T has an impact on both the queries and updates as discussed in Section 4.4.1.

4.6.2.1 Effect of T on Queries

Given that T_{ref} is determined according to Equation 4.14, the average query cost is as follows:

$$CQ_{avg} = \begin{cases} N\left[L_q^2 + \frac{V^2}{12}(4T^2 - 2T^2\frac{T}{T_{up}}) + \frac{L_qV}{6T}(6T^2 - 2T^2\frac{T}{T_{up}})\right], & \text{if } T \leq T_{up} \\ N\left[L_q^2 + \frac{V^2}{12}(T^2 + T_{up}^2) + \frac{L_qV}{6T}(3T^2 + T_{up}^2)\right], & \text{if } T > T_{up} \end{cases}$$

The average query cost CQ_{avg} always increases with larger T. In order to minimize the query cost, T should be as small as possible.

4.6.2.2 Effect of T on Updates

Now let us consider the update, concerning the average cost of each "real" update, i.e., update issued by the object actively.

Lemma 4.10 When $T \leq T_{up}$, the average cost of an update is:

$$CU_{avg} = CU_2 + \frac{T_{up} - T}{T}CU_m$$

Proof. Suppose that objects update evenly in T_{up} time. $\frac{1}{T_{up}}N$ objects update at each timestamp. Just before the transition, a total number of $\frac{T}{T_{up}}N$ objects have been deleted from the older sub-tree and inserted into the younger sub-tree. The remaining N-$\frac{T}{T_{up}}N$ objects need to be migrated. The average cost of a "real" update is

$$\begin{aligned} CU_{avg} &= \left[CU_2\frac{T}{T_{up}}N + \frac{T_{up}-T}{T_up}CU_mN\right]/\frac{T}{T_{up}}N \\ &= CU_2 + \frac{T_{up}-T}{T}CU_m \end{aligned}$$

□

Lemma 4.11 When $T > T_{up}$, the average cost of an update is:

$$CU_{avg} = \frac{T_{up}}{T}CU_2 + \frac{T-T_{up}}{T}CU_1$$

Proof. Till $t = T_{up}$, all objects have been updated to the younger subtree actively and the total cost is CU_2N. After T_{up} and right before the transition, all subsequent updates operate on the younger sub-tree only. In the remaining $T - T_{up}$ time, there are a total number of $\frac{T-T_{up}}{T_{up}}N$ updates and the cost is $CU_1\frac{T-T_{up}}{T_{up}}N$. As a result, the average update cost is

$$\begin{aligned} CU_{avg} &= \left[CU_2N + CU_1\frac{T-T_{up}}{T_{up}}N\right]/\frac{T}{T_{up}}N \\ &= \frac{T_{up}}{T}CU_2 + \frac{T-T_{up}}{T}CU_1 \end{aligned}$$

□

Combining Lemma 4.10 and Lemma 4.11,

$$CU_{avg} = \begin{cases} CU_2 + \frac{T_{up}-T}{T}CU_m, & \text{if } T \leq T_{up} \\ \frac{T_{up}}{T}CU_2 + \frac{T-T_{up}}{T}CU_1, & \text{if } T > T_{up} \end{cases}$$

The three types of updates incur different costs, as we will see in Section 4.8.3. In essence, $CU_2 > CU_1 > CU_m$. If $T \leq T_{up}$, the average update

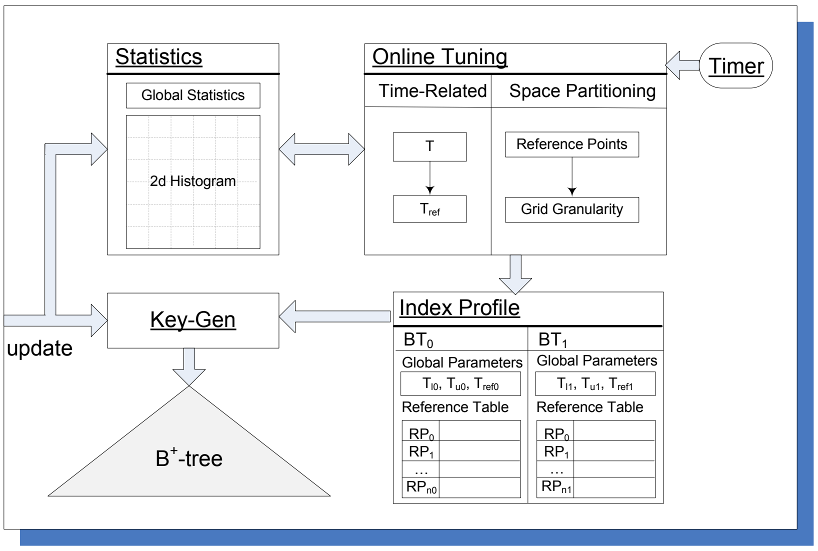


Figure 4.6: Online tuning framework

cost is minimized when no migration happens at $T = T_{up}$. Otherwise, if $T > T_{up}$, it is better to maximize the percentage of single tree updates, and T should be as large as possible.

The length of sub-tree time interval T has an opposite effect on query and update costs. In Section 4.8.5, we will investigate the effect of T with respect to various query/update ratios via empirical studies.

4.7 Self-Tuning of the ST^{2}B-tree

As discussed in Section 4.3, the multi-tree design makes the ST^{2}B-tree feasible for tuning. Section 4.5 and Section 4.6 provide guidelines for choosing optimal values for the various parameters used in the ST^{2}B-tree. We now introduce how self-tuning is realized on the ST^{2}B-tree.

Figure 4.6 shows the tuning framework of the ST^{2}B-tree. The tuning framework adds four components on top of the underlying DBMS: the Index Profile, the Key-Gen, the Statistics and the Online Tuning.

4.7.1 Index Profile

Index Profile maintains the current settings of the ST^{2}B-tree. As shown in Figure 4.6, for each sub-tree BT_i, the global profile contains three parameters: the time interval of the sub-tree $[T_{l_i}, T_{u_i}]$ and the reference time T_{ref_i}. If we do not want to tune T—the length of the time interval covering a sub-tree—there is no need to maintain T_{l_i}, T_{u_i} and T_{ref_i}. When T is fixed, T_{l_i} and T_{u_i} can be derived from the current time, and T_{ref_i} is known according to Equation 4.1.

Besides global parameters, Index Profile keeps a reference table for the reference points in each sub-tree. Each reference point RP_j has an entry in the reference table, including its position, size of its grid G_j and granularity of its grid λ_j. If eager updates are allowed, Index Profile also keeps the degree of eagerness $\mathcal{D}_e$ currently used by the index.

4.7.2 Key-Gen

The Key-Gen module works as an interface between the ST^{2}B-tree and the underlying B$^+$-tree. On receiving an object update, it reads the index settings from the Index Profile that are necessary for computing KEY_{ST^2}, including the reference time and reference points of the younger sub-tree. It then calculates KEY_{ST^2} according to Equation 4.4. Finally, the update is performed over the B$^+$-tree with KEY_{ST^2} and new location and velocity of the object.

4.7.3 Statistics

The purpose of tuning is to make the index adaptive to the workload. The Statistics module maintains statistics about the workload in the current time interval, i.e., the time interval of the younger sub-tree. The statistics will be used to tune the index at the next transition time. While dealing with an object update, the statistics is updated accordingly. Right after the tuning process finishes, the statistics is cleared for the next time interval.

Generally, two kinds of statistics are maintained. The global statistics contains statistics about all objects and queries in the entire space. In order to tune the length of sub-tree time interval T as discussed in Section 4.6.2, the following statistics are required: 1) the total number of objects N, 2) maximum update time T_{up}, 3) three types of update cost CU_1, CU_2 and CU_m as defined in Table 4.2, 4) update/query ratio R, and 5) average query side length L and max_v for query enlarging. In addition, if the eager update technique is applied, the global statistics maintains one more field which is the number of objects N_m that are migrated at the last transition time.

Besides the global statistics, we use a $2d$ histogram to maintain regional statistics. The $2d$ histogram consists of $n \times n$ buckets, each of which maintains the statistics of a cell in the space. Specifically, the entire space is partitioned evenly into $n \times n$ square sized cells (different from the cells used for indexing in Section 4.3.1). In the histogram, the bucket of cell c_{ij}, where i and j denote the row and column number of the cell, is a tuple $h_{ij} = (\overrightarrow{p_{ij}}, n_{ij})$, where n_{ij} is the estimated number of objects in that cell and $\overrightarrow{p_{ij}}$ is the centroid of objects in the cell. The statistics maintained by the histogram summarizes the distribution in difference regions (cells) all over the space, which is necessary for the tuning purpose.

4.7.3.1 Histogram Maintenance

Suppose the current time interval is $[T_l, T_u]$ and the reference time for the younger sub-tree is T_{ref}. The next transition time should be T_u, when the online tuning process starts. The tuning process aims to find the best space partitioning for the next time interval $[T_u, T_u + T]$. During that time, all objects will be indexed at a new reference time T'_{ref}. Therefore, objects are estimated and counted at the time instance of T'_{ref} in the histogram.

Specifically, given an update $\langle oid, \overrightarrow{p}, \overrightarrow{v}\rangle$ at t_{up}, the histogram is updated as follows:

1. Estimate o's position at the time instance T'_{ref}:

$$\overrightarrow{p}'' = \overrightarrow{p} + \overrightarrow{v} \cdot (T'_{ref} - t_{up}) = \overrightarrow{p} + \overrightarrow{v} \cdot (T_{ref} + T - t_{up})$$

   Since the optimal length of the next time interval T is determined only when the tuning process completes, we simply assume that T will not change for the next time interval while maintaining the histogram. T'_{ref} can be determined accordingly, where $T'_{ref} = T_{ref} + T$. Then,

$$\overrightarrow{p}'' = \overrightarrow{p} + \overrightarrow{v} \cdot (T_{ref} + T - t_{up})$$

   Note that if the update affects only the youngest sub-tree, the old record is deleted from the youngest sub-tree. The statistic should be updated in a reverse manner first, i.e., re-computing the centroid by subtracting old position of the object and decreasing the number of objects.

2. h_{ij} of the cell that $\overrightarrow{p}''$ belongs to is updated

$$\overrightarrow{p_{ij}} = \frac{n_{ij} \cdot \overrightarrow{p_{ij}} + \overrightarrow{p}''}{n_{ij} + 1},$$

$$n_{ij} = n_{ij} + 1.$$

Thus, $\overrightarrow{p_{ij}}$ is always the centroid of all objects estimated to be in cell c_{ij} at T'_{ref}.

$$\overrightarrow{p_{ij}} = \frac{\sum_{k=1}^{n_{ij}} \overrightarrow{p''_k}}{n_{ij}}$$

4.7.4 Online Tuning

The Online Tuning module is responsible for executing the tuning process. At each transition time, i.e., the upper boundary of current time interval, the Timer triggers the Online Tuning module to start the tuning procedure. Based on the statistics maintained, it determines new parameters for the ST^{2}B-tree. At the end of the tuning procedure, parameters in the Index Profile are updated accordingly.

As shown in Figure 4.6, the tuning can be applied in two aspects. On the one hand, we can tune the length of the sub-tree time interval and the reference time. On the other hand, we can improve the space partitioning by adjusting the set of reference points and their grid granularity.

4.7.4.1 T and T_{ref}

As shown in Section 4.6.2, the value of T affects update and query performance in opposite ways. In order to get the best tradeoff, we can also tune the value of T with respect to the latest query and update loads. With the global statistics maintained by the Statistics Module, the Online Tuning module can estimate the average query and update costs. In order to improve query performance, the system can tune down the value of T; in order to minimize the average update cost, the system can tune up the value of T. The final decision is made depending on the main concern of the system, either query response time or supporting more updates/objects.

Given that T_{up} is maintained as a global statistics in the Statistics module, once T is determined, we can get the T_{ref} that minimizes average query cost according to Equation 4.14.

4.7.4.2 Reference Points and Grid Granularity

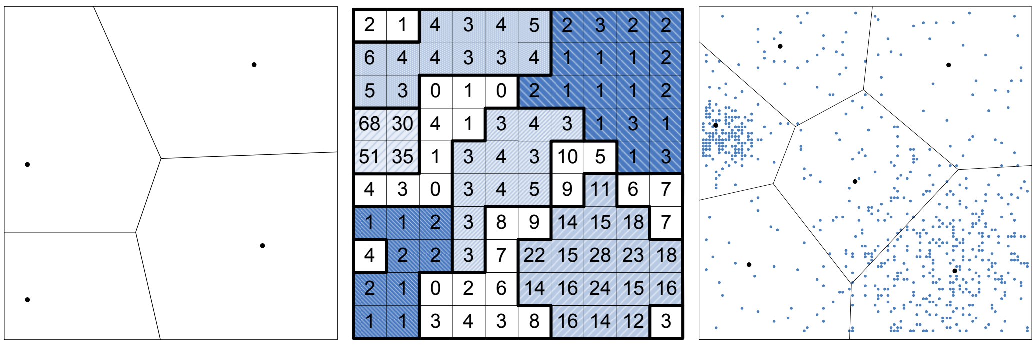
The ST^{2}B-tree can dynamically adjust to different space partitioning. However, since the data is highly dynamic, it is difficult to find an optimal partitioning. Even if such an optimal partitioning exists, it is costly to discover it; moreover, its optimality is bound to be short-lived because of the dynamics of the system. Therefore, we aim to rapidly find a moderate set of reference points that roughly, but effectively, partitions the space based on density differences, so that the tuning procedure can be done online without deferring any other operations.

- Finding Reference Points via Region Growing

The tuning procedure is triggered by the timer at each transition time. With the histogram, e.g., Figure 4.7(b), we identify dense and sparse regions by region growing. Recall that the Statistic module maintains a histogram with $n \times n$ buckets, each of which stores information about a cell in the space. Note here these cells of the histogram are detached from the cells which are used by the reference points for the purpose of indexing. The cells of the histogram are over all the space simply to have an idea of density distribution.

Region growing is a technique widely used in image segmentation for finding adjacent similar pixels. In image processing, similarity of pixels is defined over color, brightness, etc. For us, each cell in the histogram acts as a pixel. Two cells are said to be similar if they have a similar number of objects. Algorithm 4.5 shows the procedure of region growing.

First, we take the previous reference points as the seeds for growing. Since the distribution and density of moving objects change gradually, the positions of the reference points should move slightly. Starting from cell c,



(a) Old reference points (b) Region growing (c) New reference points

Figure 4.7: Example of region growing for finding reference points

Algorithm 4.5: Region Growing

```
Input  : Current reference points $\{RP_1, \dots, RP_k, \dots\}$
Output: Set of regions $\{R\}$
1  $RS = \emptyset$;
2  foreach previous reference points $RP_k$ do
3      $c$=the cell that contains $RP_k$;
4      if $c$ is unmarked then
5          Add $c$ to a new region $R$;
6          Mark $c$;
7          Growing ($c$, $R$);
8          Add $R$ to $RS$;
9  while there is $c$ that is unmarked do
10     Add $c$ to a new region $R$;
11     Mark $c$;
12     Growing ($c$, $R$);
13     Add $R$ to $RS$;
14 return $RS$;
```

we examine its neighboring cells. If a neighboring cell c' does not belong to any existing region and $\frac{|c.n - R.max_n|}{R.avg_n} \leqslant \epsilon$ and $\frac{|c.n - R.min_n|}{R.avg_n} \leqslant \epsilon$, c' is added into the region R of c. $R.max_n$, $R.min_n$, $R.avg_n$ are the maximum, minimum and average number of objects of cells in R currently. ϵ is a predefined threshold that defines similarity. The growing procedure terminates when all the cells belong to some region.

The output of the region growing algorithm is a set of regions that have

Algorithm 4.6: Growing

```
Input  : $c$, $R$
Output: $R$
1 foreach neighbor cell $c'$ of $c$ do
2     if $c'$ is unmarked and $\frac{|c.n - R.max_n|}{R.avg_n} \leqslant \epsilon$ and $\frac{|c.n - R.min_n|}{R.avg_n} \leqslant \epsilon$ then
3         Add $c'$ to $R$;
4         Mark $c$;
5         Growing ($c'$, $R$);
```

similar object density. The centers of the resultant regions are marked as the reference points. More specifically, a resultant region R consists of several adjacent cells. The center of R, i.e., the reference point RP, is calculated as:

$$RP.\overrightarrow{p} = \frac{\sum_{c_{ij} \in R} n_{ij} \cdot \overrightarrow{p_{ij}}}{\sum_{c_{ij} \in R} n_{ij}}$$

The object density for RP is

$$RP.\rho = \frac{\sum_{c_{ij} \in R} n_{ij}}{|R|}$$

where $|R|$ is the number of cells in R.

Figure 4.7 shows a running example of finding reference points by Algorithm 4.5. Region-growing starts with four previous reference points as shown in Figure 4.7(a). Figure 4.7(b) shows the resultant regions ($\epsilon = 1$) filled with different patterns and enclosed by thick lines. Then all regions that contain no more than 3 cells are pruned. Finally, we get 6 regions (shaded regions). As shown in Figure 4.7(c), the region growing method roughly identifies 6 reference points, which further partition the space into disjoint Voronoi cells. We use the cells in R (with a similar number of objects) for estimating density for RP, ignoring the other cells which are noted as noises. When the reference points are determined, the grid granularity for each RP_i can be computed according to Equation 4.7.

- Alternative Methods

Intuitively, we can also apply density-based clustering methods to partition the space. Examples of density based spatial clustering methods include DBSCAN [31] and OPTICS [7]. However, none of these existing methods facilitates online tuning. First, they can only find dense areas; sparse regions may be completely disregarded. Second, density-based clustering methods are time-consuming. DBSCAN takes seconds to cluster a few thousand data points, even in the presence of a spatial index. While the tuning procedure is running, all updates have to be suspended. An update costs a few milliseconds over a B$^+$-tree on average, which means that thousands of updates may need to be postponed during the tuning procedure. This is not acceptable for online tuning of an index meant to support high update load.

Yet another practical approach to select reference points is to consider the characteristics of real world moving objects, e.g., city traffic. In reality, hotspots remain hotspots, no matter how many objects there are. Prominent landmarks, such as major road junctions and commercial centers, always attract more vehicles than the other places. These hotspots can be used as reference points most of the time. On the other hand, we can also discover that real traffic often exhibits seasonal patterns, either daily, weekly or monthly. For example, many vehicles move toward the downtown area of a city between 8 to 9am and travel back to the residential suburbs at around 5 to 6pm every weekday. Based on the above observations, reference points can also be computed off-line based on historical data. We can compute and preserve the reference points for each time slice that the data shows similar patterns regularly. An online tuning module can then choose the set of preset reference points of the right slice of time as the tree rolls over with time. However, this method is only acceptable for a fairly stable environment. The preset settings may not be suitable once the environments changes for a non-trivial amount of time. For instance, road construction may last long enough to disrupt the system's responsiveness but not long enough to warrant changes to the indexes. It is a better choice if the tuning process can select the reference points and other parameters based on the latest workload at real-time and incurring imperceptible overhead.

4.7.4.3 Degree of Eagerness $\mathcal{D}_e$

Last but not least, if eager update is employed, we can also vary the degree of eagerness $\mathcal{D}_e$ according to the number of objects being migrated N_m last time. If $N_m \leq \xi$, we may increase $\mathcal{D}_e$ by one; otherwise decrease $\mathcal{D}_e$. Here, ξ is a user defined threshold, which depends on the latency that the system can afford to wait for object migration.

4.8 Performance Evaluation

In this section, we report the results of an exhaustive experiential study on the ST^{2}B-tree and the online-tuning framework. We first specify some settings of the experiments, and then show and explain the experimental results.

4.8.1 Experiment Setup

The experiments in this section follow the standard evaluation procedure introduced the benchmark as presented in Chapter 3. Using the data generator included in the benchmark, we generate two kinds of workload, uniform and Gaussian datasets. Generally, the uniform datasets are used for evalu-

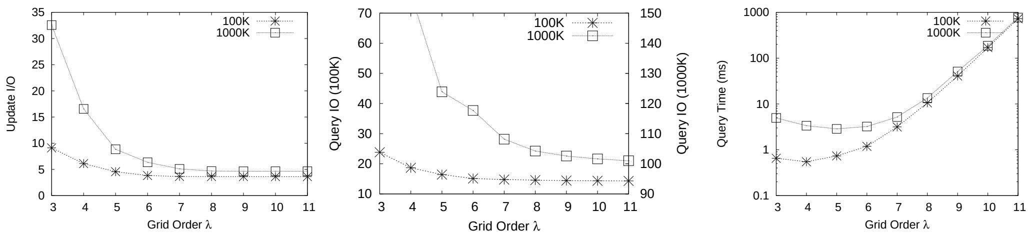
Table 4.3: Parameters, their value ranges and default values

Parameter	**Setting**
Space domain	**100,000×100,000m^2**
Data size	**100K**, ..., 1M
Maximum object speed	10m/tu, ..., **100m/tu**
Maximum update interval T_{up}	**120tu**
Range query size	**1,000×1,000m^2**, ...,10,000×10,000m^2
Number of neighbors, k	**10**, ..., 100
Query Predictive Time	0tu, 10tu, ..., **60tu**, ..., 120tu
Time duration	**240tu**, 1200tu
Buffer size (number of pages)	**50**
Disk page size (KB)	**4**
Number of hotspots	**10**
Query/Update ratio	100:1, ..., **1:100**, ..., 1:10,000
Number of threads	1, 2, 4, ..., 128, **10**
Length of sub-tree time interval T	$0.1T_{up}$, $0.2T_{up}$, ..., $0.9T_{up}$, T_{up}
Offset of reference time T_{ref}-T_l	0, $0.2T$, $0.4T$, ..., T, $1.2T$, ..., $2T$
Degree of eagerness $\mathcal{D}_e$	**0**, 1, 2, 4, 8, 32, 64

ating the benefits of tuning time-related parameters, the eagerness of eager update strategy, etc. Since we intend to investigate the imbalance and changes in the workloads, we use Gaussian workloads to test the impact on the density-based space partitioning scheme of the ST^{2}B-tree. Specifically, the data generator randomly selects some points in the space as hotspots. Each hotspot uses a Gaussian distribution to generate objects around it. The queries can be either uniformly distributed or not; in the case of non-uniform queries they follow the same distribution as the objects. In order to evaluate the ST^{2}B-tree and the self-tuning framework, we select four of the six moving-object indexes that have been studied in Section 3.5, i.e., the B^x-tree [50], the B^{dual}-tree [123], the TPR*-tree [100] and STRIPES [78], and compared them with the ST^{2}B-tree. These four indexes represent indexes of different underlying structure. The same implementations and settings as in used Section 3.5 were adopted in the experiments reported in this section.

Table 4.3 summarizes the settings of workload used in the experiments, where default values of variable parameters are shown in bold. tu is short for timestamp. Parameters are shown in groups based on their meanings. The experiments use the same range and default settings as the benchmark. In order to evaluate the tuning effect, we vary the tunable parameters such as T, T_{ref}, $\mathcal{D}_e$. Parameter ϵ for region growing is fixed at 1 in all the experiments. The execution time for region growing increases with the number of cells in the 2d-histogram. In our experiments, we use 100×100 cells in the 2d-histogram to keep the tuning time to be lower than 5ms, so that the system will not be stalled by the tuning process.

All the indexes are implemented in C++. All experiments are conducted on a IBM ThinkPad with Pentium M 1.86GHz processor, 1.0GB RAM and 60G SATA Disk, running Windows XP. All the results are the average for 10 runs. As the authors in [50, 100, 101, 18] do in the experiments, we also use a block file with 4KB blocks to simulate a disk with 4KB pages. A LRU buffer with 50 pages is used. We use a tool, *Lavalys*® EVEREST, to test the random I/O speed on disk and memory respectively. The speeds of read and write of 4KB clusters on disk are ∼20MB/s and ∼12MB/s respectively. The speeds of memory reads and writes are ∼3356MB/s and ∼2770MB/s. The simulating experiments count the exact I/Os with our experimental settings. However, since we cannot prohibit the operating system from using the physical memory, the results actually shows the time for memory I/Os. Since all the indexes are implemented with the same disk manager, we did a fair lateral comparison between all indexes by simulation.



