12-2006

Yellow Tree: A Distributed Main-memory Spatial Index Structure for Moving Objects

Hariharan Gowrisankar

YELLOW TREE – A DISTRIBUTED MAIN-MEMORY SPATIAL INDEX STRUCTURE FOR MOVING OBJECTS

By

Hariharan Gowrisankar

B.E Anna University, Chennai, India 1999

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Spatial Information Science and Engineering)

The Graduate School

The University of Maine

December, 2006

Advisory Committee:

Silvia Nittel, Assistant Professor in Spatial Information Science and Engineering, Advisor

Max J. Egenhofer, Professor in Spatial Information Science and Engineering

Michael F. Worboys, Professor in Spatial Information Science and Engineering

YELLOW TREE – A DISTRIBUTED MAIN-MEMORY SPATIAL INDEX STRUCTURE FOR MOVING OBJECTS

By Hariharan Gowrisankar

Thesis Advisor: Dr Silvia Nittel

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Spatial Information Science and Engineering)
December, 2006

Mobile devices equipped with wireless technologies to communicate and positioning systems to locate objects of interest are common place today, providing the impetus to develop *location-aware* applications. At the heart of location-aware applications are moving objects or objects that continuously change location over time, such as cars in transportation networks or pedestrians or postal packages. Location-aware applications tend to support the tracking of very large numbers of such moving objects as well as many users that are interested in finding out about the locations of other moving objects. Such location-aware applications rely on support from database management systems to model, store, and query moving object data. The management of moving object data exposes the limitations of traditional (spatial) database management systems as well as their index structures designed to keep track of objects' locations. Spatial index structures that have been designed for geographic objects in the past primarily assume data are foremost of static nature (e.g., land parcels, road networks, or airport locations), thus requiring a limited amount of index structure updates and reorganization over a period of time. While handling moving objects however, there is an incumbent need for continuous

reorganization of spatial index structures to remain up to date with constantly and rapidly changing object locations.

This research addresses some of the key issues surrounding the efficient database management of moving objects whose location update rate to the database system varies from 1 to 30 minutes. Furthermore, we address the design of a highly scaleable and efficient spatial index structure to support location tracking and querying of large amounts of moving objects. We explore the possible architectural and the data structure level changes that are required to handle large numbers of moving objects. We focus specifically on the index structures that are needed to process spatial range queries and object-based queries on constantly changing moving object data. We argue for the case of main memory spatial index structures that dynamically adapt to continuously changing moving object data and concurrently answer spatial range queries efficiently. A proof-of-concept implementation called the *yellow tree*, which is a distributed main-memory index structure, and a simulated environment to generate moving objects is demonstrated. Using experiments conducted on simulated moving object data, we conclude that a distributed main-memory based spatial index structure is required to handle dynamic location updates and efficiently answer spatial range queries on moving objects. Future work on enhancing the query processing performance of yellow tree is also discussed.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Silvia Nittel for giving me the opportunity to pursue this line of research as well as her time, support, guidance and patience. I would also like to express my sincere thanks to my advisory committee members, Dr. Max Egenhofer and Dr. Mike Worboys. I am very grateful to the faculty, staff and all my colleagues in the Department of Spatial Information Science and Engineering, especially Lars, Anuket, Arda, Greg, Pete, Ram, Chitra, Sharad, Farhan, Vijay, Sabeshan and others.

Special mention goes to Kanna, Lalitha, Nitin, Uday, Preeti, and Arvind for their support. I am also obliged for the assistance received from the University of Maine. Special mention goes to Mireille and Sarah Joughin from The Office of International Programs.

I would like to convey my most heartfelt thanks to my parents, sister, in laws and the rest of my family for their love, support and encouragement; it has been most instrumental to my success. Finally, thanks to my wife Malini for standing by and helping me through everything. This work is partially supported by the National Science Foundation under NSF grant number EPS-9983432. This support is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Technological advancements such as wireless communication, location tracking, miniaturization of devices, and computer hardware have provided us with accurate positioning systems, ubiquitous wireless devices, and an affordable small-form computer hardware that offers better performance. More and more people own devices such as cell phones communicating through wireless technologies, personal digital assistants, and positioning systems or location sensors. Growing interest in these technologies has enabled such applications as Mobile Workforce Management, Asset Tracking, Location Based Services, Multi-user gaming, and Wireless Emergency Services. An interesting aspect of all these applications is that the end users are becoming more and more mobile and thus generate time varying, location-rich data.

A location-based application provides two types of services: *push* and *pull*. In pull services, a moving object requests services based on its location, while in push services, information is passed on to moving objects voluntarily via a trigger based on their current or future location (WAP 2002). For applications to cater to moving objects, a key aspect is their ability to handle large amounts of location data that are continuously updated. Traditionally applications rely on database management system (DBMS) software to store, update, and extract data efficiently (Ramakrishnan and Gehrke 2000). A DBMS internally employs algorithms to organize, store, and retrieve data in such a way that creating, updating, deleting, archiving, and querying large amounts of data is efficient.

Although DBMSs have been successful in modeling and querying most of comparatively static real world phenomenon via relations or objects, the support for mobile or continuously moving objects has been lacking. A DBMS that is intended to support moving object data is termed a *moving objects database* (MOD) (Wolfson et al. 1998).

Data of continuously moving objects pose new challenges for MODs in the areas of location modeling, query processing, indexing, and accounting for inherent positional uncertainty in location data. For MOD applications to be effective, the queries by mobile users need to be executed efficiently, meaning that the query response for the user should be as fast as possible. Typical queries in such a system