(a) Update I/O (b) Query I/O (c) Query time

Figure 4.8: Effect of the grid granularity

4.8.2 Tunable Parameters

Before comparing with other moving-object indexes, we first investigate the effect of the tunable parameters discussed in Section 4.5 and Section 4.6. In this set of experiments, we use uniform datasets to get a clearer understanding of the effect of these parameters.

4.8.2.1 Effect of Grid Granularity

We first study the effect of grid granularity empirically to verify the analysis in Section 4.5 and to determine the optimal grid order. We test on two uniform workloads, including 100K and 1M moving objects. Since the objects are uniformly distributed, the object density in the whole space is the same. The ST^{2}B-tree will have only one reference point at the center of the space. The query workload consists of 100 uniform range queries with default settings.

Figure 4.8 illustrates the overall performance with grid order λ varying from 3 to 11. We make the following observations:

1. The update I/O increases significantly when $\lambda \leq \frac{1}{2}(log_2 N - log_2 C_L)$ (about 7 for 1M objects) and hardly changes with finer partitioning.

2. When $\lambda \leq \frac{1}{2}(log_2 N - log_2 C_L)$, the query I/O increases with smaller

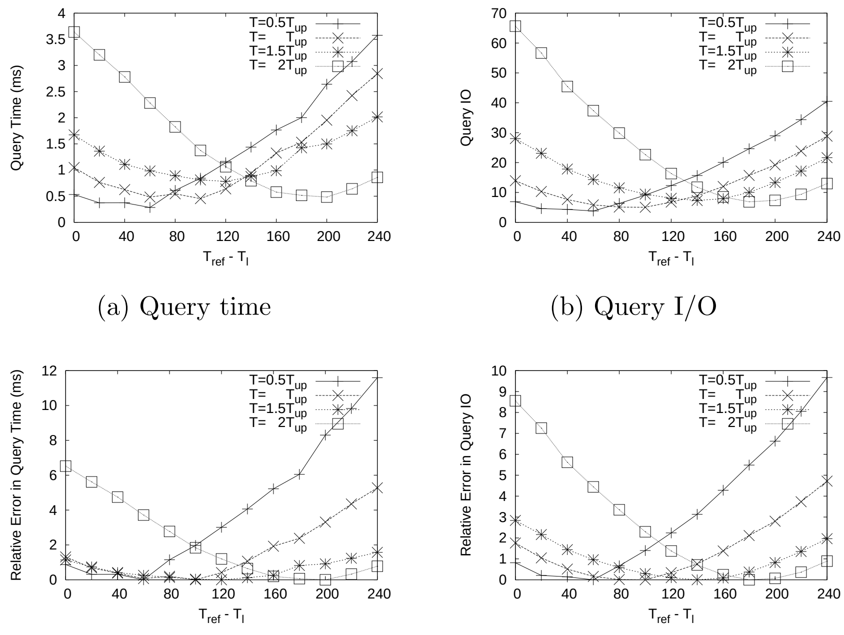
value of λ. However, with larger λ, the number of average query I/O hardly changes. In Figure 4.8(b), the query I/O of the 1M dataset is labeled with the right y-axis with different values, because we do not intend to compare the I/O of 100K and 1M datasets but to show the trend of I/O changes with grid order λ.

3. The query processing time increases dramatically with a larger λ due to the increasing number of key retrievals. Note that the query processing time increases when λ becomes smaller. This can be explained by the fact that with excessively large grid cells, very few number of $1d$ searches are required. However, the I/O cost increases significantly, which contributes more to the processing time. In addition, it incurs more time to prune away a large number of false positives when the grid cells are too large.

Based on the above observations and the analysis in Section 4.5, we set the grid granularity λ to $\lceil \frac{1}{2}(log_2N - log_2C_L) \rceil$ as in Equation 4.7, which results in the best tradeoff between update and query performance. In the remainder of this section, this rule is applied to the selection of global space partitioning for the B^x-tree; while for the ST^{2}B-tree, it guides the selection of grid granularity of each reference point.

4.8.2.2 Effect of Reference Time

We now see the effect of the reference time. As discussed in Section 4.6.1, the reference time only affects the query performance. Therefore, we investigate the query time and I/O with respect to different choices of reference time. We vary the value of $T_{ref} - T_l$ from 0 to $240tu$ ($2T_{up}$), i.e., the offset from the reference time to the lower boundary of the time interval of the subtree. The maximum update time T_{up} is fixed at the default value of 120tu.



(a) Query time

(b) Query I/O

(c) Relative error in query time

(d) Relative error in query I/O

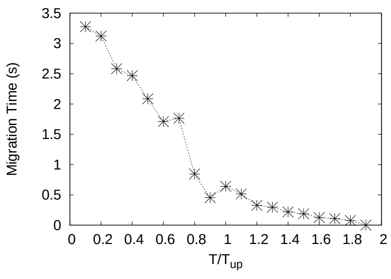
Figure 4.9: Effect of the (relative) reference time

Figure 4.9 shows the average cost when the length of the sub-tree time interval T is set to $0.5T_{up}$, T_{up}, $1.5T_{up}$ and $2T_{up}$.

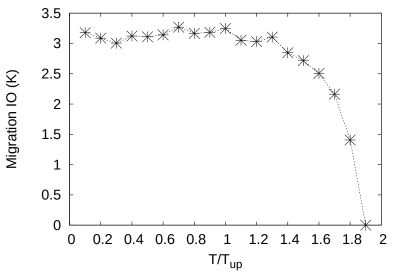
As shown, when $T = 0.5T_{up}$, the query cost is minimized when $T_{ref} - T_l = 60$, which is equal to T. When $T \geq T_{up}$, i.e., $T = T_{up}, 1.5T_{up}$ and $2T_{up}$, the query cost is minimized when $T_{ref} - T_l = 120tu, 140tu, 200tu$ respectively. The results in Figure 4.9 verify the analysis in Section 4.6.1. The optimal reference time is selected as Equation 4.14. While T_{ref} deviates from the optimal value, the query cost increases monotonically.

4.8.2.3 Effect of the Length of sub-tree Time Interval T

In this experiment, we investigate the problem caused by object migration during rollover. The length of the sub-tree time interval T varies from $0.1T_{up}$ to T_{up} (if T is larger than T_{up}, no migration occurs). In order to see the pure effect of object migration, eager update is disabled.

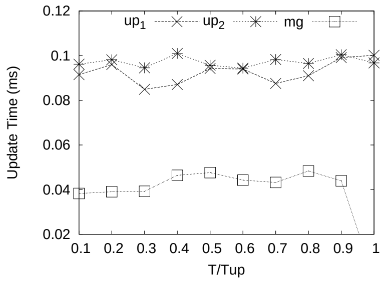


(a) Migration time

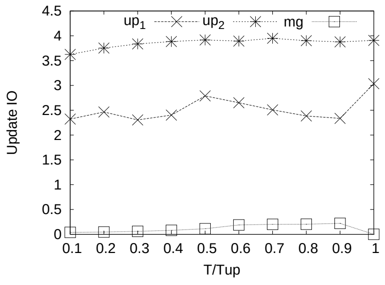


(b) Migration I/O

Figure 4.10: Effect of sub-tree life time T on object-migration



(a) Update time

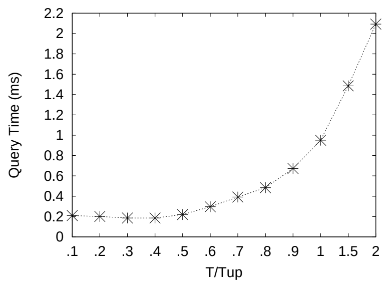


(b) Update I/O

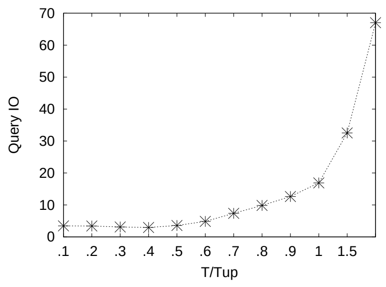
Figure 4.11: Effect of sub-tree life time T on updates

Figure 4.10 shows the total running time and number of I/Os of one migration. As expected, the total migration time is proportional to the number of un-updated objects left in the old sub-tree, which decreases with larger T. When T is only 10% of the maximum update time T_{up}, there is a delay of more than 3s to finish the migration. If T increases to about $90\%T_{up}$, the migration time is reduced to around 0.5s. Note that the migration I/Os is less affected when T is smaller than $90\%T_{up}$. This is because migration is done by deleting objects from the older sub-tree and inserting them into the younger sub-tree in a batch mode, which saves a number of I/Os, especially with the help of the LRU buffer.

Figure 4.11 illustrates the corresponding costs for different types of updates, where "up$_1$" represents updates that involve only one sub-tree, "up$_2$"



(a) Query time



(b) Query I/O

Figure 4.12: Effect of sub-tree life time T on queries

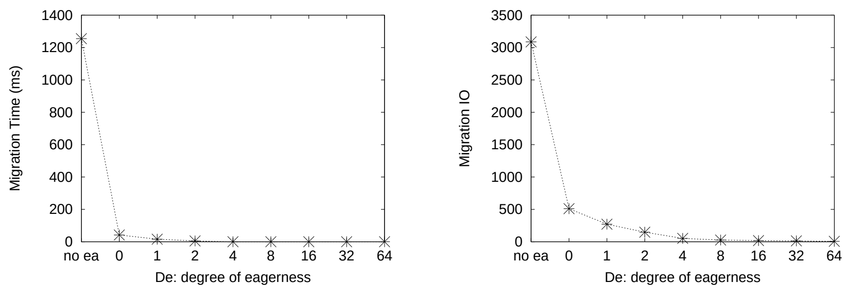
represents updates that affect both sub-trees and "mg" denotes the amortized cost of migrating one object. The amortized migration cost is the minimum, i.e., around 0.04ms and no more than 0.1 I/O access. A single-sub-tree update (up_1) incurs about 2/3 I/Os of an update involving both sub-trees.

Although the total migration time seems to be not very long (only a few seconds), it is still not acceptable in a continuous running MOD. As shown in Figure 4.11, the average update time is only about 0.1 millisecond. During the migration time, the MOD is capable of handling tens of thousands of updates.

Finally, Figure 4.12 shows the effect of T on query performance. As expected, the query cost, including the CPU time and the number of I/O accesses increase with T.

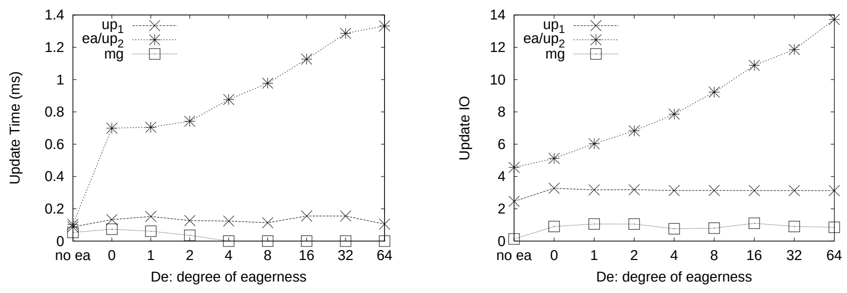
4.8.3 Effect of Eager Updates

Now we proceed to investigate the effect of eager updates. T is set to be half of the maximum update time, i.e., $0.5T_{up}$. In this set of experiments, we vary the degrees of eagerness $\mathcal{D}_e$ from 0 to 64. The corresponding benefit on migration cost and overhead on update cost are shown in Figure 4.13 and



(a) Migration time (b) Migration I/O

Figure 4.13: Effect of the degree of eagerness $\mathcal{D}_e$ on object-migration



(a) Update time (b) Update I/O

Figure 4.14: Effect of the degree of eagerness $\mathcal{D}_e$ on updates

Figure 4.14 respectively. The leftmost point (marked as "no ea") in both figures represents the corresponding result of the case where eager update is disabled.

First, Figure 4.13 shows the benefit of eager updates in terms of total time and I/O cost of an migration. If there are no eager updates, it requires about 1.3 seconds for migrating objects during rollover. However, when eager update is enabled, the migration cost decreases dramatically. Specifically, when $\mathcal{D}_e = 0$, which means the smallest degree of eagerness, the migration time is reduced to about 50ms. When $\mathcal{D}_e = 4$, the migration time is around 5ms only, which is much more acceptable considering the continuity of a MOD.

Eager update technique undoubtedly relaxes the constraints on T by

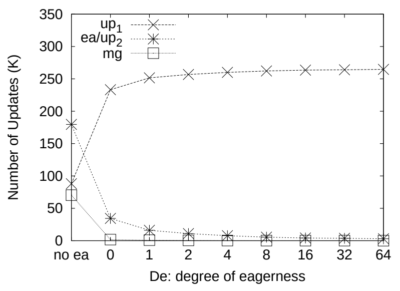


Figure 4.15: Effect of the degree of eagerness $\mathcal{D}_e$ on the number of updates

reducing the migration cost, but at the expense of an increase in update cost. Figure 4.14 shows the average update cost. As shown, the migrating cost (“mg”) is still the smallest. The cost of single tree update (“up_1”) is more or less the same as that in Figure 4.11, since eager update is only applicable to updates involving two sub-trees as described in Algorithm 4.4. There is a significant increase on the average cost of updates (“up_2”) when eager update is applied. When $\mathcal{D}_e = 0$, the update is about 7 times slower than normal update (the one marked with “no mig”), while the increase in the number of I/Os is no more than 1. Intuitively, when $\mathcal{D}_e = 0$, the update cost should not increase since the deletion and insertion access the same tree nodes as a normal update. In practice, by deleting and inserting more records from a leaf may cause leaf merge or split, which incurs additional I/Os. With larger $\mathcal{D}_e$, the I/O cost increases more significantly, since the insertions may happen in leaf nodes other than the leaf node where the core object, i.e., the object causes the eager update, is inserted.

Figure 4.15 shows the number of updates of different types. With higher degree of eagerness, i.e., larger $\mathcal{D}_e$, not only the number of objects to be migrated decreases, but also the number of updates involving both sub-trees. Through eager updating, a number of updates are avoided and reduced into single sub-tree updates. Therefore, as we can see, the number of single tree updates (“up_1”) increases with $\mathcal{D}_e$.

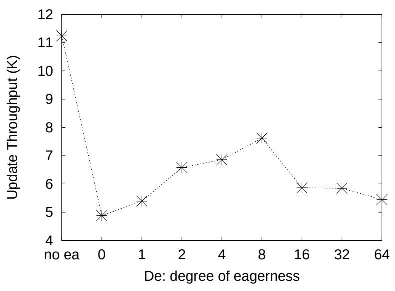


Figure 4.16: Effect of the degree of eagerness $\mathcal{D}_e$ on update throughput

Finally, Figure 4.16 shows the effect of $\mathcal{D}_e$ on the total update throughput, i.e., the total number of active updates processed in unit time. Updates caused by migration are not counted here since they are additional workload introduced by the design constraints of the system. When there is no eager update, the throughput is quite high since the update cost is the lowest as shown in Figure 4.14. With eager updates, although the throughput becomes much smaller, the system will not be paused for object migration. From Figure 4.15, we can observe that the percentage of single tree updates increases with larger $\mathcal{D}_e$. Since single tree updates are less costly, the total update throughput increases when $\mathcal{D}_e$ increases. When $\mathcal{D}_e$ is larger than 8, the throughput starts to drop. This is because when $\mathcal{D}_e$ is large enough, the percentage of single tree updates does not increase substantially, while the cost of eager updates does increase.

In summary, regarding the migration cost, the benefit of eager updating is already significant when $\mathcal{D}_e$ is as small as 0 or 1. With larger $\mathcal{D}_e$, the further improvement is minor. Considering the migration time and the update cost, we believe that setting $\mathcal{D}_e$ to 0 or 1 is sufficient enough for the tuning purpose.

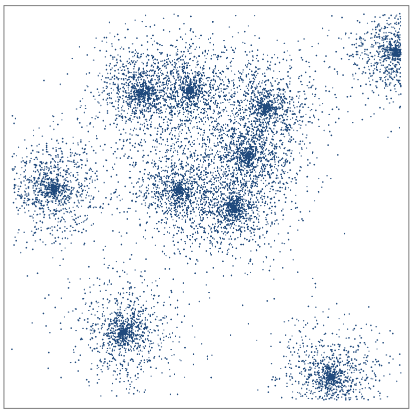


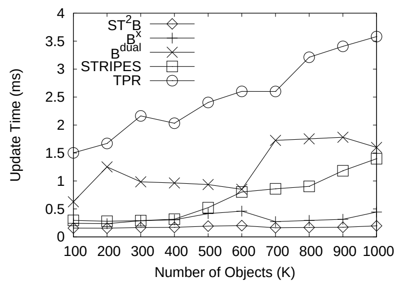
Figure 4.17: Distribution of default gaussian workload (10 hotspots)

4.8.4 Spatial Diversity

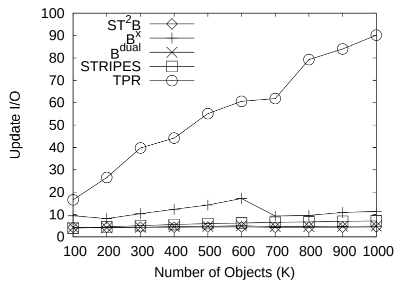
We now investigate the effectiveness of the ST^{2}B-tree with regard to the spatial diversity of moving objects. We use a Gaussian workload generated with 10 randomly selected hotspots as default. Figure 4.17 shows a sample of the workload used (some hotspots are close to others and cannot be clearly seen). In order to examine the effect of data skew only, we keep the distribution and cardinality of workload unchanged with time in this set of experiments. For the same purpose, eager updating is disabled and T is fixed to T_{up} so as to guarantee that there is no object migration. The indexes run up to $2T_{up}$ (240tu). The query workload consists of 100 queries. The queries are evaluated every 12 timestamps and the average costs are record.

4.8.4.1 Scalability Test

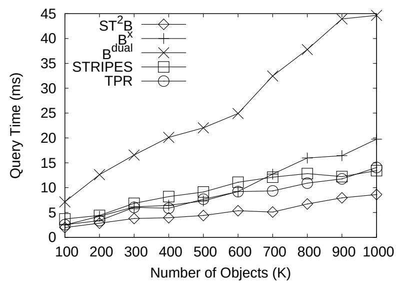
First, Figure 4.18 shows the effect of the data size on query performance. The number of objects varies from 100K to 1M, with an increment of 100K. The results of all the indexes (except the ST^{2}B-tree) conform to the findings we made in Chapter 3. Specifically, as shown in Figures 4.18(a)–4.18(b), the



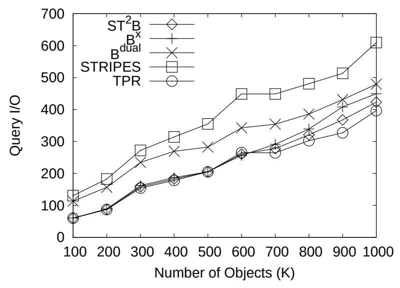
(a) Update time



(b) Update I/O



(c) Query time



(d) Update I/O

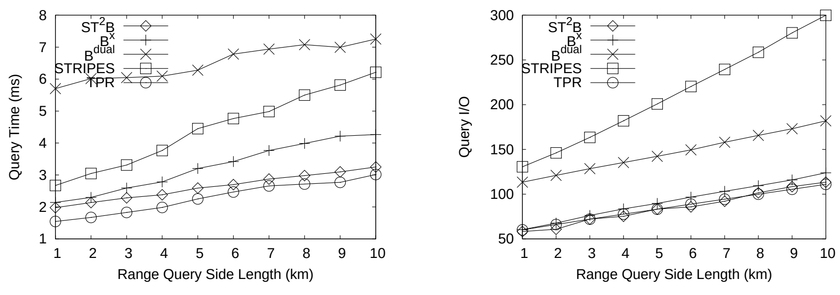
Figure 4.18: Effect of data size

TPR*-tree incurs a high update cost in both I/O and time. This is because of the overlap between MBRs that results in the TPR*-tree having to search multiple paths in an update. The MBR adjustment during an update operation further degrades the update time. The update time of the TPR*-tree is about 7 and 14 times higher than that of the ST^{2}B-tree with 100K and 1M objects respectively. The B^x-tree has fast update, but the number of update I/Os is higher than the others except the TPR*-tree. The number of update I/Os of the B^x-tree is affected by the granularity of space partitioning with regard to the data distribution. Since the workload is skewed, a uniform grid leads to some densely populated cells which break the balance of the B$^+$-tree by introducing many overflow pages. Consequently, the update I/O cost increases. Owing to the data-adaptive space partitioning, the ST^{2}B-tree has both fairly constant update time, i.e., about 0.2ms, and number of I/Os, which is around 4. STRIPES and the B^{dual}-tree, although having smaller number of update I/Os than the B^x-tree, take longer time to process a query.

Figures 4.18(c)–4.18(d) show the average query cost with respect to the number of objects. As expected, for all indexes, the query cost increases linearly with the data size. This is because more objects need to be retrieved in a given query region for a larger dataset. The ST^{2}B-tree, the B^x-tree and the TPR*-tree incur similar number of I/Os. The cost of the ST^{2}B-tree and the B^x-tree are mainly determined by the number of objects contained in the enlarged query region, while the TPR*-tree has been specifically designed to reduce its I/O cost over the original TPR-tree. The B^{dual}-tree incurs a larger number of I/Os. This is partially because the partitioning in the velocity dimensions makes some nearby objects of different velocities distributed into different leaf nodes, while in the ST^{2}B-tree and B^x-tree, nearby objects are clustered together. Among all the indexes, STRIPES is the most expensive regarding the I/O cost due to the unbalancing structure and low utilization space of the quad-tree.

Regarding the query processing time, the B^x-tree is not able to handle hotspots well due to its use of one single grid granularity, and causes many false positives to be retrieved and examined, and therefore incurs higher I/Os. The ST^{2}B-tree is most efficient in terms of query time due to its adaptive use of appropriate grid granularity when it rolls forward with time.

In Figure 4.18, the B^x-tree consistently outperforms the B^{dual}-tree. This is somewhat surprising, since it does not conform to the findings in [123], where the B^{dual}-tree is expected to beat the B$^+$-tree in terms of the number of I/Os. We find the following reasons for this inconsistency. In [123], the disk page is only 1K bytes and no buffer is used. Query processing of the B^x-tree incurs larger number of I/Os, since it has to revisit upper-



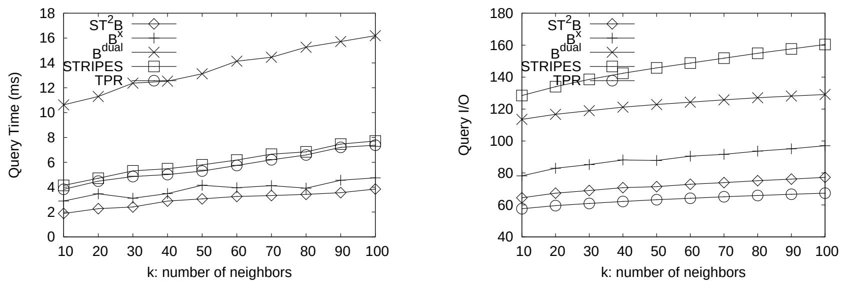
(a) Query time (b) Query I/O

Figure 4.19: Effect of range query size

level tree nodes several times. However, in our experiments, a small LRU buffer is used and intermediate nodes are kept in memory most of the time. The buffering effect brings a considerable decrease on the number of querying processing I/Os. As for the query processing time, [123] does not provide any result on the query processing time of the B^{dual}-tree. In our experiments, we find that the decomposition and overlapping testing of the MORs in the B^{dual}-tree are both time-consuming tasks. Therefore, the query processing time of the B^{dual}-tree is consistently higher than all the other indexes.

4.8.4.2 Size of Query

Figure 4.19 shows the average cost of processing range queries. All the indexes are tested with square-sized range queries with side length varying from 1km to 10km. As expected, the query cost of each index increases with an increasing query window size. Larger windows contain more objects and therefore lead to more node accesses. In general, Figure 4.19 shows the same relative order between the curves as in Figure 4.18. The TPR*-tree, the B^x-tree and the ST^{2}B-tree have the similar number of I/Os. With the default 100K Gaussian workload, the B^x-tree spends twice the query processing time of the TPR*-tree. As discussed in Section 2.3, the TPR*-tree is a data



(a) Query time (b) Query I/O

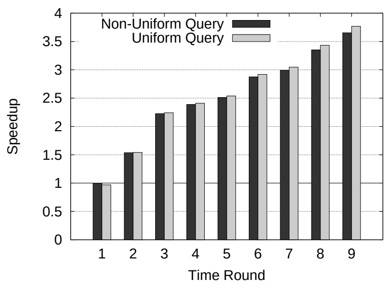
Figure 4.20: Effect of number of neighbors

partitioning index, which is less affected by the data distribution comparing with the B^x-tree. The ST^{2}B-tree shortens the processing time of the B^x-tree by about 30% for queries with 10km side length. The ST^{2}B-tree improves the B^x-tree with more adaptive space partitioning. As a result, the effect of data skew is mitigated. The B^{dual}-tree takes the longest processing time and incurs twice number of I/Os than the TPR*-tree, the B^x-tree and the ST^{2}B-tree. As for STRIPES, the processing time increases the fastest among all indexes because of the significant increase in the number of I/Os.

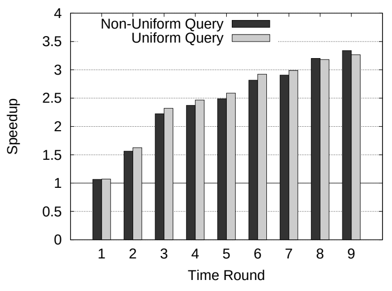
Figure 4.20 examines the performance of kNN queries further, with the number of neighbors k varying from 10 to 100. As for kNN queries, all indexes have a slight increase in I/O cost and query processing time. The I/O cost of the kNN queries, which depends on the number of objects in the expanded query region, is less sensitive to k for the B^x-tree and the ST^{2}B-tree. In terms of the number of I/Os, the ST^{2}B-tree surpasses the B^x-tree by a greater margin for kNN queries than for range queries. The kNN queries in both B^x-tree and the ST^{2}B-tree are conducted as incremental range queries with the initial search region estimated from the objects density. The cost depends largely on the accuracy of the estimated search radius. The B^x-tree makes such estimation using global object density. As a result, in dense regions, the B^x-tree starts the kNN search with an oversized search region; in sparse regions, the B^x-tree starts with a small search region, but has to expand the region for many times to find the kNNs. Both affect the query processing time and I/Os. The ST^{2}B-tree, on the other hand, starts the search with a more accurate radius according to the object density around the reference points. Owing to more accurate search region, the performance of the ST^{2}B-tree on kNN queries is less affected by the data skew. The TPR*-tree incurs fewer I/Os because of its branch-and-bound kNN search algorithm. The kNN query performance of STRIPES and the B^{dual}-tree are similar to their performance on range queries, contributing the highest number of I/Os and the highest processing time respectively.

4.8.5 Temporal Diversity

Next, we examine the effectiveness of the ST^{2}B-tree's self-tuning to adapt to the time-dependent changes in data cardinality. All the objects follow the same distribution used in the previous experiments (Figure 4.17). We build the indexes in the first round and run them for another 9 rounds of time. Each round is 120s. In each round, each object updates once and the whole index will be refreshed after the round. The number of queries is 1% of the number of updates each round. This is to simulate the real applications where many moving objects will keep on updating their positions, and the number of positional updates significantly outnumbers the number of queries. We study the total time of processing the updates and queries in each round as a measure of the overall performance. Then we compute the speedup introduced by the self-tuning feature of the ST^{2}B-tree,



(a) Range query



(b) kNN query

Figure 4.21: Benefit of index tuning with increasing data size

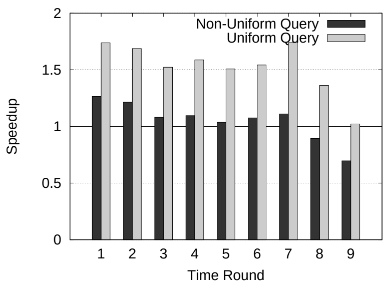
defined as:

$$\text{speedup} = \frac{\text{total processing time of the B}^x\text{-tree}}{\text{total processing time of the ST}^2\text{B-tree}}$$

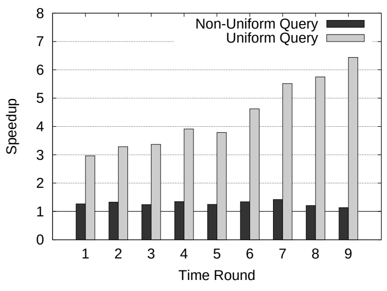
The granularity of the B^x-tree is selected using the initial number of objects, while the space partitioning of the ST^{2}B-tree is dynamically tuned in accordance to the workload. When the data is uniformly distributed, the performance of the ST^{2}B-tree degrades to that of the B^x-tree with only one reference point at the center of the space. In other words, the B^x-tree is a static version of the ST^{2}B-tree which completely ignores the distribution and changes of objects. Hence we compare the ST^{2}B-tree with the static B^x-tree to show the effectiveness of the self-tuning features. The running time of the self-tuning process is included in the total processing time of the ST^{2}B-tree.

First, we start with 100K objects and add another 100K each round. Figure 4.21 shows the speedup brought by self-tuning in each round of time. The speedup introduced by self-tuning grows with time, when the hotspots and data distribution change with time.

Initially, the B^x-tree selects the granularity of space partitioning with 100K objects. It then uses a grid with large cells (about $3000 \times 3000\text{m}^2$). With the increasing number of objects in the following rounds, the update



(a) Range query

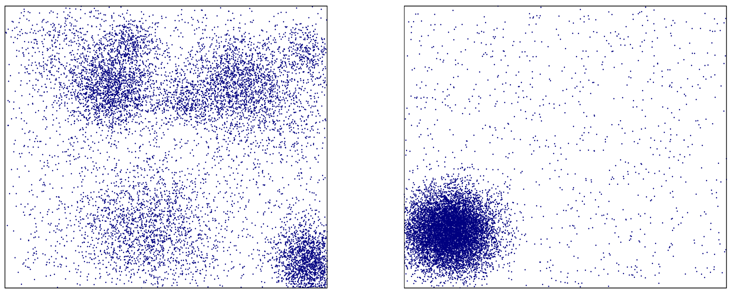


(b) kNN query

Figure 4.22: Benefit of index tuning with decreasing data cardinality

performance degrades, because the increase in the number of overflow pages affects the balance of the underlying B$^+$-tree. On the other hand, since the ST^{2}B-tree partitions and indexes objects according to the distribution and density, the update cost remains at about 0.2ms all the time. Since the B^x-tree uses a large cell, it saves on query processing time according to our findings in Section 4.8.2.1. However, with carefully chosen granularity of space partition, the query processing time of the ST^{2}B-tree is higher than the B^x-tree only in the dense regions. In those sparse regions, the ST^{2}B-tree might use even larger grid cells, which would reduce the query processing time. Therefore, combining all these facts, the overall speedup introduced by the self-tuning of the ST^{2}B-tree over static B^x-tree, especially when there are more updates than queries, are obvious and significant.

Figure 4.22 shows the results of a reverse process. Starting with 1M objects, the number of objects being indexed decreases by 100K per round. The B^x-tree now uses a fine grid with smaller cells (about 200×200m^2) to partition the entire space. As we can see, the speedup introduced by the self-tuning to the system is just a little higher with non-uniform queries. That is because the cost of the B^x-tree is also near optimal with such a fine grid. Non-uniform queries follow the same distribution as objects, and



(a) Round 5 (b) Round 9

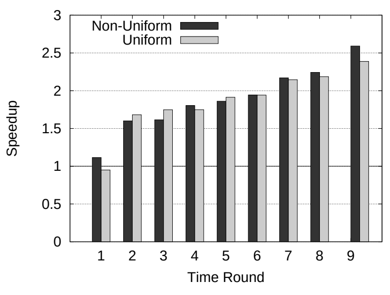
Figure 4.23: Distributions of workloads in spatio-temporal test

therefore queries are concentrated at those dense regions. Now, in those dense regions, the ST^{2}B-tree also employs fine grid. Therefore, the system speedup introduced by tuning is less significant. However, for the uniform queries, the ST^{2}B-tree gains more by tuning with the data workload. The ST^{2}B-tree reduces the processing time of queries in the sparse regions by using larger grid cells. The overall performance gain is much more significant than for non-uniform queries.

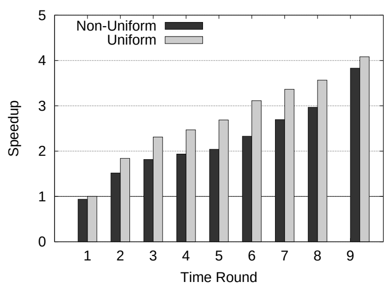
4.8.6 Spatio-Temporal Diversity

Now we further investigate the performance of the self-tuning phase of the ST^{2}B-tree with regard to the changes of objects distribution with time. We generate a set of workloads in which the skewness of objects increases with time. In round 0, we build the indexes with 1M uniformly distributed objects. Next, in round 1, the objects are generated with 10 hotspots. Subsequently, the number of hotspots is reduced by 1 each round. Finally, in round 9, there is only one hotspot. Figure 4.23 shows the snapshots of objects at round 5 (moderately skewed) and round 9 (highly skewed). The query-update ratio is still 1:100.

Figure 4.24 shows the changes of system speedup with time. As ex-



(a) Range query



(b) kNN query

Figure 4.24: Benefit of index tuning with changing data distributions

pected, the gain of the self-tuning ST^{2}B-tree over the static B^x-tree increases when the data become even more skewed with time. During round 1, the ST^{2}B-tree is comparable to the B^x-tree. Since the objects are uniformly distributed, the region-growing algorithm will result in only one reference point and hence the ST^{2}B-tree degenerates to a B^x-tree with only one reference point. However, with skewed objects joining in the subsequent rounds, the ST^{2}B-tree gradually outperforms the B^x-tree owing to the self-tuning phases, which are equipped with adaptive space partitioning and granularity of indexing. For range queries (Figure 4.24(a)), the ST^{2}B-treeoutperforms the B^x-tree by about 2 times in round 9 for both uniform and non-uniform query workloads. For kNN queries (Figure 4.24(b)), the performance gain is much higher, which is about 4 times.

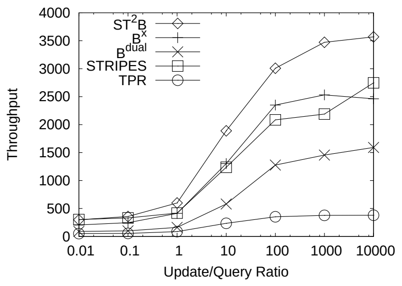
4.8.7 Throughput Test

Finally, we evaluate all indexes in a multiple-user environment. We use a multi-thread program to simulate the real multiple-user environment. All indexes adopt the concurrently control mechanism as included in the benchmark in Section 3.4, i.e., B-link [61] for the B$^+$-tree, R-link [75] for the R-tree, and the native 2-phase lock of the quad-tree for STRIPES.

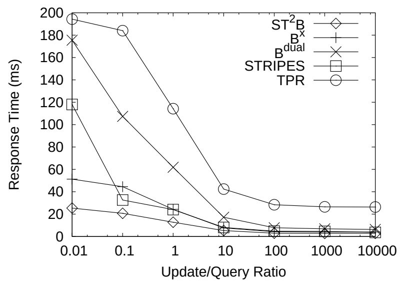
The default 1M dataset as shown in Figure 4.17 is used. The query workload consists of range queries with default settings, following the same distribution of the data objects. Updates and queries arrive evenly in time and are loaded into a task pool and then randomly distributed to each free working thread. The performance is measured by two metrics: throughput and response time. The throughput is defined as the average number of tasks finished in a unit time (1tu). The results are the average of 10 runs of simulation.

Figure 4.25 shows the throughput and response time with the query-update ratio varying from 100:1 to 1:1000 using 10 working threads. In real moving-object applications, the update load caused by the changes in object locations and moving speed is much higher than the query load, and the query-update ratio is to simulate such scenario. As expected, the throughput of the indexes increases significantly with more updates and the response time decreases. The queries, which hold shared lock on the node being accessed, do not prevent the other queries. However, although queries allow other read operations, they block the update operations, and by design of the experiment, the updates contribute more to the throughput. The updates access only a few nodes in the index and can finish very quickly. The (range) queries, on the other hand, have to traverse multiple paths and read many leaf nodes (data nodes); hence they take longer than the updates. Since the throughput is defined as the number of operations completed by the indexes every second, the updates contribute more to it. Therefore, when the percentage of updates in the workload increases, the throughput increases and the response time decreases accordingly.

Figure 4.26 shows the effect of the number of threads under workload whose query-update ratio is 1:100. The number of threads varies from 1 to

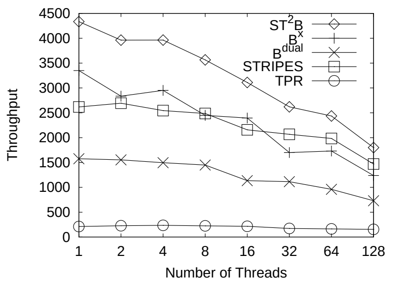


(a) Throughput

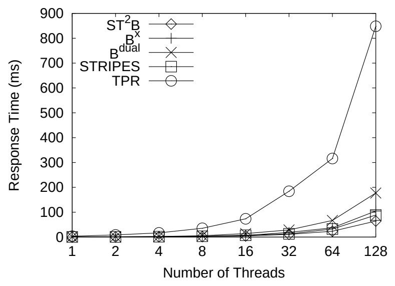


(b) Response time

Figure 4.25: Effect of update/query ratio

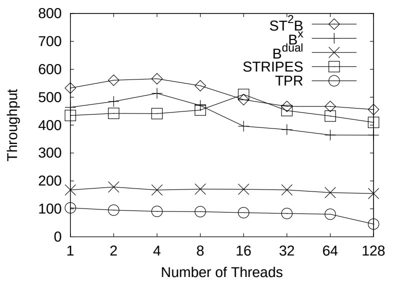


(a) Throughput

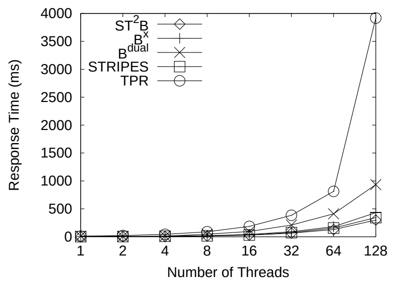


(b) Response time

Figure 4.26: Effect of number of threads (update/query=100:1)



(a) Throughput



(b) Response time

Figure 4.27: Effect of number of threads (update/query=1:1)

128. In general, the throughput reduces with increasing number of threads for all indexes. The response time increases with the number of threads being used. An update locks exclusively the node being accessed and all the concurrent requests for reading/writing the node are suspended. As the workload includes more updates than queries, the indexes are frequently being write-locked. The throughput decreases with more threads and each thread waits for a longer time for its turn to access the tree.

However, with more queries, the throughput first increases and then reduces with increasing number of threads. As shown in Figure 4.27, when the workload consists of 50% queries and 50% updates, the throughput reaches the peak with about 2 threads or 4 threads for both indexes. As more threads are introduced, they start to compete for resources and the throughput reduces as a result. Because the queries hold a shared lock on the node being accessed, it will not suspend the other query operations. Therefore, the degree of concurrency becomes higher with more queries. With more queries, the throughput reaches the peak with more threads. For example, when query-update ratio is 10:1, the peak of the throughput is 4 threads or so. However, in moving-object databases, there are typically more short updates than queries, so we omit the results for such workload composition.

As can be observed from Figure 4.26 and Figure 4.27, the indexes hit thrashing point after the number of threads increases to a certain point and this is when the throughput starts to decrease after hitting the peak. We note that the throughput and the response time can be improved by implementing some admission controls to throttle the amount of work being performed concurrently. However, the admission control introduces another dimension of effect to the performance, which has not been taken into account here.

4.9 Summary

In this chapter, we studied the problem of index tuning in moving-object databases. We identified several forms of data diversity in moving-object applications that existing indexes are unable to handle effectively. We then proposed the ST^{2}B-tree index that can automatically adjust itself to avoid performance degradation caused by the data diversity. To adapt to space diversity, the ST^{2}B-tree partitions the data space using a set of reference points. Each reference point uses its own individual grid to partition its Voronoi cell. The grid granularity is determined by object density around a reference point. By monitoring the distribution and density of objects continuously, the ST^{2}B-tree dynamically determines a different set of reference points, and adaptively adjusts the granularity of space partitioning. We also proposed methods to choose the reference points, and introduced a guideline on the optimal choice of granularity. To deal with the time diversity, the ST^{2}B-tree employs a "multi-tree" approach, where two sub-trees are used to index objects regarding their last update time. The basic idea is to rebuild the sub-trees periodically and alternately, during which the newly identified reference points and granularity are used. To guarantee the correctness, the sub-tree to be rebuilt must be empty. If not, the remaining objects must be migrated to the other sub-tree manually. To overcome this constraint, we proposed a novel eager update mechanism that can reduce the migration load while incurring little or even no additional cost. An extensive experimental study against several state-of-the-art indexes showed the superiority of the ST^{2}B-tree, confirming that the ST^{2}B-tree is efficient, robust and scalable with respect to data distribution, volume and concurrent operations. More importantly, equipped with the self-tuning capability, the ST^{2}B-tree is also adaptive to changes in workload.

Chapter 5

An Adaptive Updating Protocol for Moving Object Databases

For decades, research on databases has been focused on the design of indexing techniques and query processing algorithms to improve the performance of the database system. In moving-object databases, while a variety of indexing techniques have been proposed to accelerate the processing of workload in the database, i.e., updates and queries, no much attention has been paid to the possibility of reducing the workload in the first place. In this chapter, we explore this possibility and introduce a generic and adaptive updating protocol. Compared with existing updating mechanisms, the proposed protocol provides larger space of tolerance for object-tracking, freeing objects from frequent updates. The experimental results confirm that the proposed protocol significantly reduces the number of updates, thereby reducing the overall workload of the system.

5.1 Introduction

While most of the existing studies on moving-object databases focus on enhancing indexing and query processing efficiency, less effort has been made to address the issue of performance optimization from a more fundamental perspective. Although the numbers of objects and queries are not controlled by the system*, the actual workload of the system can vary greatly with different system designs. In particular, the updating workload of the system, i.e., the number of updates, is decided by the representation of objects and the updating protocol. For example, updates can be less frequent if linear motion model is used to estimate objects' movement, compared to the case that objects are stored as spatial points (i.e., last known locations) in the system.

Considering the limited room of technical advances on indexing structures and query processing algorithms, it is even important now to reconsider the design of the updating protocol. In this chapter, we make the first attempt at a systematic investigation on this possibility. In particular, we propose a new adaptive updating protocol, called *STSR-based Updating Protocol*, which reduces the object updating frequency by maintaining only relaxed motions instead of exact ones in the database. The design of the protocol is on the basis of a careful analysis on the advantages and disadvantages of the existing protocols. Compared with existing updating protocols introduced in Section 2.2, the proposed STSR-based updating protocol is equipped with three major features to enhance the performance, as follows.

First, instead of the traditional object representations, the proposed protocol keeps an *S*patial-*T*emporal *S*afe *R*egion (STSR) for each object. The

*Assume that the system wants to serve as many objects and queries as possible. The numbers of objects and queries are application-dependent, which are not controlled by the database system.

STSR allows approximation on object motion in both spatial and velocity spaces, leading to higher tolerance of tracking accuracy and hence a decrease in the number of updates in the database. On the other hand, to answer queries accurately, the system is required to probe the latest status of objects when their STSRs are inadequate in returning the exact query results.

Second, the linear motion model is adopted instead of other complex high-order models. While achieving benefits on computational cost and index efficiency with the simple motion model, our empirical studies show that there is no significant difference on prediction quality even when more complicated motion model is employed instead. In addition, since the linear model is adopted by most of existing indexes, it enables the protocol to work seamlessly with existing indexing structures.

Third, the STSR-based updating protocol takes both historical updating records and query distribution into consideration, rendering a cost optimization model and an automatic tuning mechanism highly adaptive to the changing world. This enables our STSR-based updating protocol to outperform existing solutions in maximizing the savings on the number of updates.

Finally, we summarize the contributions of this chapter as follows:

- We present a new and adaptive updating protocol for reducing the updating workload while guaranteeing the efficiency and accuracy of predictive queries.

- We propose a cost model to estimate the update workload incurred by specific moving object(s), based on the historical records of updates and recent queries in the vicinity.

- We discuss cost-based optimization strategies to reduce the updating workload, which tries to construct optimized spatial-temporal safe regions to minimize the expected updating workload in the future.
- We extend our protocol by allowing the system to output approximate but accuracy bounded query results, to cut unnecessary updates further and reduce the impact of fast moving vehicles.
- We extensively evaluate the performance of our proposal with a variety of index structures on different real and synthetic datasets.

The remainder of the chapter is organized as follows. Section 5.2 describes some preliminary concepts required in this chapter. Section 5.3 introduces the STSR-based updating protocol and the basic algorithms for update and query processing. Section 5.4 presents an algorithm for finding the optimized STSR and some related optimization techniques. Section 5.5 discusses the integration problem of the new protocol. Section 5.6 introduces approximate query processing with the STSR-based updating protocol. Section 5.7 reports the experimental study and Section 5.8 finally summarizes the chapter.

5.2 Preliminaries

Assume there are n moving objects, $O = \{o_1, o_2, \ldots, o_n\}$, being monitored in the system. Following the common assumption in moving-object databases, the temporal dimension is represented as discrete time instances $T = \{0, 1, \ldots, t, \ldots\}$. The exact location of object o_i at t is denoted by a vector $l_i^t = \langle l_i^t.x, l_i^t.y\rangle$. Similarly, the velocity of o_i at t is denoted by $v_i^t = \langle v_i^t.x, v_i^t.y)\rangle$. With the linear motion model, the *Predicted Location* of

o_i at time s $(s \geq t)$, denoted by pl_i^s, can be derived from l_i^t and v_i^t, where

$$\begin{cases} pl_i^s.x = l_i^t.x + v_i^t.x \cdot (s-t) \\ pl_i^s.y = l_i^t.y + v_i^t.y \cdot (s-t) \end{cases} \tag{5.1}$$

5.2.1 Spatio-Temporal Safe Region

Before delving into the detail of our updating protocol, we first introduce the concept of *Spatio-Temporal Safe Region*, as defined below.

Definition 5.1. Spatio-Temporal Safe Region (STSR).
Given a moving object o_i, a *S*patio-*T*emporal *S*afe *R*egion (STSR) of o_i is represented by a tuple $R(o_i) = \langle LR, VR, t_r, t_e \rangle$, where LR is a rectangle in the physical space (i.e., the space where the object moves), VR is a rectangle in the velocity space, t_r is the reference time, and t_e is the expiry time $(t_e > t_r)$.

Given an STSR $R(o_i)$, the location rectangle LR is bounded by $\left[LR.x^{\vdash}, LR.x^{\dashv}\right]$ and $\left[LR.y^{\vdash}, LR.y^{\dashv}\right]$ on the two spatial dimensions respectively. Similarly, the velocity rectangle VR is bounded by $\left[VR.x^{\vdash}, VR.x^{\dashv}\right] \times \left[VR.y^{\vdash}, VR.y^{\dashv}\right]$. Intuitively, LR relaxes the location of o_i at reference time t_r, and VR encloses the possible velocities of o_i before t_e. Given the STSR $R(o_i)$, a *Predicted Region* is a region that encloses all possible locations of o_i at time t before the expiry time t_e, as defined below.

Definition 5.2. Predicted Region.
Given an STSR $R(o_i) = \langle LR, VR, t_r, t_e \rangle$, the predicted region of o_i at time t $(t_r \leq t \leq t_e)$, $P_i^t = \left[P.x^{\vdash}, P.x^{\dashv}\right] \times \left[P.y^{\vdash}, P.y^{\dashv}\right]$, is the maximal spatial

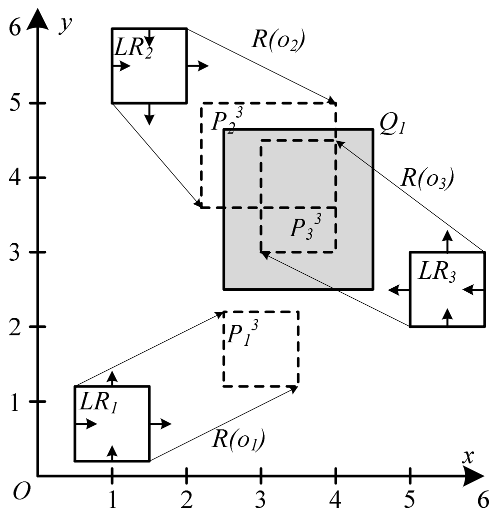


Figure 5.1: Examples of STSR

rectangle expanded from LR with respect to VR, where

$$\begin{cases} P.x^{\vdash} = LR.x^{\vdash} + VR.x^{\vdash}(t - t_r) \\ P.x^{\dashv} = LR.x^{\dashv} + VR.x^{\dashv}(t - t_r) \\ P.y^{\vdash} = LR.y^{\vdash} + VR.y^{\vdash}(t - t_r) \\ P.y^{\dashv} = LR.y^{\dashv} + VR.y^{\dashv}(t - t_r) \end{cases} \tag{5.2}$$

The definition above assumes that the predictive time t is no earlier than the reference time t_r, which can be easily relaxed. When $t < t_r$, the predicted region is calculated with the "reverse" velocity bounding rectangle, which is $\left[VR.x^{\dashv}, VR.x^{\vdash}\right] \times \left[VR.y^{\dashv}, VR.y^{\vdash}\right]$, and the predicted region is

$$\begin{cases} P.x^{\vdash} = LR.x^{\vdash} + VR.x^{\dashv}(t - t_r) \\ P.x^{\dashv} = LR.x^{\dashv} + VR.x^{\vdash}(t - t_r) \\ P.y^{\vdash} = LR.y^{\vdash} + VR.y^{\dashv}(t - t_r) \\ P.y^{\dashv} = LR.y^{\dashv} + VR.y^{\vdash}(t - t_r) \end{cases} \tag{5.3}$$

As an example, Figure 5.1 shows the STSRs of three objects $\{o_1, o_2, o_3\}$, and Table 5.1 specifies the details of these STSRs. Based on Definitions 5.1–

Table 5.1: Details on the STSRs in Figure 5.1

STSR	**LR**	**VR**	$\mathbf{t_r}$	$\mathbf{t_e}$
$R(o_1)$	$[0.5, 1.5] \times [0.2, 1.2]$	$[1, 1] \times [0.5, 0.5]$	1	4
$R(o_2)$	$[1, 2] \times [5, 6]$	$[1.2, 2] \times [-1.4, -1]$	2	5
$R(o_3)$	$[5, 6] \times [2, 3]$	$[-1, -1] \times [0.5, 0.75]$	1	5

5.2, we can get the predicted regions of the objects at timestamp $t = 3$, shown as dashed rectangles (P_1^3, P_2^3, P_3^3) in Figure 5.1.

5.2.2 Consistency Verification

An STSR $R(o_i)$ is valid at time t if $R(o_i)$ can capture all predicted locations at all subsequent timestamps before t_e. Specifically, $R(o_i)$ is said to be consistent with o_i at t if both of the following two conditions hold: 1) the exact location l_i^t of o_i remains in the predicted region P_i^t inferred from $R(o_i)$; and 2) the predicted location pl_i^s remains in the predicted region P_i^s for any s $(t < s \leq t_e)$. Note that the consistency depends on the locations only; $R(o_i)$ remains valid even if the velocity of o_i at time t is out of the velocity rectangle VR.

According to Definitions 5.1–5.2, it is straightforward for an object to verify the consistency between its current movement and the STSR, by simply checking the predicted locations of the object at every timestamp before the expiry time. Since the linear model is used to derive the predicted regions, the verification process can be simplified by checking predicted locations at only two or three timestamps. In particular, the object first tests whether the STSR is expired, i.e., $t > t_e$. If so, the algorithm returns a negative answer immediately. Otherwise, the object continues to check whether the exact location l_i^t and the predicted location $pl_i^{t_e}$ at the expiry time are covered by predicted regions P_i^t and $P_i^{t_e}$ respectively. Besides, if

Algorithm 5.1: Consistency Verification

```
Input  : Current time $t$, exact location $l_i^t$ and velocity $v_i^t$ at $t$,
         current STSR $R(o_i) = \langle LR, VR, t_r, t_e \rangle$
Output: true, if $R(o_i)$ is consistent with $l_i^t$ and $v_i^t$; otherwise false
1  if $t > t_e$ then
2      return false;
3  Compute the predicted region $P_i^t$ w.r.t $R(o_i)$;
4  if $l_i^t$ is out of $P_i^t$ then
5      return false;
6  Compute the predicted region $P_i^{t_e}$ w.r.t. $R(o_i)$;
7  Compute the predicted location $pl_i^{t_e}$ w.r.t. $l_i^t$ and $v_i^t$;
8  if $pl_i^{t_e} \notin P_i^{t_e}$ then
9      return false;
10 if $t < t_r$ then
11     Compute the predicted location $pl_i^{t_r}$ w.r.t. $l_i^t$ and $v_i^t$;
12     if $pl_i^{t_r} \notin LR$ then
13         return false;
14 return true;
```

the reference time t_r is after current time t, it is also necessary to check whether the predicted location $pl_i^{t_r}$ is covered by the location rectangle LR. The complete verification algorithm is shown in Algorithm 5.1.

We then give examples to show why checking predicted regions at these two or three timestamps is sufficient to prove the consistency of an STSR. Figure 5.2 shows two STSRs $R(o_1)$ and $R(o_2)$, where the solid points denote the exact locations at current time t and the hollow points represent the predicted locations inferred from the exact locations and velocities at t based on Equation 5.1. As for o_1, the reference time is early than t (i.e., $t_r < t < t_e$). Since the exact location at t is inside the location rectangle (i.e., $l_1^t \in LR_1$) and the predicted location at t_e is covered by the predicted region (i.e., $pl_1^{t_e} \in P_1^{t_e}$), $R(o_1)$ is consistent with all predictions from t to t_e based on o_1's exact location and velocity at t. On the other hand, the reference time of $R(o_2)$ is after t (i.e., $t < t_r < t_e$). The predicted regions

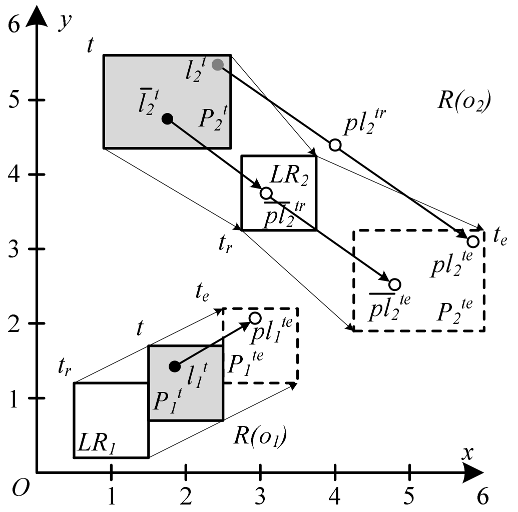


Figure 5.2: Examples of checking the consistency of STSR

between t and t_r are thus extended backwards according to Equation 5.3. If we test only the locations at the current timestamp and expiry time, some false-positive STSR may pass the verification wrongly. As we can see in Figure 5.2, although the predicted regions can bound the predicted locations at t and t_e respectively (i.e., $l_2^t \in P_2^t$ and $pl_2^{t_e} \in P_2^{t_e}$), the predicted region $P_2^{t_r}$ fails to cover $pl_2^{t_r}$. In this case, $R(o_2)$ is inconsistent with o_2's prediction at t_r. To prune such false positives, the algorithm continues to verify the predicted location at the reference time. The STSR is valid if it passes all three verifications, as the predictions at $\overline{l}_2^t$, $\overline{pl}_2^{t_r}$ and $\overline{pl}_2^{t_e}$ imply. Otherwise, the algorithm returns a negative answer, as for l_2^t, $pl_2^{t_r}$ and $pl_2^{t_e}$ in the example. Since linear interpolation is employed, if the predicted regions at the two ending timestamps cover the corresponding predicted locations, all predicated regions at intermediate timestamps do as well.

5.2.3 Predictive Queries

The STSR-based updating protocol introduced in this chapter is specially optimized for predictive range queries in moving-object databases. Given a querying rectangle QR in location space and the querying time t_q, the predictive range query finds all the objects with predicted locations in QR at time t_q.

Definition 5.3. Predictive Range Query.
Given the moving-object database O at current time t, a predictive range query, in terms of querying region RQ and querying time t_q ($t_q \geq t$), finds all objects in O whose predicted locations are inside RQ at t_q, i.e., $\{o_i \in O \mid pl_i^{t_q} \in RQ\}$.

Although we constrain our discussion in range queries throughout the chapter, it is easy to see that other queries, such as the kNN query, can be answered with a series of range queries[†]. If the database system always returns the complete result to any query, the system is said to commit to *Exact Query Processing*. When the system performance is more important than the completeness of the query result, it is possible to reduce the workload by employing *Approximate Query Processing* instead. In the following, we give a formal definition on approximate query processing.

Definition 5.4. α-Approximate Query Processing.
Given the moving-object database O at current time t, a predictive range query, in terms of querying range RQ and querying time t_q ($t_q \geq t$), finds a set R of objects whose predicted location in RQ at time t_q, i.e., $\{o_i \in R \mid pl_i^{t_q} \in RQ, o_i \in O\}$, and $|R| \geq \alpha \cdot |\{o_i \in O \mid pl_i^{t_q} \in RQ\}|, 0 < \alpha \leq 1$.

Approximate query processing is found to be sufficient in many real applications of moving-object databases. For example, in a traffic monitoring system, the users are only interested in analyzing the traffic volume at specific time. A rough number of vehicles, instead of the exact vehicles, are

[†]See kNN query processing of the B^x-tree in [50] and the ST^{2}B-tree in Chapter 4.

more than enough for the users to understand the traffic conditions of the regions of interest. In the following sections, we first focus on the design of the STSR-based updating protocol for exact query processing. In Section 5.6, we extend the protocol to support α-approximate query processing.

5.3 STSR-Based Updating Protocol

Generally speaking, the STSR-based updating protocol consists of two types of updates: the active update and the passive update. In the following, we discuss these two updates in detail.

5.3.1 Active Update

With the STSR-based updating protocol, each object o_i is always associated with one (and only one) STSR $R(o_i)$, which is stored in the database as well as the memory of the client device. At each instance of time, the object checks whether its STSR $R(o_i)$ remains consistent with its latest motion prediction with its current location and velocity, using Algorithm 5.1. If inconsistency exists, the object issues an active update to the database including its current location and velocity.

At the server side, the database system continuously listens to any incoming active updates from the objects. If an object updates its location and velocity at time t, the system updates the STSR of the object stored in the database accordingly. Specifically, the system first calculates a new STSR for o_i based on the updated record. The new STSR is sent to the object o_i, while the old STSR of o_i in the database is replaced by the new one. An outline of the update procedure can be found in Algorithm 5.2.

Algorithm 5.2: Update with STSR

```
Input : Object $o_i$, current location $l_i^t$, current velocity $v_i^t$, current
        time $t$
1 Calculate a new STSR $R(o_i) = \langle LR, VR, t_r, t_e \rangle$ according to $l_i^t$ and $v_i^t$;
2 Send $R(o_i)$ to $o_i$;
3 Replace the STSR of $o_i$ in the database with $R(o_i)$;
```

5.3.2 Query Processing and Passive Update

While active updates are initiated by the objects themselves due to significant motion changes, passive updates are issued when the database processes queries. Typically, range queries in most moving-object databases are processed using a filter-and-refine approach, which determines candidate objects based on their predicted regions and verifies them by probing passive updates if necessary. A candidate set is constructed first by retrieving all objects whose predicted regions overlap with the query region QR at query time t_q. For each candidate object o_i, if the predicted region of o_i is completely covered by the query range, the object can be safely marked as a positive answer to the query. Otherwise, a request is sent to the object for an update on its current location and velocity, which will be used by the server to make a more accurate prediction. Subsequently, the object is listed in the query answer if the new predicted location is still inside the query region. Algorithm 5.3 presents the general working flow for answering range queries based on the concept of STSRs.

Let us recall the example shown in Figure 5.1, and see how predictive range queries are answered with STSRs. If a range query is issued in the rectangle region $QR = [2.5, 4.5] \times [2.5, 4.5]$ at querying time $t_q = 3$, the predicted regions can be calculated according to the inference equations (Equations 5.2–5.3). Take object o_3 as an example. The predicted region at $t = 3$, P_3^3, is the rectangle $[3, 4] \times [3, 4.5]$. Since P_3^3 is covered by the query

Algorithm 5.3: Range Query with STSR

```
Input  : Query region $QR$, query predictive time $t_q$
Output: Query result $R$
1  Find the object set $O' \subseteq O$ that the predicted region $P_i^{t_q}$ overlaps
   with $QR$ for any $o_i \in O'$;
2  foreach $o_i \in O'$ do
3      if $P_i^{t_q}$ is covered by $QR$ then
4          Include $o_i$ in the query result $R$;
5      else
6          Send a probe request to $o_i$ for current location and velocity;
           // When the update arrives at the system, Algorithm 5.2 is
              triggerred.
7          Compute the new $pl_i^{t_q}$ with current location and velocity of $o_i$;
8          if $pl_i^{t_q} \in RQ$ then
9              Include $o_i$ in the query result $R$;
10 return the (complete) query result $R$;
```

region completely, o_3 is a positive answer given that o_3's STSR is consistent with its current motion prediction. On the other hand, there is no overlap between P_1^3 and QR, implying that o_1 is a negative result. Unlike o_1 or o_3, the case of o_2 is more complicated, since the predicted region P_2^3 partially overlaps with QR. To clarify if o_2 is in the query result or not, the system needs to probe a passive update from o_2 for its current motion parameters, i.e., the location and velocity. On receiving the update, the query result and the STSR of o_2 are updated accordingly.

5.4 Optimization Techniques

To put the STSR-based updating protocol in use, there are two issues to be resolved. First, to minimize the updating workload of the system, it is important to find the optimal STSR given an object update (Step 1 in Algorithm 5.2), as the size of STSR has opposite effects on the numbers of active and passive updates. Second, considering the variety of indexing structures in moving-object databases, objects are stored and retrieved in different ways, even if the same linear motion model is adopted. Therefore, it remains unclear so far how the STSR-based updating protocol can be integrated into existing indexes. This is important for efficient retrieval of candidate objects in the filter step of the query processing (Step 1 in Algorithm 5.3).

In this section, we focus on the first issue. First, Section 5.4.1 provides a detailed study on the effect of the size of STSR. Then, in Section 5.4.2, we present an algorithm of finding the optimized STSR. Section 5.4.3 suggests some optimization techniques to enhance the performance of the system further. The integration problem will be discussed in the next section.

5.4.1 Cost Model

In this sub-section, we present a cost model estimating the probable validity of a given STSR. As introduced in Section 5.3, there are two types of updates, namely *Active Update* and *Passive Update*. Either active or passive update leads to a new STSR. We use $P_a(R(o_i))$ and $P_p(R(o_i))$ denote the probabilities of active and passive update happening on $R(o_i)$ before the expiry time t_e. An STSR remains valid until the expiry time t_e with probability:

$$P_{valid}(R(o_i)) = (1 - P_a(R(o_i))) \cdot (1 - P_p(R(o_i))) \tag{5.4}$$

An optimal STSR should maximize P_{valid}. Intuitively, a larger LR and VR lead to a lower $P_a(R(o_i))$ but a higher $P_p(R(o_i))$, and vice versa. To maximize Equation 5.4, it is necessary to estimate both probabilities first.

5.4.1.1 Active Update Probability

An active update is issued by object o_i if the previous STSR is no longer consistent with the current motion prediction. Given an STSR $R(o_i)$, $P_a(R(o_i))$ is the probability of inconsistency happening before the expiry time t_e of $R(o_i)$. Without the exact future trajectory of o_i, it is hard to estimate $P_a(R(o_i))$. If we assume the previous motion model does not change, $R(o_i)$ will always be valid until t_e. On the other hand, it is hard to indicate possible changes in the motion without any additional knowledge on the exact future trajectory. However, if all the similar historical trajectories are recorded in the database, we can make an estimation on the active update probability based on the statistical information. Unfortunately, this solution is impractical due to the high cost in both storage and processing on the trajectories. To facilitate effective and efficient statistical estimation, the database system maintains a set of recent STSR update records, as defined below, to sample the historical trajectories of objects.

Definition 5.5. STSR Update Record.
An STSR update record is a tuple $\langle R(o_k), o_k, l_k^{t_u}, v_k^{t_u}, t_u \rangle$, where k is the identity of the associated moving object, t_u is the time when the update record is generated, $l_k^{t_u}$ and $v_k^{t_u}$ are the location and velocity of o_k at t_u, and $R(o_k) = \langle LR, VR, t_r, t_e \rangle$ is the latest STSR of o_k before the update time t_u.

The STSR update records are maintained in a separate table in the database, called the *Update Record Table*. A record is inserted into the table when: 1) an update from o_k is received due to the violation on $R(o_k)$, or 2) the previous STSR $R(o_k)$ expires. In the first case, the location and velocity at update time are written into the record. In the second case, NULL values are inserted instead. This implies that each STSR issued in

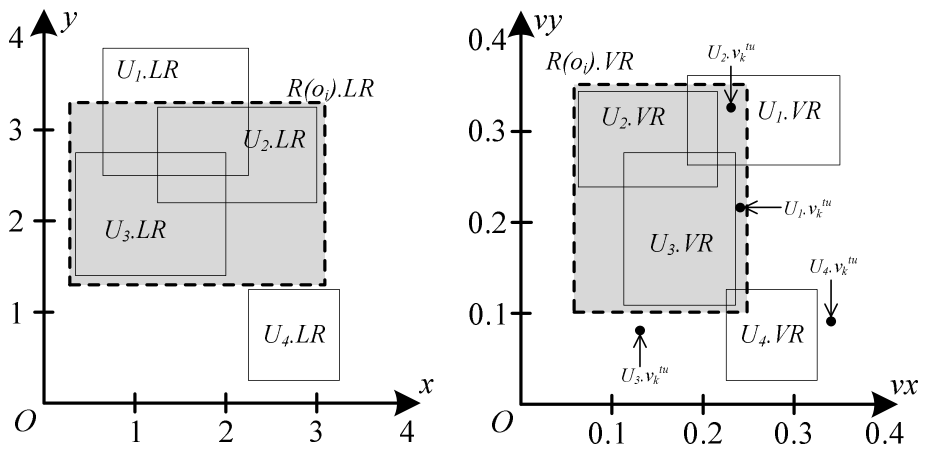


Figure 5.3: Coverage of STSR on update records

the past has a record kept by the database system in the update history table.

Next, we introduce the concept of *Record Coverage* to evaluate the robustness of a new STSR with respect to similar STSR update records in the update record table. Specifically, an STSR $R(o_i)$ covers an STSR update record $U = \langle R(o_k), o_k, l_k^{t_u}, v_k^{t_u}, t_u \rangle$, if 1) $R(o_i).LR$ covers $R(o_k).LR$, and 2) $R(o_i).VR$ covers both $R(o_k).VR$ and $v_k^{t_u}$. Without ambiguity, we use $R(o_i) \supseteq U$ denote the coverage relationship, in which U is a specific update record. In Figure 5.3, we present an example on the coverage relationship with an STSR $R(o_i)$ and four update records $\{U_1, U_2, U_3, U_4\}$. The location rectangle LR of the update records in the physical space is shown with solid thin borders on the left. Similarly, the velocity rectangle VR and the velocity at update time $v_k^{t_u}$ are plotted in the velocity space on the right. Given $R(o_i).LR$ and $R(o_i).VR$ as gray rectangles with dashed thick borders in both spaces, $R(o_i)$ covers U_2 by the definition above. U_3 is not covered by $R(o_i)$ since the updated velocity $v_k^{t_u}$ of U_3 is out of the velocity rectangle $U_3.VR$.

When the STSR $R(o_i)$ covers an update record U, it remains consistent until the expiry time if o_i follows the same trajectory of o_k when U was recorded for o_k. This motivates the following definition of *Coverage Rate* to approximate the probability of an active update on the new STSR $R(o_i)$.

Definition 5.6. Coverage Rate.

Given an STSR $R(o_i)$ and a group of similar STSR update records U_{NN}, the coverage rate of $R(o_i)$ is measured by $\frac{|\{U_k \in U_{NN} \mid R(o) \supseteq U_k\}|}{|U_{NN}|}$

We now discuss the similar update record set U_{NN} in the above definition. To get all update records related to the object o_i, the system retrieves all update records $U_k \in U_{NN}$, if the location rectangle $U_k.LR$ and velocity rectangle $U_k.VR$ of the record U_k cover the location $l_i^{t_r}$ and velocity $v_i^{t_r}$ of o_i at reference time t_r respectively. As a summary, we get the active update probability from the historical perspective as

$$P_a(R(o_i)) = 1 - \frac{|\{U_k \in U_{NN} \mid R(o_i) \supseteq U_k\}|}{|U_{NN}|} \tag{5.5}$$

5.4.1.2 Passive Update Probability

A passive update is issued when it is not sufficient to decide whether an object meets the query with its STSR stored in the database, i.e., the predicted region partially overlaps with the query region. To estimate the number of passive updates for a given STSR, it is necessary to predict the probability of partial overlap. To simplify the model and save computational cost, we relax the probability by including any overlap even if the predicted region is completely covered by the query region. This relaxation does not affect estimation error much since the query range is usually not large enough to cover many predicted regions.

Following the existing assumptions on the performance analysis of range queries [87, 123], we assume the querying location and querying time follow the uniform distribution in spatial space and temporal space. The probability of a predicted region overlapping any range query at time t is thus proportional to the volume of the predicted region.

For an STSR $R(o_i)$ issued at update time t_u, the total volume of all predicted regions at timestamps between t_u and t_e is denoted by $Vol(R(o_i))$. By using the following notations to represent the side lengths of the location rectangle and the velocity rectangle, i.e.,

$$\begin{cases} LD_x = LR.x^{\dashv} - LR.x^{\vdash} \\ LD_y = LR.y^{\dashv} - LR.y^{\vdash} \\ VD_x = VR.x^{\dashv} - VR.x^{\vdash} \\ VD_y = VR.y^{\dashv} - VR.y^{\vdash} \end{cases} \tag{5.6}$$

The total volume can be further simplified as follows.

$$Vol(R(o_i)) = \sum_{t=t_u}^{t_e} (LD_x + VD_x(t - t_r)) \cdot (LD_y + VD_y(t - t_r)) \tag{5.7}$$

If the expected number of queries happening at each timestamp is N and the volume of the whole spatial space is S, the probability of not meeting any range query is approximated by the ratio of total volume with respect to the expected query volume, i.e., $1 - P_p(R(o_i)) = \max\{1 - \frac{Vol(R(o_i)) \cdot N}{(t-t_r) \cdot S}, 0\}$. Finally, we get the probability of a passive update as shown below.

$$P_p(R(o_i)) = \min\left\{\frac{Vol(R(o_i)) \cdot N}{(t - t_r) \cdot S}, 1\right\} \tag{5.8}$$

5.4.2 Calculation of the Optimal STSR

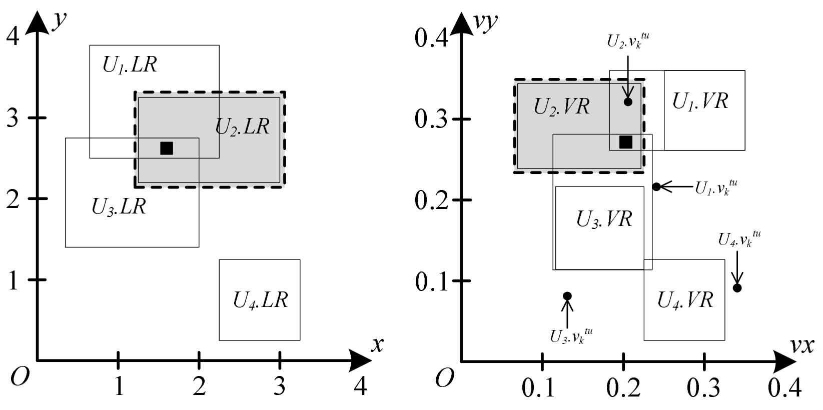
Based on the cost model presented above, we present an algorithm to find the optimal STSR $R(o_i)$ that minimizes the expected update probability. The estimation on the active update probability $P_a(R(o_i))$ depends on the number of update records covered by $R(o_i)$. This implies that there are only

Algorithm 5.4: Optimized STSR Computation

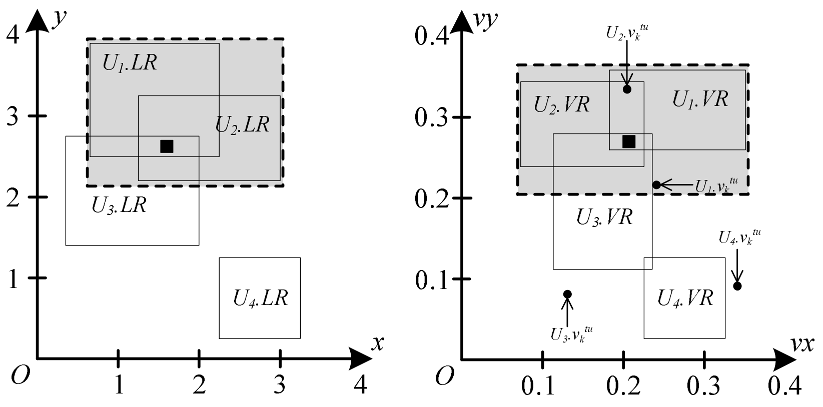
```
Input  : Current location $l_i^t$, current velocity $v_i^t$, expiry time $t_e$
Output: The optimal STSR $R(o_i)$
1  Search all update records covering $l_i^t$ and $v_i^t$ and store them in $U_{NN}$;
2  Initialize $R(o_i)$ with $LR = \{l_i^t\}$, $VR = \{v_i^t\}$ and $t_e$;
3  Initialize covered update record set $CR = \emptyset$;
4  Initialize the cost $Cost(R(o_i)) = \infty$;
5  while $Cost(R(o_i))$ does not converge do
6      Set optimal expansion record $U^*$ as null;
7      Set optimal expanded STSR $R^*$ as null;
8      foreach update record $U_j \in U_{NN}$ do
9          Construct $R'$ by expanding $R(o_i)$ to cover $U_j$;
10         Estimate $P_a(R')$ and $P_p(R')$;
11         if $Cost(R') < Cost(R^*)$ then
12             Replace $R^*$ with $R'$;
13             Replace $U^*$ with $U_j$;
14     if $U^*$ is not null then
15         Replace $R(o_i)$ with $R^*$;
16         Move $U^*$ from $U_{NN}$ to $CR$;
17 return $R(o_i)$;
```

$2^{|U_{NN}|}$ different values for the possible active update probability for $R(o_i)$. Each possible value is associated with a group of covered records. This motivates our optimization technique of modeling the record covering with a series of discrete events. Algorithm 5.4 outlines the steps of finding the optimal STSR for an object. To find the optimized STSR, the algorithm iteratively refines the current STSR. An initial STSR $R(o_i)$ is first created with minimal LR and minimal VR covering only l_i^t and v_i^t of o_i respectively. In each subsequent iteration, it tries to expand the STSR to cover one more update record from the remaining uncovered records in U_{NN}. If the estimated update cost does not further decrease after an iteration, the optimization procedure stops and returns the final STSR.

In Figure 5.4, we illustrate an example of the optimization algorithm, using the data shown in Figure 5.3. The solid square points are the loca-



(a) First Iteration



(b) Second Iteration

Figure 5.4: An running example of STSR optimization algorithm

tion and velocity of the object o_i at t_r. At the beginning of the algorithm, the STSR $R(o_i)$ is initialized with the minimum square covering the solid square points in both spaces. Since the inclusion of any update record has the same reduction effect on $P_a(R(o_i))$, the optimal update record to cover next actually has the minimum increase on the passive update probability $P_p(R(o_i))$. By checking all update records, U_2 is selected according to the selection criterion. The STSR $R(o_i)$ grows in both spatial and velocity spaces to cover the update record U_2, as shown in Figure 5.4(a). In the second iteration, as shown in Figure 5.4(b), the update record U_1 is picked since the decrease on $P_a(R(o_i))$ is still larger than the increase on $P_p(R(o_i))$. The algorithm terminates after the second iteration, when there is no other expansion that can further reduce the estimated cost. Note that this algorithm works in a greedy manner. Hence, it does not guarantee convergence to the global optimum.

The retrieval of the similar update record set U_{NN} discovers all update records covering both the location and velocity of the current object. To support such retrieval process efficiently, an R-tree index is built on the update records with respect to their location and velocity rectangles. Given the index structure, U_{NN} is simply retrieved with the issuance of a point query at location l_i^t and velocity v_i^t. The computation of $P_p(R(o_i))$ takes constant time since the total volume $Vol(R(o_i))$ can be summed up quickly by Equation 5.7. In each iteration, all the remaining update records are tested in sequence. This leads to $O(m^2)$ complexity in the worst case, if m update records are retrieved from the index structure.

5.4.3 Reducing Computation Cost

The above STSR calculation algorithm runs in quadratic complexity in terms of the number of STSR update records. Thus, the computation cost on the STSRs can be very high if the system runs the Algorithm 5.4 for every single update. We then introduce two simple strategies to reduce the computational cost, namely the *Static STSR*, and *Global Dynamic STSR* strategies. To distinguish them from the basic strategy, which finds the optimal STSR for each object update individually, we call the basic solution the *Personal Dynamic STSR* strategy.

With the *Static STSR* strategy, there is a group of fixed parameters $\{\Delta_l, \Delta_v, \Delta_t\}$. Δ_l and Δ_v are rectangles in the spatial and velocity space, centered at the origins of corresponding spaces respectively. Δ_t is a positive constant value that specifies the length between the reference time and expiry time. For an update of object o_i with location l_i^t, velocity v_i^t and time t, the location rectangle LR of $R(o_i)$ is initialized by moving Δ_l to make l_i^t align with the center of the Δ_l, i.e.,

$$\begin{cases} LR.x^{\vdash} = l_i^t.x + \Delta_l.x^{\vdash} \\ LR.x^{\dashv} = l_i^t.x + \Delta_l.x^{\dashv} \\ LR.y^{\vdash} = l_i^t.y + \Delta_l.y^{\vdash} \\ LR.y^{\dashv} = l_i^t.y + \Delta_l.y^{\dashv} \end{cases}$$

Similarly, the velocity rectangle VR of $R(o_i)$ is also constructed by expanding the velocity with margins in Δ_v on both dimensions. The expiry time of $R(o_i)$ is $t + \Delta_t$. This strategy is supposed to incur minimal computation cost, since the parameters are never updated after the specification at the beginning. As an example, if the parameter set is $\{\Delta_l = (-1, 1) \times (-2, 1), \Delta_v = (-0.2, 0.1) \times (-0.1, 0.3), \Delta_t = 5\}$, object o_i updates at time $t_r = 10$ with location $l_i^t = (10, 15)$ and velocity $v_i^t = (1, 2)$, the new STSR $R(o_i)$ is constructed with $LR = (9, 11) \times (13, 16)$, $VR = (0.8, 1.1) \times (1.9, 2.3)$ and $t_e = 15$.

The *Global Dynamic STSR* strategy, similar to the *Static STSR* strategy, adopts a set of parameters $\{\Delta_l, \Delta_v, \Delta_t\}$ for all objects in the database. However, the parameter set changes dynamically according to the system workload, and each parameter set is valid only in a limited time interval. All updates are handled with the global parameters that are valid at the updating time, as they are with the static strategy. Compared with the *Static STSR* strategy, the *Global Dynamic STSR* strategy incurs some computational cost for calculating new parameter set when the previous global parameter set is expiring. In addition, the computation on parameter re-computation can be run off-line when the system has free CPU cycles available for use, without affecting the performance of the database system.

As shown in Algorithm 5.4, *Personal Dynamic STSR strategy* always finds the optimal STSR for each object individually. However, as for the *Global Dynamic STSR* strategy, the global STSR parameters should be optimized for all objects and refreshed from time to time. In the following, we present some modifications to Algorithm 5.4 to utilize it for finding the optimal STSR parameters for *Global Dynamic STSR* strategy.

In order to find the optimal STSR for an object, Algorithm 5.4 (Line 1) first retrieves the STSR update records U_{NN} covering the updating location of the object. However, in global STSR computation, there is no such focal object, since the STSR parameters are object- and location-independent, which will act on all objects without distinction. Therefore, instead of the set of update records in U_{NN}, all STSR update records maintained in the system are utilized. Specifically, we first align the centers of all STSRs in the update records to the origin of our coordinate system. After the alignment operation, a virtual“focal” object is created. The location of the focal object is at the origin; the velocity of the focal object the average speeds of the objects on both dimensions. Then, Algorithm 5.4 is applied on the virtual object and the aligned records. Finally, the global parameters $\{\Delta_l, \Delta_v, \Delta_t\}$ are set according to the STSR returned by the algorithm, i.e., $\Delta_l = LR$, $\Delta_v = VR$ and $\Delta_t = t_e - t_r$.

5.5 Integration with Existing Indexes

The STSR-based updating protocol can be seamlessly integrated into almost all existing indexing structures in moving-object databases. In this section, we focus on incorporating the proposed protocol into two fundamental data structures: the TPR-tree [87] and the B$^+$-tree based indexes [50, 123]. The protocol can be applied to other mainstream indexes in similar ways.

5.5.1 The TPR-tree and Variants

As reviewed in Section 2.3, in the TPR-tree and its variants such as the TPR*-tree, each object has an entry in some leaf node, containing its location and velocity at the reference time. To incorporate the STSR-based updating protocol into the TPR-tree, we only need to make some minor modifications on the leaf nodes. Instead of the exact location and velocity of an object, the STSR of the object is stored as the underlying object representation in the leaf node. Such change on the leaf nodes will not affect the intermediate nodes. In the TPR-tree, each intermediate entry links to a child node and maintains a spatio-temporal bounding rectangle, which consists of a *MBR* and a *VBR* bounding the possible locations and velocities of objects within the corresponding child respectively. The location rectangle LR and the velocity rectangle VR of an STSR work in the same manner as the *MBR* and *VBR* of the TPR-tree. Therefore, no changes in the intermediate nodes are required for enabling the integration.

On the other hand, since the overall structure of the TPR-tree remains unchanged and every object entry, namely the STSR, can be regarded as the traditional spatial-temporal bounding rectangle, the existing updating and query processing algorithms on the TPR-tree can be reused without any modification. Similarly, other auxiliary mechanisms applicable to the TPR-tree, such as the R-link concurrency control strategy [75], can still be adopted directly.

In summary, with the minor change on the leaf nodes, the STSR-based updating protocol can be easily embedded into the TPR-tree and all aforementioned algorithms on the STSRs (Algorithms 5.2–5.4) can be applied directly.

5.5.2 The B$^+$-tree-based Indexes

Besides the TPR-tree and its variants, the B$^+$-tree-based indexes, such as the B^x-tree [50] and the B^{dual}-tree [123], make another prominent category of indexes in moving-object databases. This category of indexes uses the B$^+$-tree as the basic indexing structure, which sorts objects by one-dimensional keys. To enable the incorporation, similar to what we do to the TPR-tree, object's location and velocity are replaced with its STSR in corresponding leaf entry of the underlying B$^+$-tree and no other modifications are done to tree structure. Actually, the B$^+$-tree is a mature index implemented in all commercial databases, which we cannot and should not change its internal structure.

5.5.2.1 The B^x-tree

In the B^x-tree, the spatial space is divided into small cells of equal width on both dimensions. Objects are ordered by the id of the cell covering its predicted location at the reference time. This implies that the location rectangle LR of the STSR stored in the B^x-tree must also be discretized to get a linear order before inserting into the underlying B$^+$-tree.

There are two possible solutions to support the STSR-based protocol with the B^x-tree. The first one is to allow STSRs to occupy a few spatial cells. In this case, the STSR will be discretized to more than one cells, and hence it is necessary to store multiple copies of the STSR with different keys in the tree. This solution shows high flexibility in the shape of the STSR, however, reducing the storage efficiency and incurring extra processing costs on queries.

An alternate solution constrains the location rectangle of an STSR to cover exactly one cell in the partitioned spatial space. Since the STSR is constructed with constraint, the STSR of an object may not be the optimal one found by Algorithm 5.4, but a "near-optimal" one that meets the constraint. This solution sacrifices some of the tuning ability.

However, with the constraint on the shape of the location rectangle, each object has only one entry in the B^+-tree, just like it does in the original design of the B^x-tree. Therefore, both the updating and querying algorithms on the B^x-tree can be simply adopted without any modifications. Other optimizations for the B^x-tree, such as object grouping and cell size tuning introduced with the ST^2B-tree in the previous chapter and the Blink-tree concurrency control in [61], can also be applied directly. Based on this consideration, in our empirical studies in Section 5.7, we adopt the second solution while incorporating the STSR-based updating protocol into the B^x-tree, where the location rectangle of an STSR always aligns with the space partition of the B^x-tree.

5.5.2.2 The B^{dual}-tree

The B^{dual}-tree is another indexing structure built on top of the B^+-tree, but adopting the updating and querying algorithms of the TPR-tree. Similar to the TPR-tree, the B^{dual}-tree also has the notions of *MBR* and *VBR*. However, the *MBR* and *VBR* are not stored explicitly in the tree, but are derived from the one-dimensional key. Specifically, the one-dimensional key in the B^{dual}-tree is obtained based on the discretization of both spatial and velocity space. Therefore, the *MBR* and *VBR* can be derived from the

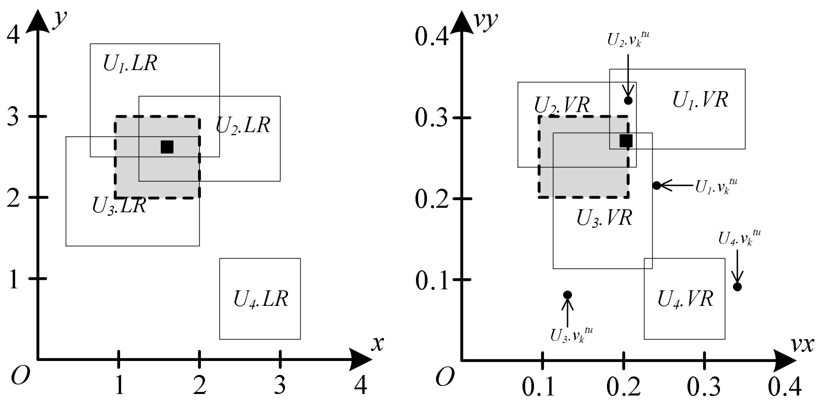


Figure 5.5: Initial location and velocity rectangle for B^{dual}-tree

corresponding key reversely.

Similar to the TPR-tree, the updating and querying algorithms of the B^{dual}-tree are directly applicable to a B^{dual}-tree with the STSR-based updating protocol. The only modification is on the initialization of the STSR (second step in Algorithm 5.4). In particular, the STSR at the beginning is initialized by expanding the location and velocity to the minimal cells containing them. In Figure 5.5, we present the initial STSR for the same moving object update in Figure 5.4. If the widths of the cells in the spatial space and the velocity space are 1 and 0.1 respectively, the new STSR before the first iteration in Algorithm 5.4 is constructed with $LR = (1, 2) \times (2, 3)$ and $VR = (0.1, 0.2) \times (0.2, 0.3)$. The subsequent expansion iterations continue normally as introduced in Section 5.4.

5.6 Approximate Query Processing

In our updating protocol, any object, whose predicted region partly overlaps with the querying range, must update its new location and velocity to the server via a passive update. This requirement is crucial when the system commits to returning fully accurate query answers, meaning the query results of Algorithm 5.3 are exactly the same as if all objects update their movement information at every timestamp. This requirement can be simply relaxed by allowing approximate query processing in the system, when the users are comfortable with query results accurate and almost complete. In this section, we explore more on this direction and propose some optimization techniques on approximate query processing to enhance the system performance. In the following, we first give a formal definition on approximate query processing.

Definition 5.7. α-Approximate Query Processing.
Given a positive real number α $(0 < \alpha \leq 1)$, the query processing algorithm is α-approximate, if for any range query (QR, t_q) with exact result R^*, the algorithm always returns some query result R that $R \subseteq R^*$ and $|R| \geq \alpha |R^*|$.

The definition above implies that 1) there is no false positive in any α-approximate query result and 2) the completeness ratio of approximate query result is guaranteed when α is large enough. In particular, when $\alpha = 1$, it degenerates to traditional exact query processing. By relaxing α to some number smaller than 1, the database engine is able to exploit the weaker completeness requirement, by probing only a smaller fraction of candidate objects.

In Algorithm 5.5, we outline the approximate range query processing algorithm. Compared with the traditional query processing algorithm for complete results in Section 5.3, the new algorithm has one additional parameter α, which indicates the minimal percentage of positive answers returned to the user. Instead of executing passive updates in random order on the candidate objects, the algorithm of approximate queries sorts all the candidate objects based on some specific order. When retrieving the new status of the objects in order, the algorithm stops and returns the cur-

Algorithm 5.5: Approximate Range Query Processing with STSR

```
Input  : Query range $QR$, query time $t_q$, tolerance parameter $\alpha$
Output: Approximate query result $R$
1  Find the object set $C \subseteq O$ that the predicted region $P_i^{t_q}$ overlaps
   with $QR$;
2  Sort all objects in $C$ based on some pre-defined order, and initialize
   result buffer $R$;
3  foreach $o_i \in C$ do
4      Remove $o_i$ from $C$;
5      if $P_i^{t_q}$ is totally covered by $QR$ then
6          Include $o_i$ in the query result $R$;
7      else
8          Send a probe request to $o_i$ for current location and velocity;
9          Compute new $pl_i^{t_q}$ with new location and velocity;
10     if $pl_i^{t_q} \in RQ$ then
11         Include $o_i$ in the query result $R$;
12     if $|R| \geq \alpha(|R| + |C|)$ then
13         break;
14 return the (partial) query result $R$;
```

rent results when the result buffer has reached desirable quantity, i.e., when $|R| \geq \alpha(|R| + |C|)$.

Given an index on the object set O, the complexity of retrieving object set C (line 1) is $\mathcal{O}(\log |O|)$; sorting objects in C (line 2) takes $\mathcal{O}(\log |C|)$ time; checking the objects (line 3-13) takes $\mathcal{O}(\alpha|C|)$ time. The complexity of Algorithm 5.5 is $\mathcal{O}(\log |O| + \log |C| + \alpha|C|)$. Suppose $|C| \ll |O|$, the complexity of Algorithm 5.5 is $\wr(\log |O|)$.

Lemma 5.8 Algorithm 5.5 always returns α-approximate results to all predictive range queries, regardless of the probing order on C.

Proof. It is straightforward to see that all objects selected in the result buffer R must be exact, since all these objects are either 1) with predicted region completely covered by the querying range or 2) with exact predicted location in querying range after passive update. In the following, we prove

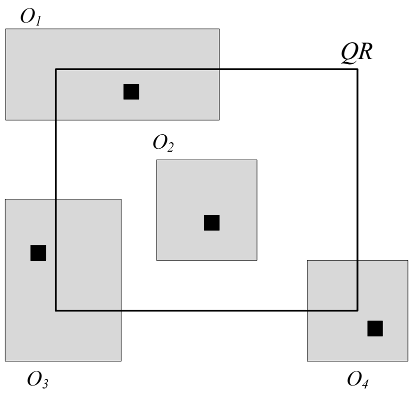


Figure 5.6: Example of approximate query processing with STSR

the completeness in terms of the parameter α. Assume the exact answer to the predictive range query $\langle QR, t_q \rangle$ is R^*, we prove in the following that the algorithm always returns some R with $|R| \geq \alpha |R^*|$.

After an iteration, the algorithm keeps two sets of the objects, R and C. R contains all objects already verified as positive answers to query $\langle QR, t_q \rangle$, and C consists of all unverified objects. If R' is the subset of C satisfying the condition of the query $\langle QR, t_q \rangle$, we have $R^* = R \cup R'$, and

$$|R^*| = |R| + |R'|$$

The algorithm terminates in two cases. In the first case, it stops when $|R| \geq \alpha(|R| + |C|)$. Because $|R'| \leq |C|$, we have

$$\frac{|R|}{|R^*|} = \frac{|R|}{|R| + |R'|} \geq \frac{|R|}{|R| + |C|} = \alpha$$

In the second case, the algorithm visits all candidates and returns the exact query result $R = R^*$. For both cases, the results will definitely satisfy the condition in Definition 5.4. □

In Figure 5.6, we present an example of approximate query processing. In this example, there are four objects whose predicted regions overlap with the query range. Their predicted locations (derived from their current locations and velocities) are represented by solid square points in the figure. In particular, o_2 is completely covered by the query range, implying that o_2 is an exact answer to the query. The other three moving objects, o_1, o_3 and o_4, are required to report their current locations and velocities for more accurate investigation. If the query quality parameter is set at 0.5, i.e., $\alpha = 0.5$, and o_1's new predicted location turns out to reside in the querying range QR, the passive updates for o_3 and o_4 can be waived. Even if o_3 and o_4 are exact answers to the query, the exclusion of o_3 and o_4 does not risk the guarantee on the completeness. On the other hand, if the system decides to probe with passive updates in the order of $\{o_4, o_3, o_1\}$, all objects have to update their new information, since both o_4 and o_3 are out of the query range after the updates.

The example above implies that the performance of the approximate query processing algorithm heavily relies on the order of probing passive updates from candidate objects. If the order is not well optimized, the approximate query processing may gain no benefit, still having to probe updates from all candidate objects in the worst case. Intuitively, the example above shows that the performance can be improved when objects that are positive answers to the query are probed before other objects. In the rest of the section, we study the design of the passive updating order.

5.6.1 Order for Individual Query

For every object o_i in the candidate set C, we assume that σ_i is the probability of o_i in the result of query $\langle QR, t_q \rangle$. Our greedy local ordering method simply ranks all the candidates objects on the probabilities $\{\sigma_i\}$ non-increasingly. In the following, we prove that sorting candidate objects in this order optimizes the expected number of passive updates for the approximate processing on any single query $\langle QR, t_q \rangle$.

Theorem 5.9 The order based on the probabilities $\{\sigma_i\}$ for query $\langle QR, t_q \rangle$ minimizes the expected number of objects requested for passive updates, regardless of the quality parameter α.

Proof. Assume that we use another order $\{\sigma_{\pi(i)}\}$ instead of the non-increasing order $\{\sigma_i\}$. There is at least one pair of objects o_i and o_j that $\sigma_i > \sigma_j$ and $\pi^{-1}(i) > \pi^{-1}(j)$. By swapping between o_i and o_j in the object order, the probability of seeing ith object in the query result increases. Therefore, the expected number of probes to answer α-approximate predictive range query is less. Therefore, such swap must the query processing efficiency by incurring less passive updates by expectation. This implies that it is always optimal by sorting the objects based on their σ_is. $\square$

Although the theorem above proves the optimality of the greedy ordering method, it remains difficult and unclear on how to calculate the exact probability σ_i. We simply use the ratio of overlap as an estimation of the probability σ_i. Recall the example in Figure 5.6. If σ_i is defined as the overlapping ratio, we have $\sigma_1 = 0.425$, $\sigma_2 = 1$, $\sigma_3 = 0.33$ and $\sigma_4 = 0.25$. The resulting order will be $\{\sigma_2, \sigma_1, \sigma_3, \sigma_4\}$. Note that o_2 will not be probed since the probability of o_2 in the query result is exactly 1.

5.6.2 Order for Multiple Queries

Local ordering minimizes the expected number of passive updates for a single query. When there are multiple queries under processing at a single timestamp, running local orderings on all these queries may not be an optimal option, since some objects may involve in multiple queries. In Figure 5.7, we show an example of multiple predictive range queries involving

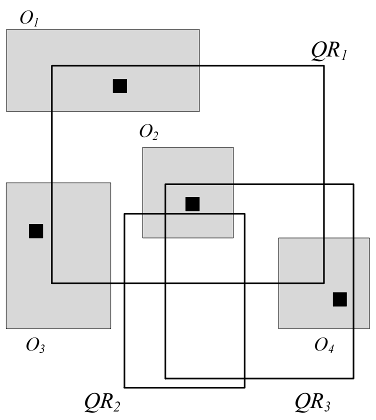


Figure 5.7: Example of multiple approximate query processing

the predicted locations of four moving objects. The optimal ordering for the first query $\langle QR[1], t_q[1]\rangle$ is no longer optimal for such query combination. For query $\langle QR[2], t_q[2]\rangle$, there is no need to probe any objects except o_2 whose predicted region overlaps with $QR[2]$; for query $\langle QR[3], t_q[3]\rangle$, o_2 and o_4 should be probed. Therefore, $\{\sigma_2, \sigma_4, \sigma_1, \sigma_3\}$ is a better order since we can avoid probing o_1 if $\alpha = 0.5$.

Assuming the system receives a group of queries as $\{\langle QR[1], t_q[1]\rangle, \ldots, \langle QR[h], t_q[h]\rangle, \ldots\}$, submitted to the system on the same timestamp, we use σ_i^h to denote the probability of o_i's predicted location resides in $QR[h]$ at query predictive time $t_q[h]$. An optimal global ordering provides an order on all candidate objects, which is capable of minimization on expected passive updates on these objects. Unfortunately, the problem of finding optimal global ordering is NP-hard. In the following, we present a polynomial-time reduction from *Multiset Covering Problem* [84] to global ordering problem.

Theorem 5.10 Finding optimal global ordering is generally NP-hard problem.

Proof. In *Multiset Covering*, there is a multiset $M = \{(I_1, n_1), \ldots, (I_k), (n_k)\}$ indicating there are n_i copies of item I_i in the set. Another set of candidate covers, i.e., $S = \{S_1, S_2, \ldots, S_m\}$, are also provided, in which each S_j is

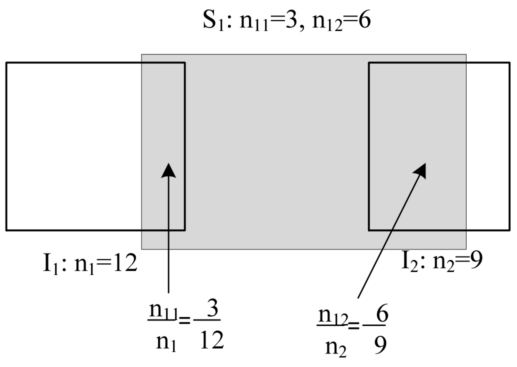


Figure 5.8: Example of reduction from multiset covering to global ordering

itself a multiset on the item collection $\{I_1, \ldots, I_k\}$. A cover subset $S' \subseteq S$ is a valid cover of the multiset M, if every I_i appears at least n_i times in S'. A cover set S^* is optimal, if S^* is the smallest among all valid cover sets. Interestingly, the problem is NP-hard, even when there are only 2 items, i.e., $k = 2$. Therefore, we only need to construct a reduction from multiset covering problem with $k = 2$ to our global ordering problem. In particular, we construct $k = 2$ parallel queries in the 2-dimensional space, as is shown in Figure 5.8. For every candidate cover S_j, we build a predicted range covering the left query with overlap of $\frac{n_{1j}}{n_1}$ and right query with overlap of $\frac{n_{2j}}{n_2}$. Here, n_{ij} is number of items I_i in cover S_j and n_i is the expected number of objects in target multiset. It is thus easy to show that the optimal cover set can be retrieved, if the optimal global ordering on our problem is available. This proves that the problem of computing global ordering is also NP-hard. $\square$

Due to the hardness of the problem, we simply apply some greedy strategy to construct some suboptimal global ordering on the candidate objects, as is shown in Algorithm 5.6. In our algorithm, we maintain a set of *Unresolved Queries*. At the beginning, the set of unresolved queries Q^u is initialized by including all queries from the users, i.e., $Q^u = \{\langle QR[1], t_q[1] \rangle, \ldots,$

Algorithm 5.6: Multiple Query Processing

```
Input  : Query set $\mathcal{Q} = \{\langle QR[1], t_q[1]\rangle, \ldots, \langle QR[h], t_q[h]\rangle, \ldots\}$,
         tolerance parameter $\alpha$
Output: Query result set $\mathcal{R} = \{R_1, \ldots, R_h, \ldots\}$
1  Initialize unresolved query set
   $\mathcal{Q}^u = \{\langle QR[1], t_q[1]\rangle, \ldots, \langle QR[h], t_q[h]\rangle, \ldots\}$;
2  Initialize query result set $\mathcal{R} = \emptyset$;
3  while $Q^u \neq \emptyset$ do
4      Find $o_i$ with maximal $\sum_{\langle QR[h], t_q[h]\rangle \in Q^u} \sigma_i^h$;
5      Send probe request to $o_i$ for passive update;
6      Update all query result in $Q^u$ based on current status of $o_i$;
7      foreach $\langle QR[h], t_q[h]\rangle \in Q^u$ do
8          if enough results are available in $R_h$ for $\langle QR[h], t_q[h]\rangle$ then
9              Include current results $R_h$ in $\mathcal{R}$;
10             Remove $\langle QR[h], t_q[h]\rangle$ from $Q^u$;
11 return $\mathcal{R} = \{R_1, \ldots, R_h, \ldots\}$;
```

$\langle QR[h], t_q[h]\rangle, \ldots\}$. In following iterations, the algorithm selects next object o_i for probing with the maximal probabilities to the unresolved queries, i.e., $\sum_{\langle QR[h], t_q[h]\rangle \in Q^u} \sigma_i^h$. After receiving the recent status of o_i, our algorithm updates all the query results remaining in Q^u. A query $\langle QR[h], t_q[h]\rangle$ is removed from Q^u, if $|R_h| \geq \alpha(|R_h| + |C_h|)$, where R_h is the current result to the query and C_h is the objects waiting for further investigation. The algorithm thus recalculates the global ordering of the remaining objects and starts next iteration. The iterations terminate when all queries in Q^u are cleared.

Suppose we have n objects to check, the complexity of finding the o_i (line 4) is $\mathcal{O}(\log n)$. In the worst case, we need to probe all n objects to resolve all the queries. In each iteration, only one object is probed and eliminated, whereas $n = n - 1$; and the object is validated with each query in Q^u. Therefore, the complexity of Algorithm 5.6 is $\mathcal{O}(n \log n + n|Q^u|)$.

5.7 Experimental Evaluation

In this section, we report our empirical studies that evaluate effectiveness and efficiency of the STSR-based updating protocol.

5.7.1 Experimental Settings

Three sources of real and *semi*-real datasets were used in our experiments. Figure 5.9 shows the maps of road networks of the datasets.

TRK: TRK is a real dataset provided by R-tree Portal [2], which contains the trajectories of 276 trucks moving in Athens metropolitan area (see Figure 5.9(a)). The trucks update at a rate of 30*sec*.

EC: EC is another real dataset as a part of the e-Courier datasets [1]. e-Courier keeps track of the movements of all its couriers all over UK (see Figure 5.9(b)). The couriers report their locations (GPS records) every 10*sec*. We crawled e-Courier for one week and extracted 587 objects (i.e., trajectories) that moved nonstop for over 120ts, i.e., had at least 120 continuous updates in the system.

SIN: Due to the lack of large real moving object datasets, we used Brinkhoff generator [12] to generate a set of synthetic movements based on the real road map of Singapore (see Figure 5.9(c)). We generated SIN datasets of different sizes and used the one containing 100K objects by default.

Table 5.2 summarizes the specifications of the three data sources above, including the data space, maximum object speed and the mapping from physical time to logical time (a timestamp).

On the other hand, the query load of the experiments consists of a given number of predictive range queries. Each of these range queries is square-

(a) TRK: Athens metropolitan

(b) EC: UK

(c) SIN: Singapore

Figure 5.9: Maps of various data sources

sized, with a preset side length. Since the datasets differ in data space size, we use *qlen* to represent the percentage of the side length of the query over the length of the entire data space. In the experiments, *qlen* is varied from 0.25% to 4% (e.g., 120–2058m for SIN datasets). Queries follow the same distribution as the objects. Specifically, the location of a randomly picked object is used as the center of the query. The average predictive time of queries varies from $1ts$ to $256ts$. The query frequency $qfqy$ varies from 1 to 256, meaning that there are 1 to 256 queries per timestamp on average.

In summary, Table 5.3 lists the parameters and their values used in the experiments, where default values for variable parameters are shown in bold.

Table 5.2: Specifics of data sources

Data source	**TRK**	**EC**	**SIN**
Space (long side)	47255.6m	287409.0m	51455.0m
Maximum speed	33.5m/s	54.3m/s	325m/s
Logical timestamp	30sec	10sec	1sec

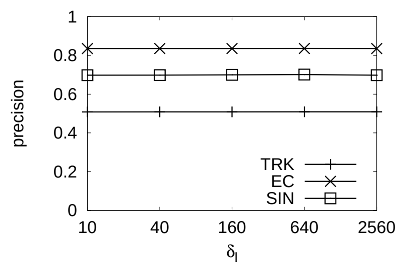
Table 5.3: Experimental parameters and values

Parameter	**Setting**
Datasets	TRK, EC, **SIN**
Time duration	120ts
Number of objects *numObj*	25K, 50K, **100K**, 200K, 400K
Query side length *qlen*	0.25%, 0.5%, **1%**, 2%, 4%
Query predictive time *qpdt*	1ts, 4ts, **16s**, 64ts, 256ts
Query frequency *qfqy*	1, 4, **16**, 64, 256
α	0.2, 0.4, 0.6, 0.8, 1.0
Δ_t	1ts, 2ts, 4ts, 8ts, **16ts**, 64ts, 256ts
δ_l	10m, 40m, **160**, 640m, 2560m
δ_v	1m/ts, 4m/ts, **16m/ts**, 64m/ts, 256m/ts

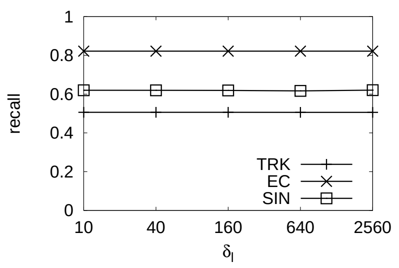
All the programs are implemented in C++ and run on a PC with 2.33 GHz Intel Core2 Duo CPU, 2.25 GB RAM and 200 GB SATA disk.

5.7.2 The Results

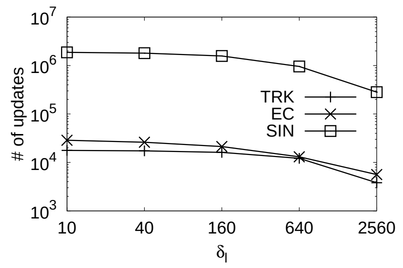
In this section, we discuss the experimental results on the basis of the settings introduced in previous section. Specifically, Section 5.7.2.1 focuses on the effectiveness of STSR-based protocol without using any index. Section 5.7.2.2 compares our proposal against existing updating protocols. Then, in Section 5.7.2.3, we show the results of the performance of approximate query processing. Section 5.7.2.4 presents results when different implementation strategies are applied on different index structures.



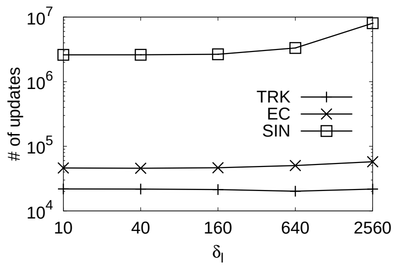
(a) Query Precision



(b) Query Recall



(c) # of Active Updates



(d) # of All Updates

Figure 5.10: Effect of δ_l on TRK, EC and SIN

5.7.2.1 The basic STSR-based updating protocol

We first studied the performance of the basic STSR-based updating protocol, regardless of the underlying index. We investigated the effect of the three elements of the STSR: the spatial and velocity rectangles and the valid time duration, i.e., the length of time between expiry time and reference time. Following the instructions in Section 5.4.3, a static global parameter set is used in each experiment in this section, where

$$\begin{cases} \Delta_l.x^{\vdash} = \Delta_l.x^{\vdash} = -\delta_l \\ \Delta_l.y^{\dashv} = \Delta_l.y^{\dashv} = \delta_l \\ \Delta_v.x^{\vdash} = \Delta_v.x^{\vdash} = -\delta_l \\ \Delta_v.y^{\dashv} = \Delta_v.y^{\dashv} = \delta_l \end{cases} \tag{5.9}$$

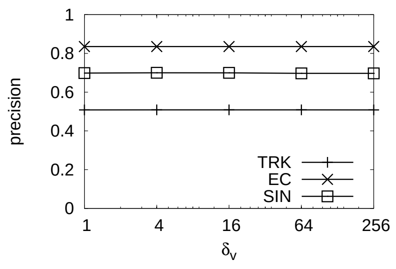
The parameters δ_l, δ_v, and Δ_t, whose values are listed in Table 5.3, are introduced in the experiments to control the size of the STSR.

Figures 5.10–5.12 show the effect of δ_l, δ_v and Δ_t on the updating workload and the quality of query results.

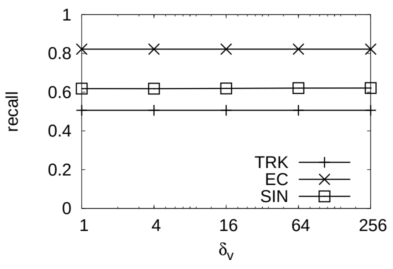
First, it is worth noting that query precision and recall are not affected by the shape of the STSR. During query processing, an object is added to or excluded from the results if its predicted region is fully contained or disjoint with the query region. Otherwise, a passive update is invoked. The query precision and recall are primarily decided by the predictability of object's motion itself, i.e., predicting based on the location and velocity of the object at query issuing time.

Hence, we put our focus on the number of updates. As δ_l increases, meaning a larger spatial rectangle LR of the STSR, the number of active updates decreases at the expense of more passive updates, resulting in an increase in the total number of updates. As shown in Figure 5.11, δ_v has a similar effect on the numbers of both updates. With a larger δ_v, the predicted region expands faster. The STSR is more likely to enclose the location of the object and fewer active updates are incurred. On the other hand, a larger predicted region has higher probability to intersect with the query region and more passive updates are required consequently.

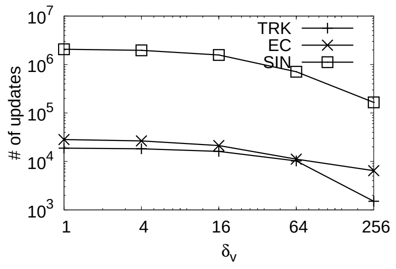
The temporal duration of STSRs Δ_t, on the other hand, affects the update performance differently. The number of active updates and the total number of updates both decrease with a longer temporal length Δ_t. When Δ_t is smaller than a given threshold (i.e., $4ts$ for TRK and EC, $16ts$ for SIN), the smaller Δ_t is, the faster the STSR expires. In this case, the number of active updates increases with smaller Δ_t. However, when Δ_t is larger than the threshold, the number of active updates increases slowly with larger Δ_t. Since the expiry time is far away in the future, a small deviation on the velocity will lead to a quite large distance between the predicted location



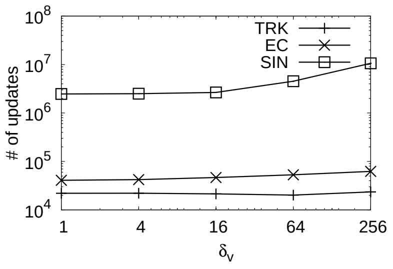
(a) Query Precision



(b) Query Recall



(c) # of Active Updates



(d) # of All Updates

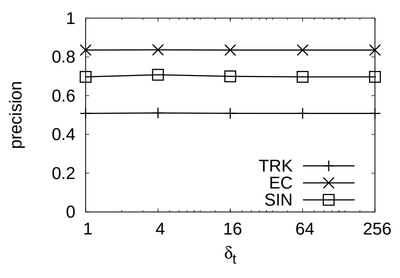
Figure 5.11: Effect of δ_v on TRK, EC and SIN

and the predicted region at the expiry time. Therefore, the STSRs stand higher probability to fail in the consistency verification. On the other hand, the number of total number of updates is more stable since the increase in active updates also brings in a decrease in passive updates in return.

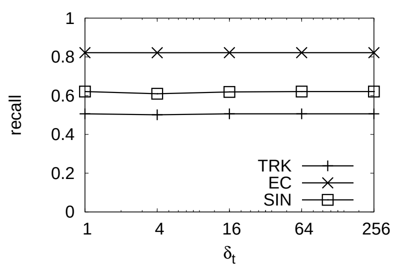
5.7.2.2 Motion Functions

We next investigated the effect of different updating protocols on both updating and querying performance. We compared the STSR-based updating protocol with the improved spatial bounded updating protocol [95], as introduced in Section 5.1. In [95], the authors propose the improved spatial bounded updating protocol together with the STP-tree, which integrates the protocol into the TPR-tree. In the rest of this section, we call this improved spatial bounded updating protocol "STP" for short.

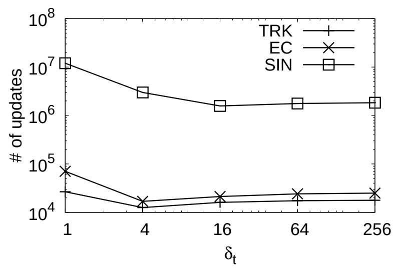
The STP updating protocol uses recursive functions to model the move-



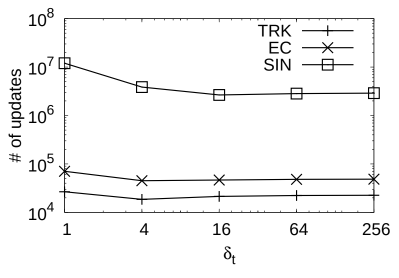
(a) Query Precision



(b) Query Recall

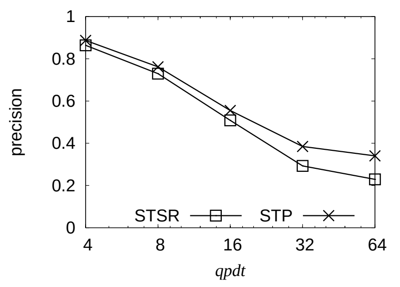


(c) # of Active Updates

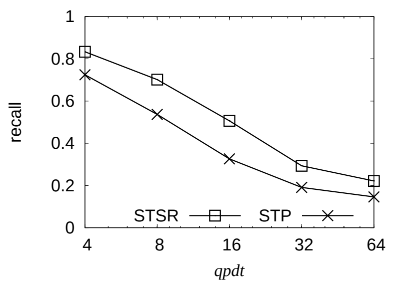


(d) # of All Updates

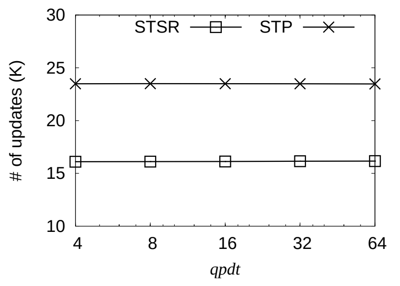
Figure 5.12: Effect of Δ_t on TRK, EC and SIN



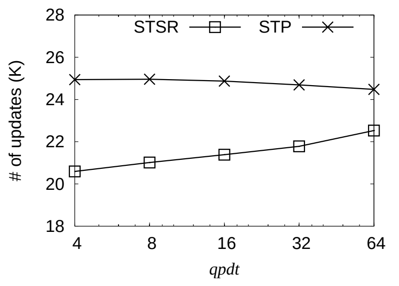
(a) Query Precision



(b) Query Recall

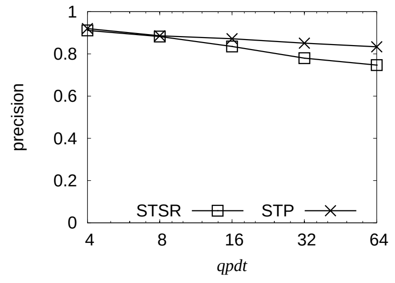


(c) # of Active Updates

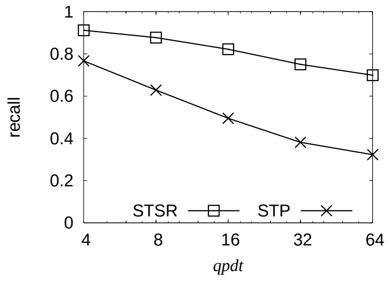


(d) # of All Updates

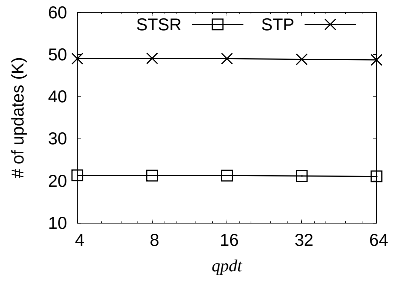
Figure 5.13: Effect of query predictive time *qpdt* on TRK



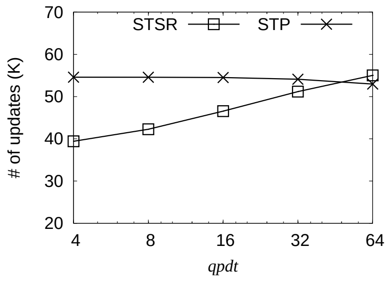
(a) Query Precision



(b) Query Recall



(c) # of Active Updates



(d) # of All Updates

Figure 5.14: Effect of query predictive time *qpdt* on EC

ments of objects. Specifically, a client (object) keeps h historical records and derives a recursive motion function from the h records. Then, a D dimensional polynomial function is derived as a simple approximation of the recursive motion function. On an update, the polynomial function is sent to the server, and the system can predict object location using the polynomial function. The recursive function is more complex than the polynomial function and therefore is supposed to make a more accurate prediction on the object's future locations. An active update is issued if the spatial error between the predicted locations derived by the polynomial in the server and the recursive function is larger than the error bound de in any of the subsequent H timestamps. In our experiments, h, D, de and H are set to 8, 5, 160m and 30ts respectively. The query processing of the STP is similar to that of the STSR, i.e., the system asks for an update (passive) if it cannot determine whether the object is inside the query region or not.

Figures 5.13–5.14 show the results while varying the query predictive time *qpdt* on TRK and EC respectively. Since the results in this section are largely related to the predictability of the moving object data, it makes more sense to investigate the real data. Regarding this, we only show results on the two real datasets, i.e., TRK and EC, throughout this section.

For both TRK and EC, the STP method results in a higher query precision, while the STSR protocol has a higher query recall. The differences in precision and recall increase with the query predictive time. On TRK, when $qpdt = 16$, the precision of STSR is about 10% less than that of STP; however, the recall is about 1.6times of that of STP. On EC, when $qpdt = 64$, the difference in query precision is less than 0.05, while difference in query recall is as large as 0.5.

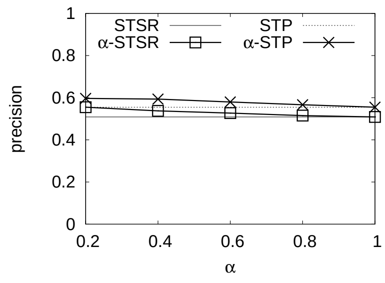
Considering the updating load, the number of active updates is less affected by the query predictive time for both methods, while the STSR effectively incurs fewer the number of active updates. The number of total updates of the STP is pretty stable with the predictive time. However, as for the STSR protocol, the number of total updates increases rapidly with the query predictive time. Since the predicted region is larger with longer predictive time, the objects are more likely to issue passive updates. When $qpdt = 16$, the number of total updates of STSR protocol is 20% less than that of STP; when $qpdt = 64$, on EC, the number of total updates of STSR is slightly higher than that of STP.

As shown in Table 5.2, one timestamp of TRK(EC) corresponds to $30sec(10sec)$. Therefore, a predictive time of $qpdt = 64$ means to find the objects in specific region after $32mins(10mins)$. Based on this considerations, although our experiments show that STSR incurs higher updating costs than STP on EC when query predictive time is larger than 64, we argue that the prediction is meaningful only on a near future.

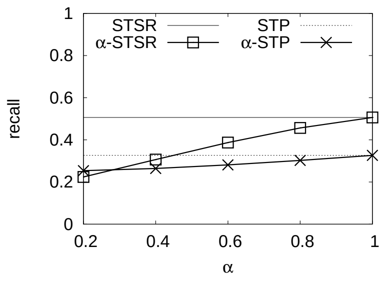
5.7.2.3 Approximate Query Processing

Next, we study the performance of processing approximate queries (Algorithm 5.5 and Algorithm 5.6). First, Figures 5.15–5.16 show the results on TRK and EC datasets varying the approximate query quality parameter α. In the figures, "STSR" and "STP" represent exact query processing with STSR and STP protocols; "α-STSR" and "α-STP" denote corresponding approximate query processing algorithms. Note that when $\alpha = 0$, there is no passive updates at all. When $\alpha = 1$, the approximate query processing algorithm becomes the same as the exact query algorithm, returning the complete results to all queries. Algorithm 5.6 is adopted here to process multiple queries at a timestamp, and, on average, there are 16 queries at each timestamp by default.

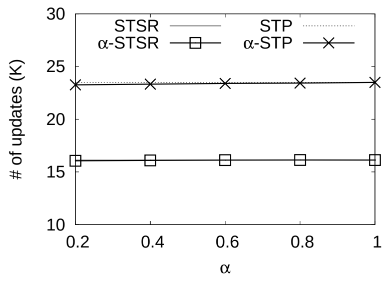
Generally, a larger value of α leads to a lower query precision but higher recall, where the increase in query recall is more significant. The query recall increases linearly with α, since the larger α is, a larger percentage of predicted answers are required before the approximate query processing terminates. On the contrary, the query precision increase slightly since when more objects are returned for a query, the chance of a false positive becomes higher. On the other hand, considering the updating workload, we can hardly observe any effect of α on the number of active updates, however, there is a distinct increase in the total number of updates with α. This is because the approximate query processing largely reduces the number of query-driven, passive updates. The fewer answers are required by a query, the larger the save on passive updates is. As for EC dataset, when $\alpha = 0.6$, i.e., the approximate query processing algorithm returns 60% of the positive



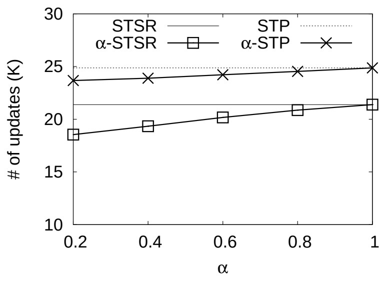
(a) Query Precision



(b) Query Recall

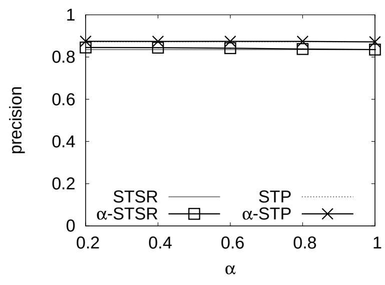


(c) # of Active Updates

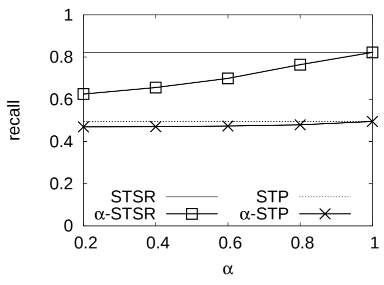


(d) # of All Updates

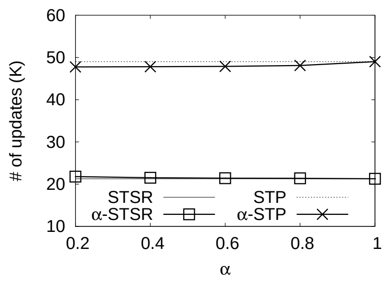
Figure 5.15: Effect of approximate query parameter α on TRK (order for multiple queries)



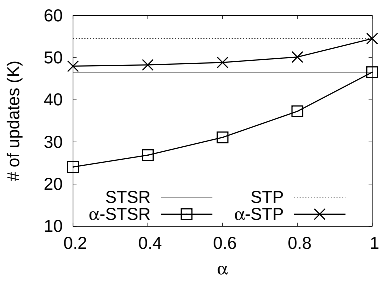
(a) Query Precision



(b) Query Recall

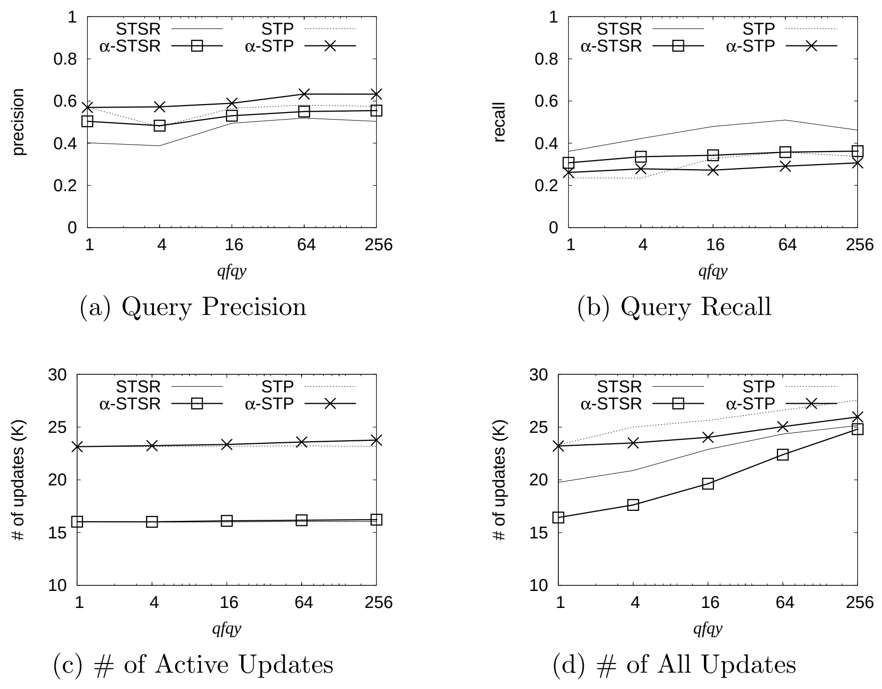


(c) # of Active Updates



(d) # of All Updates

Figure 5.16: Effect of approximate query parameter α on EC (order for multiple queries)



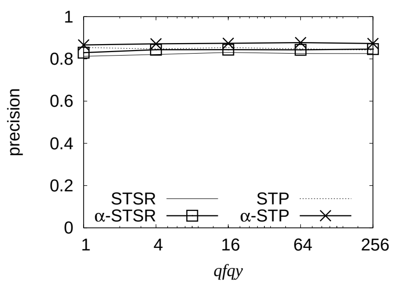
(a) Query Precision (b) Query Recall

(c) # of Active Updates (d) # of All Updates

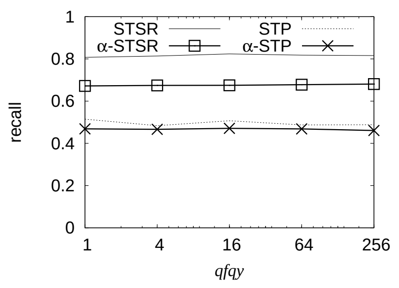
Figure 5.17: Effect of query frequency $qfqy$ on TRK

query answers, with a decrease of 12% (3% resp.) on the query recall, the STSR (STP resp.) updating protocol saves upto 30% (15% resp.) on the number of updates. As we can see in the figures, to enable approximate queries is an effective and efficient option to further reduce the updating workload.

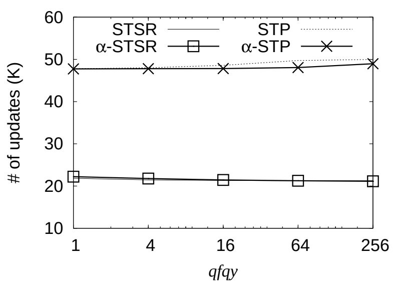
Second, we examine the effect of query frequency $qfqy$ on the performance of approximate query processing. We varied the number of queries per timestamp from 1 to 256. The performances of *0.5*-approximate queries ($\alpha = 0.5$) on both datasets are shown in Figures 5.17–5.18 respectively. Intuitively, the query frequency $qfqy$ does not affect the query precision and recall, as shown in Figure 5.18 for EC dataset. As for TRK dataset in Figure 5.17, the query precision and recall oscillate with $qfqy$, which implies that $qfqy$ has insignificant impact on the query accuracies. Since the precision and recall on TRK is quite low compared to those of EC, they are



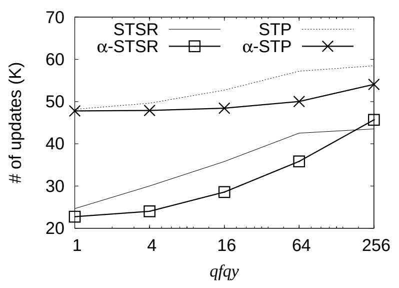
(a) Query Precision



(b) Query Recall



(c) # of Active Updates

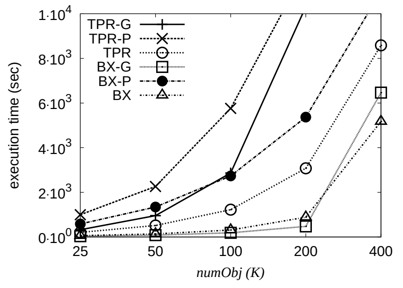


(d) # of All Updates

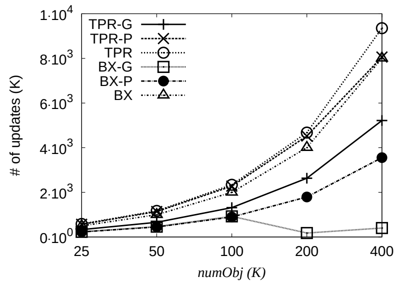
Figure 5.18: Effect of query frequency $qfqy$ on EC

more easily affected by the randomness of the experiments. As for the updates, it is as expected that the number of active updates is not affected by the query frequency, while the total number of updates increases for both exact and approximate queries. Naturally, the number of passive updates increases linearly with the number of queries per timestamp. It is notable that the total number of updates with approximate query processing increases slowly than it is with exact query processing. The gap between the increasing rates is practically proportional to the value of α, which is 0.6 in Figures 5.17–5.18.

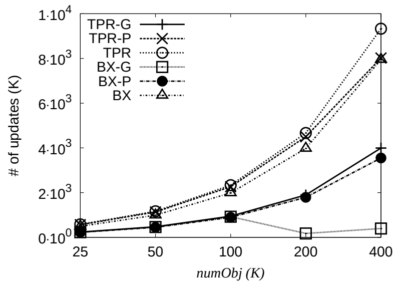
Note that in Figures 5.15–5.16, the STSR-based updating protocol improves the query recall and largely reduces the number of updates at the expense of little decrease in query precision. The performances of the two protocols are consistent with our previous findings in Section 5.7.2.2. We do not present the computation cost of approximate query processing, since



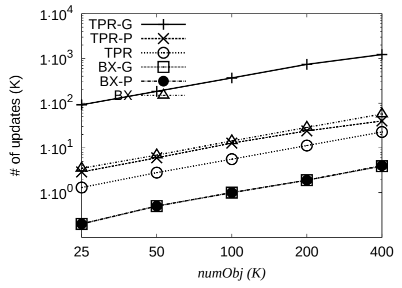
(a) # of Passive Updates



(b) # of All Updates



(c) # of Active Updates



(d) # of Passive Updates

Figure 5.19: Effect of number of objects *numObj* on SIN

the additional cost of computing the overlaps and sorting the candidate objects in Algorithm 5.6 is hardly perceptible, compared to the updating cost.

5.7.2.4 The STSR strategies

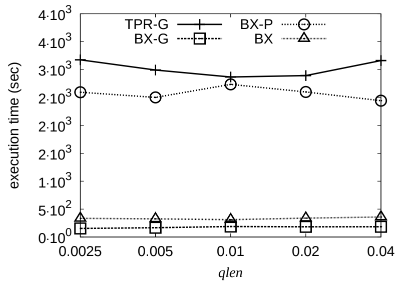
We now proceed to study the performance of different STSR strategies as introduced in Section 5.4.3, namely the *Static STSR*, the *Global Dynamic STSR* and the *Personal Dynamic STSR*. We implemented all the three STSR strategies on top of both the TPR-tree and the B^x-tree as explained in Section 5.5. In the remaining part of this section, 'BX'/'TPR' denotes using the B^x-tree/TPR-tree with static STSR; 'BX-G'/'TPR-G' denotes using the B^x-tree/TPR-tree with global dynamic STSR; 'BX-P'/'TPR-P' denotes using the B^x-tree/TPR-tree with personal dynamic STSR.

We first examined the scalability of all update strategies by varying the object cardinality from 25K to 400K. Figure 5.19 illustrates the total processing time and the number of updates. The total processing includes all computations on queries, STSR computation and updates of the moving objects, in 120 consecutive timestamps.

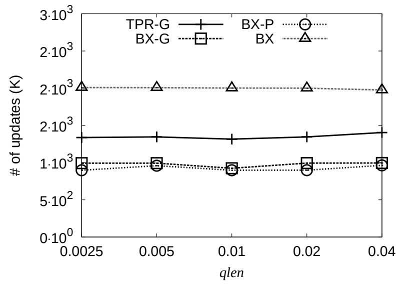
Comparing to the static and personal dynamic strategies, global dynamic strategy largely decreases the amount of active updates, while adds a number of passive updates. The personal dynamic strategy, on the other hand, reduces the number of passive updates at the expense of more active updates. This is because that with global dynamic strategy, the size of STSRs is much larger than those generated by personal dynamic strategy.

Second, we examined the effect of various query parameters on the performance of different STSR update strategies. From Figure 5.19, we have already seen that the 'TPR-G' requires the highest execution time and is the least scalable in terms of object cardinality, while the 'TPR' always incurs the highest update times. For clear illustration, we leave 'TPR-G' and 'TPR' out. The results of these two methods are omitted in the following figures and analysis.

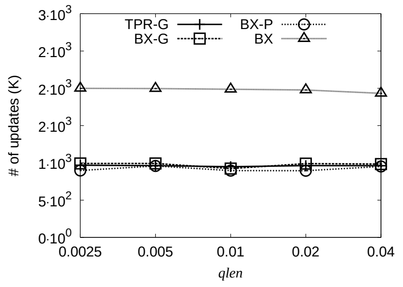
Figure 5.20 shows the performance of STSR strategies with respect to different query size. Specifically, the query side length varies from 250 to 4,000m. In general, the B^x-tree with global dynamic STSR is the best among all considering both total processing time and total number of updates. The B^x-tree with static STSR, although performs good in terms of total proceeding time, sends out the largest total number of updates. For all methods except 'TPR-G', the number of passive updates increases with the query size, since the STSRs of more objects intersects with the query region. For the 'TPR-G', the STSR is relatively large, and thus the performance is less affected by the query size.



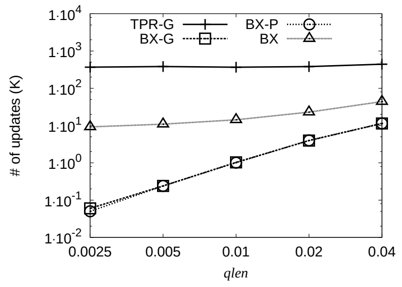
(a) # of Passive Updates



(b) # of All Updates

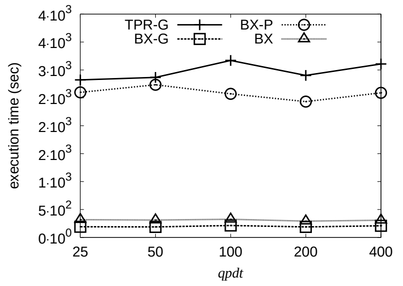


(c) # of Active Updates

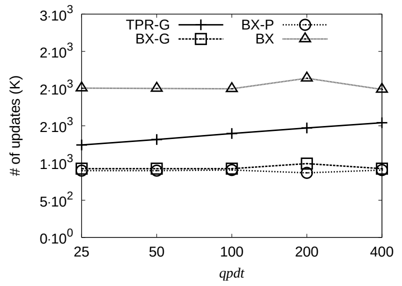


(d) # of Passive Updates

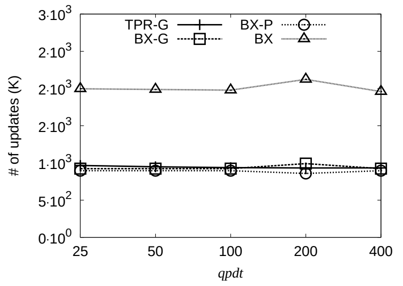
Figure 5.20: Effect of query side length *qlen* on SIN



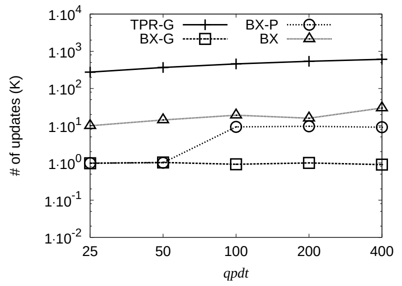
(a) # of Passive Updates



(b) # of All Updates

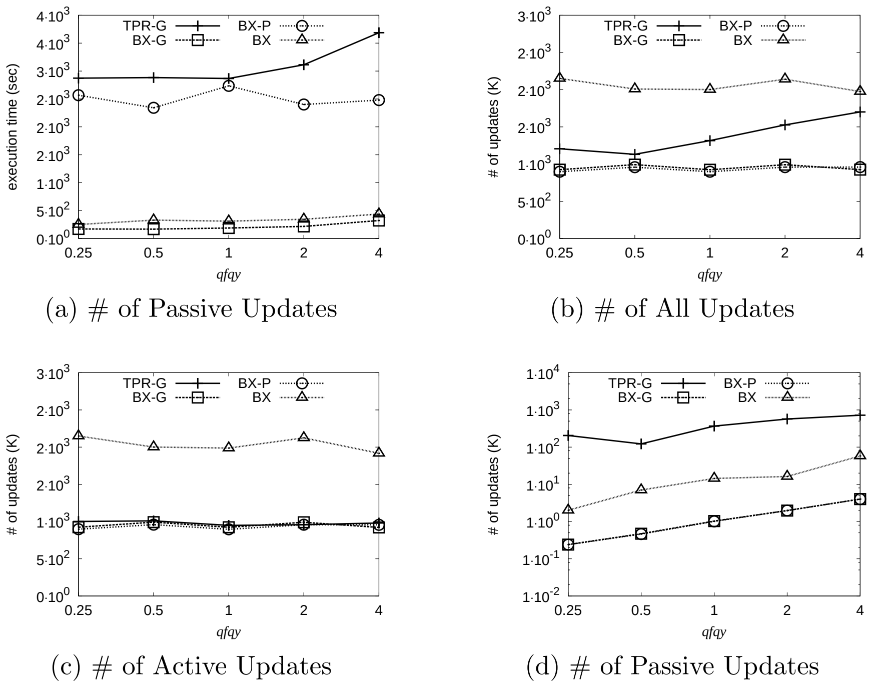


(c) # of Active Updates



(d) # of Passive Updates

Figure 5.21: Effect of query predictive time *qpdt* on SIN



(a) # of Passive Updates (b) # of All Updates

(c) # of Active Updates (d) # of Passive Updates

Figure 5.22: Effect of query frequency $pfqy$ on SIN

Figure 5.21 shows the effect of query prediction time *qpdt* on different STSR strategies. We vary the query prediction time from 0 (current query) to 120ts. As the prediction time changes, the total processing time shows the similar trends as those of other parameters, i.e., the 'BX-G' and 'BX' both run much faster than the 'GX-P' and 'TPR-G'. As 'BX-G' tunes the parameters for the STSR periodically, it has the best performance in terms of the total processing time. 'BX-P' also minimizes the total number of updates, however, at the expense of longer processing time (for computing the parameters for objects individually). In general, as the predication time increases, more passive updates are incurred.

Finally, we study the effect of query frequency $qfqy$ on the performance Figure 5.22 shows the effect of query prediction time. For the STSR strategies, with more frequent queries, the number of passive updates increases a lot while the number of active updates decreases slightly. It is worth noticing that the total processing time is not much influenced by the query frequency, although the total number of queries increases a lot (by 16times). The time for computing STSR dominates the processing time of the whole index.

5.7.3 Result Summary

In summary, the proposed STSR-based updating protocol achieves reasonably good query precision and outperforms the STP updating method regarding the query recall. In addition, the STSR-based updating protocol can effectively reduce the updating workload when query predictive time is in a reasonable range. Enabling approximate query processing can further reduce the updating workload to a large extent while still provide a guarantee on the quality of query results. Among all the STSR strategies and indexing structures, the 'BX-G', i.e., the B^x-tree with the global dynamic STSR strategy, achieves the best performance with respect to the computational and updating workload in general.

5.8 Summary

In this chapter, we propose a generic updating protocol for moving-object databases. By utilizing the concept of Spatio-Temporal Safe Region (STSR), objects actively send motion updates to the database server only when the prediction error of their current movement is no longer bounded. To guarantee the accuracy of predictive query answers, the database server asks objects for their latest locations and velocities, if they are potential answers to a query. To minimize the updating workload, a cost model is presented to estimate the approximate updating cost, depending on the most recent update records in the system. Based on the cost model, an algorithm is designed to construct an optimized STSR that minimizes the expected updating cost. To cut the cost further, the STSR-based updating protocol is extended by allowing incomplete query answers with bounded accuracy. We carefully evaluate three different implementation strategies, including static STSR, dynamic global STSR and dynamic personal STSR. Experiments on the TPR-tree and the B^x-tree show that the STSR based protocol significantly reduces the updating cost while achieving high accuracy of predictive query answers.

Chapter 6

Conclusion

6.1 Conclusion

With the increasing availability of accurate geo-positioning, e.g., using GPS receivers, and the rapid deployment of mobile devices capable of communicating wirelessly with their surroundings, it is fast becoming possible to track the locations of large populations of moving objects, e.g., individuals with mobile phones or vehicles with on-board navigation systems. This capability opens up a wide range of applications, including a variety of monitoring applications, traffic control, tourist services, and mobile commerce. Such applications greatly rely on the ability to manage and query these objects. While databases are commonly used to store, organize and retrieve large amount of data, moving-object databases are specially designed for these emerging applications, considering the characteristics of moving objects, especially the mobility.

It is well-known that indexing techniques are of primary importance to the performance of any database. While there are quite a number of indexing techniques in the literature, each of them claims that it outperforms the others, rendering it hard to choose one for a specific application. In this study, we first proposed a benchmark for evaluating important performance properties of techniques for indexing moving objects. The benchmark includes a general dataset and workload generator, together with a standard evaluation suit. The generator can produce data simulating moving objects under various circumstances; the evaluation suit examines almost every aspect of an index. We applied the benchmark to six representative moving-object indexes, i.e., the TPR-tree, TPR*-tree, RUM*-tree, B^x-tree, B^{dual}-tree and STRIPES. To the best of our knowledge, no previous studies have compared this many indexes under the same standard. The results of the benchmark study provide valuable insights into the strengths and weaknesses of these indexes, which can serve as guidelines for future index development and the selection of a proper index for a specific application.

In our benchmark study, it was found that the index performance is seriously affected by the distribution of objects. For the purpose of eliminating such performance degradation, we then examined the problem of tuning moving-object indexes to make it adaptive to the diversity and change of workloads in various applications. We first identified several forms of data diversity in moving-object applications and their impacts on existing indexes. Then, we proposed the ST^{2}B-tree index that can automatically adjust itself to avoid performance degradation caused by these data diversity. To deal with the non-uniformity in space, the ST^{2}B-tree dynamically partitions the space into regions of different object densities. For each region, the ST^{2}B-tree manages the objects inside with a uniform grid, whose granularity of the grid is adaptively adjusted to fit the object density. To adapt to the changes with time, the ST^{2}B-tree employs a "multi-tree" technique, where two sub-trees are used to index objects regarding their last update time. The basic idea is to rebuild the sub-trees periodically and alternately.

During the rebuilding process, the space partitioning is adjusted according to the latest recorded object distribution. The results of our experimental study showed the superiority of the ST^{2}B-tree over the other indexes and confirmed that the ST^{2}B-tree is adaptive to the variety and changes in workloads of moving object applications.

Given the numerous moving-objects indexes in the literature, there is little room left for improving performance through better indexing techniques. Based on this consideration, we further explored the possibility of performance optimization by reducing the updating workload of a moving-object database. A generic updating protocol was developed. With the concept of STSR, the new protocol relaxes the accuracy of object tracking, leading to a decrease in the number of updates. However, lower tracking accuracy results in higher inaccuracy of query results. To guarantee the query accuracy, updates are requested from objects that are potential answers of the query given the imprecise tracking information. We presented a cost model that analyzes the approximate updating cost and an algorithm of constructing the optimized STSR was introduced to minimize the expected updating workload. We also showed that the updating protocol is independent of the underlying index structure and can be easily incorporated into any existing indexes. The experimental results on various real moving-object datasets showed that the proposed updating protocol significantly reduces the updating workload while achieving high accuracy of query answers.

While most of the works on moving-object databases focus on designing efficient indexing techniques and query processing algorithms, this study opens the door to performance improvement from totally different perspectives. While the impact of data varieties on database performance has been overlooked, this study is the first to investigate the tuning problem exclusively for moving-object databases to handle these varieties. Furthermore, no previous work has ever explored the possibility of improving the performance by reducing the workload in the first place, with the help of a more elegant updating protocol.

6.2 Future Work

The benchmark presented in this study has already covered various datasets from uniform datasets in the ideal case to datasets that simulate in-network objects. Although we tried our best to simulate as many circumstances as possible, lack of real moving-object datasets remains a common problem of current moving-object benchmarks and researches. Although we have found some real datasets as used in Chapter 5, available real datasets are quite limited, especially in terms of size. A promising future work is to extend the benchmark with real datasets of large size.

In Chapter 4, we identified the necessity of tuning and presented a generic tuning framework for moving-object databases. Although the tuning framework is generally applicable to all moving-object databases, we only discussed and provided guidelines for the tuning of the index. A direct extension of work in this direction is to explore the potential performance benefits obtained by tuning other components of the database, such as the cache and the query optimizer.

Finally, the results in Chapter 5 revealed that the quality of predictive query answers largely relies on the predictability of the object's motion. It was also shown that the predictability is practically independent of the motion model adopted. Since objects can control there own movements, it is hard to predict their future locations and motions. However, considering the fact that an object's movement is typically restrained by the underlying road network in practice, it will be interesting to utilize historical statistics on the road network to make a more precise prediction.

Previously Published Materials

Chapter 3 revises a previous publication [20].

Chapter 4 revises previous publications [22, 21].

Chapter 5 revises previous publication [23].

Bibliography

[1] eCourier. http://api.ecourier.co.uk/. [cited at p. 188]

[2] R-tree Portal. http://www.rtreeportal.org/. [cited at p. 188]

[3] TPR*-tree. http://www.rtreeportal.org/code.html. [cited at p. 55]

[4] U.S. Census Bureau - TIGER/Line. http://www.census.gov/geo/www/tiger/. [cited at p. 51]

[5] Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian S. Jensen, Lydia E. Kavraki, Patrice Koehl, Ming C. Lin, Dinesh Manocha, Dimitris N. Metaxas, Brian Mirtich, David M. Mount, S. Muthukrishnan, Dinesh K. Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolfson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002. [cited at p. 8, 14]

[6] Mohammed Eunus Ali, Egemen Tanin, Rui Zhang, and Lars Kulik. Load balancing for moving object management in a p2p network. In *DASFAA*, pages 251–266, 2008. [cited at p. 9]

[7] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60, 1999. [cited at p. 125]

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990. [cited at p. 24, 29, 31]

[9] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002. [cited at p. 36]

[10] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J.*, 15(3):229–249, 2006. [cited at p. 36]

[11] Laurynas Biveinis, Simonas Saltenis, and Christian S. Jensen. Main-memory operation buffering for efficient r-tree update. In *VLDB*, pages 591–602, 2007. [cited at p. 31]

[12] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002. [cited at p. 51, 188]

[13] F. Warren Burton, John G. Kollias, D. G. Matsakis, and V. G. Kollias. Implementation of overlapping b-trees for time and space efficient representation of collections of similar files. *Comput. J.*, 33(3):279–280, 1990. [cited at p. 21]

[14] Ying Cai, Kien A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Mobile Data Management*, pages 27–38, 2004. [cited at p. 9]

[15] Yuhan Cai and Raymond T. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD Conference*, pages 599–610, 2004. [cited at p. 16]

[16] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003. [cited at p. 23]

[17] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997. [cited at p. 83]

[18] Nan Chen, Lidan Shou, Gang Chen, and Jinxiang Dong. Adaptive indexing of moving objects with highly variable update frequencies. *J. Comput. Sci. Technol.*, 23(6):998–1014, 2008. [cited at p. 25, 27, 128]

[19] Nan Chen, Lidan Shou, Gang Chen, Yunjun Gao, and Jinxiang Dong. Predictive skyline queries for moving objects. In *DASFAA*, pages 278–282, 2009. [cited at p. 38, 39]

[20] Su Chen, Christian S. Jensen, and Dan Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008. [cited at p. 8, 26, 29, 30, 213]

[21] Su Chen, Mario A. Nascimento, Beng Chin Ooi, and Kian-Lee Tan. Continuous online index tuning in moving object databases. *ACM Trans. Database Syst.*, 35(3), 2010. [cited at p. 213]

[22] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. St^{2}b-tree: a self-tunable spatio-temporal b$^+$-tree index for moving objects. In *SIGMOD Conference*, pages 29–42, 2008. [cited at p. 40, 213]

[23] Su Chen, Beng Chin Ooi, and Zhenjie Zhang. An adaptive updating protocol for reducing moving object databases workload. *PVLDB*, 3(1):735–746, 2010. [cited at p. 213]

[24] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Querying imprecise data in moving object environments. *IEEE Trans. Knowl. Data Eng.*, 16(9):1112–1127, 2004. [cited at p. 8, 9]

[25] Chin-Wan Chung, Sunghee Choi, and Yong-Jin Choi. Closest pair queries in spatio-temporal databases. *Comput. Syst. Sci. Eng.*, 20(2), 2005. [cited at p. 37]

[26] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. [cited at p. 24]

[27] Victor Teixeira de Almeida and Ralf Hartmut Güting. Indexing the trajectories of moving objects in networks. *GeoInformatica*, 9(1):33–60, 2005. [cited at p. 9, 22]

[28] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009. [cited at p. 25]

[29] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. BerlinMOD: a benchmark for moving object databases. *VLDB J.*, 18:1335–1368, 2009. [cited at p. 48]

[30] Tobias Emrich, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, Naixin Xu, and Andreas Züfle. Reverse k-nearest neighbor monitoring on mobile objects. In *GIS*, pages 494–497, 2010. [cited at p. 36, 37]

[31] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996. [cited at p. 125]

[32] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974. [cited at p. 31]

[33] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *SIGMOD Conference*, pages 319–330, 2000. [cited at p. 8, 14]

[34] Elias Frentzos. Indexing objects moving on fixed networks. In *SSTD*, pages 289–305, 2003. [cited at p. 9, 22]

[35] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004. [cited at p. 9, 42]

[36] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: anonymizers are not necessary. In *SIGMOD Conference*, pages 121–132, 2008. [cited at p. 10]

[37] Shuqiao Guo, Zhiyong Huang, H. V. Jagadish, Beng Chin Ooi, and Zhenjie Zhang. Relaxed space bounding for moving objects: a case for the buddy tree. *SIGMOD Record*, 35(4):24–29, 2006. [cited at p. 30]

[38] Ralf Hartmut Güting, Victor Teixeira de Almeida, Dirk Ansorge, Thomas Behr, Zhiming Ding, Thomas Höse, Frank Hoffmann, Markus Spiekermann, and Ulrich Telle. Secondo: An extensible dbms platform for research prototyping and teaching. In *ICDE*, pages 1115–1116, 2005. [cited at p. 48]

[39] Ralf Hartmut Güting, Victor Teixeira de Almeida, and Zhiming Ding. Modeling and querying moving objects in networks. *VLDB J.*, 15(2):165–190, 2006. [cited at p. 8, 9, 14]

[40] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005. [cited at p. 3]

[41] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984. [cited at p. 7, 21, 24, 29, 31]

[42] Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *SSTD*, pages 306–324, 2003. [cited at p. 38]

[43] Jiawei Han, Zhenhui Li, and Lu An Tang. Mining moving object, trajectory and traffic data. In *DASFAA*, pages 485–486, 2010. [cited at p. 10]

[44] Cecilia Hernández, M. Andrea Rodríguez, and Mauricio Marín. A p2p meta-index for spatio-temporal moving object databases. In *DASFAA*, pages 653–660, 2008. [cited at p. 9]

[45] Kathleen Hornsby and Max J. Egenhofer. Modeling moving objects over multiple granularities. *Ann. Math. Artif. Intell.*, 36(1-2):177–194, 2002. [cited at p. 8, 14]

[46] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pages 479–490, 2005. [cited at p. 41]

[47] Zhiyong Huang, Hua Lu, Beng Chin Ooi, and Anthony K. H. Tung. Continuous skyline queries for moving objects. *IEEE Trans. Knowl. Data Eng.*, 18(12):1645–1658, 2006. [cited at p. 38]

[48] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003. [cited at p. 41]

[49] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005. [cited at p. 92]

[50] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004. [cited at p. 8, 20, 24, 26, 28, 30, 33, 35, 36, 40, 41, 55, 56, 87, 95, 96, 127, 128, 162, 176, 177]

[51] Christian S. Jensen, Dan Lin, Beng Chin Ooi, and Rui Zhang. Effective density queries on continuously moving objects. In *ICDE*, page 71, 2006. [cited at p. 38]

[52] Christian S. Jensen, Dalia Tiesyte, and Nerius Tradisauskas. The cost benchmark—comparison and evaluation of spatio-temporal indexes. In *DASFAA*, pages 125–140, 2006. [cited at p. 46, 48]

[53] Christian S. Jensen, Dalia Tiesyte, and Nerius Tradisauskas. Robust b+-tree-based indexing of moving objects. In *MDM*, page 12, 2006. [cited at p. 25, 33, 56]

[54] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004. [cited at p. 8]

[55] Dmitri V. Kalashnikov, Sunil Prabhakar, Susanne E. Hambrusch, and Walid G. Aref. Efficient evaluation of continuous range queries on moving objects. In *DEXA*, pages 731–740, 2002. [cited at p. 41]

[56] James M. Kang, Mohamed F. Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, pages 806–815, 2007. [cited at p. 36, 37, 41]

[57] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Mobile Data Management*, pages 113–120, 2002. [cited at p. 8, 23, 30, 31]

[58] Caifeng Lai, Ling Wang, Jidong Chen, Xiaofeng Meng, and Karine Zeitouni. Effective density queries for moving objects in road networks. In *APWeb/WAIM*, pages 200–211, 2007. [cited at p. 9, 38]

[59] Thuy Thi Thu Le and Bradford G. Nickerson. Efficient search of moving objects on a planar graph. In *GIS*, page 41, 2008. [cited at p. 17]

[60] Mong-Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003. [cited at p. 23, 30, 31]

[61] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. [cited at p. 57, 146, 178]

[62] Yifan Li, Jiong Yang, and Jiawei Han. Continuous k-nearest neighbor search for moving objects. In *SSDBM*, pages 123–126, 2004. [cited at p. 41]

[63] Zhenhui Li, Bolin Ding, Jiawei Han, Roland Kays, and Peter Nye. Mining periodic behaviors for moving objects. In *KDD*, pages 1099–1108, 2010. [cited at p. 10]

[64] Zhenhui Li, Ming Ji, Jae-Gil Lee, Lu An Tang, Yintao Yu, Jiawei Han, and Roland Kays. Movemine: mining moving object databases. In *SIGMOD Conference*, pages 1203–1206, 2010. [cited at p. 10]

[65] Dan Lin, Elisa Bertino, Reynold Cheng, and Sunil Prabhakar. Position transformation: a location privacy protection method for moving objects. In *SPRINGL*, pages 62–71, 2008. [cited at p. 10]

[66] Dan Lin, Elisa Bertino, Reynold Cheng, and Sunil Prabhakar. Location privacy in moving-object environments. *Transactions on Data Privacy*, 2(1):21–46, 2009. [cited at p. 10]

[67] Hechen Liu and Markus Schneider. Querying moving objects with uncertainty in spatio-temporal databases. In *DASFAA*, pages 357–371, 2011. [cited at p. 9]

[68] Yiming Ma, Dmitri V. Kalashnikov, and Sharad Mehrotra. Toward managing uncertain spatial information for situational awareness applications. *IEEE Trans. Knowl. Data Eng.*, 20(10):1408–1423, 2008. [cited at p. 9]

[69] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003. [cited at p. 21]

[70] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004. [cited at p. 23, 28, 41]

[71] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001. [cited at p. 106]

[72] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, pages 634–645, 2005. [cited at p. 15, 18, 23, 28, 41, 42]

[73] Jussi Myllymaki and James H. Kaufman. Dynamark: A benchmark for dynamic spatial indexing. In *Mobile Data Management*, pages 92–105, 2003. [cited at p. 46, 48]

[74] Mario A. Nascimento and Jefferson R. O. Silva. Towards historical r-trees. In *SAC*, pages 235–240, 1998. [cited at p. 21]

[75] Vincent Ng and Tiko Kameda. The r-link tree: A recoverable index structure for spatial data. In *DEXA*, pages 163–172, 1994. [cited at p. 57, 146, 176]

[76] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, 2010. [cited at p. 21]

[77] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984. [cited at p. 23]

[78] Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. Stripes: An efficient index for predicted trajectories. In *SIGMOD Conference*, pages 637–646, 2004. [cited at p. 16, 25, 30, 34, 40, 55, 57, 127]

[79] Kostas Patroumpas and Timos K. Sellis. Monitoring orientation of moving objects around focal points. In *SSTD*, pages 228–246, 2009. [cited at p. 22, 25]

[80] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000. [cited at p. 16, 22]

[81] Dieter Pfoser and Nectaria Tryfona. Capturing fuzziness and uncertainty of spatiotemporal objects. In *ADBIS*, pages 112–126, 2001. [cited at p. 9]

[82] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002. [cited at p. 8]

[83] Cecilia Magdalena Procopiuc, Pankaj K. Agarwal, and Sariel Har-Peled. Star-tree: An efficient self-adjusting index for moving objects. In *ALENEX*, pages 178–193, 2002. [cited at p. 24, 29]

[84] Sridhar Rajagopalan and Vijay V. Vazirani. Primal-dual rnc approximation algorithms for set cover and covering integer programs. *SIAM J. Comput.*, 28(2):525–540, 1998. [cited at p. 185]

[85] Bharat Rao and Louis Minakakis. Evolution of mobile location-based services. *Commun. ACM*, 46(12):61–65, 2003. [cited at p. 2]

[86] Simonas Saltenis and Christian S. Jensen. Indexing of moving objects for location-based services. In *ICDE*, pages 463–472, 2002. [cited at p. 24, 29]

[87] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000. [cited at p. 7, 16, 20, 24, 28, 30, 31, 35, 36, 40, 41, 87, 98, 169, 176]

[88] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984. [cited at p. 7, 21, 24]

[89] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. [cited at p. 34]

[90] Jochen H. Schiller and Agnès Voisard. *Location-Based Services*. Morgan Kaufmann, 2004. [cited at p. 2]

[91] Yasin N. Silva, Xiaopeng Xiong, and Walid G. Aref. The rum-tree: supporting frequent updates in r-trees using memos. *VLDB J.*, 18(3):719–738, 2009. [cited at p. 24, 40]

[92] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997. [cited at p. 8, 14]

[93] Zhexuan Song and Nick Roussopoulos. Hashing moving objects. In *Mobile Data Management*, pages 161–172, 2001. [cited at p. 23]

[94] Zhexuan Song and Nick Roussopoulos. Seb-tree: An approach to index continuously moving objects. In *Mobile Data Management*, pages 340–344, 2003. [cited at p. 22]

[95] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD Conference*, pages 611–622, 2004. [cited at p. 16, 20, 193]

[96] Yufei Tao and Dimitris Papadias. Efficient historical r-trees. In *SSDBM*, pages 223–232, 2001. [cited at p. 21]

[97] Yufei Tao and Dimitris Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001. [cited at p. 22]

[98] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, pages 334–345, 2002. [cited at p. 41]

[99] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002. [cited at p. 41]

[100] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003. [cited at p. 20, 24, 29, 30, 32, 55, 127, 128]

[101] Yufei Tao and Xiaokui Xiao. Primal or dual: which promises faster spatiotemporal search? *VLDB J.*, 17(5):1253–1270, 2008. [cited at p. 26, 30, 48, 128]

[102] Yufei Tao, Jun Zhang, Dimitris Papadias, and Nikos Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004. [cited at p. 96]

[103] Yannis Theodoridis. Ten benchmark database queries for location-based services. *Comput. J.*, 46(6):713–725, 2003. [cited at p. 48]

[104] Yannis Theodoridis, Michalis Vazirgiannis, and Timos K. Sellis. Spatio-temporal indexing for large multimedia applications. In *ICMCS*, pages 441–448, 1996. [cited at p. 22]

[105] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004. [cited at p. 9]

[106] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *Comput. J.*, 43(4):325–343, 2000. [cited at p. 21]

[107] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Benchmarking access methods for time-evolving regional data. *Data Knowl. Eng.*, 49(3):243–286, 2004. [cited at p. 47]

[108] Haojun Wang, Roger Zimmermann, and Wei-Shinn Ku. Distributed continuous range query processing on moving objects. In *DEXA*, pages 655–665, 2006. [cited at p. 9]

[109] Longhao Wang, Yu Zheng, Xing Xie, and Wei-Ying Ma. A flexible spatio-temporal indexing scheme for large-scale gps track retrieval. In *MDM*, pages 1–8, 2008. [cited at p. 22]

[110] Paul Werstein. A performance benchmark for spatiotemporal databases. In *The 10th Annual Colloquium of the Spatial Information Research Centre*, pages 365–373, 1998. [cited at p. 47]

[111] Ouri Wolfson, Sam Chamberlain, Son Dao, Liqin Jiang, and Gisela Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998. [cited at p. 9, 17]

[112] Ouri Wolfson, Liqin Jiang, A. Prasad Sistla, Sam Chamberlain, Naphtali Rishe, and Minglin Deng. Databases for tracking mobile units in real time. In *ICDT*, pages 169–186, 1999. [cited at p. 9, 17]

[113] Ouri Wolfson, A. Prasad Sistla, Sam Chamberlain, and Yelena Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999. [cited at p. 9, 17]

[114] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *SSDBM*, pages 111–122, 1998. [cited at p. 3, 8, 14, 17]

[115] Ouri Wolfson and Huabei Yin. Accuracy and resource consumption in tracking and location prediction. In *SSTD*, pages 325–343, 2003. [cited at p. 17]

[116] Wei Wu, Wenyuan Guo, and Kian-Lee Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *ICDE*, pages 1116–1125, 2007. [cited at p. 9]

[117] Wei Wu, Fei Yang, Chee Yong Chan, and Kian-Lee Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM*, pages 132–139, 2008. [cited at p. 36, 37, 41]

[118] Tian Xia and Donghui Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, page 77, 2006. [cited at p. 36, 37, 41]

[119] Xiaopeng Xiong and Walid G. Aref. R-trees with update memos. In *ICDE*, page 22, 2006. [cited at p. 8, 24, 30, 31, 55, 56]

[120] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005. [cited at p. 15, 18, 23, 28, 41, 42]

[121] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, page 13, 2006. [cited at p. 18, 23]

[122] Xiaomei Xu, Jiawei Han, and Wei Lu. Rt-tree: An improved r-tree indexing structure for temporal spatial databases. In *4th International Symposium on Spatial Data Handling (SSDH)*, pages 1040–1049, 1990. [cited at p. 21, 22]

[123] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. The b^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3):379–400, 2008. [cited at p. 20, 25, 28, 33, 40, 55, 104, 127, 139, 140, 169, 176, 177]

[124] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001. [cited at p. 92]

[125] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005. [cited at p. 15]

[126] Meihui Zhang, Su Chen, Christian S. Jensen, Beng Chin Ooi, and Zhenjie Zhang. Effectively indexing uncertain moving objects for predictive queries. *PVLDB*, 2(1):1198–1209, 2009. [cited at p. 20]

[127] Panfeng Zhou, Donghui Zhang, Betty Salzberg, Gene Cooperman, and George Kollios. Close pair queries in moving object databases. In *GIS*, pages 2–11, 2005. [cited at p. 22, 37]

[128] Manli Zhu, Dik Lun Lee, and Jun Zhang. k-closest pair query monitoring over moving objects. In *MDM*, page 14, 2006. [cited at p. 37]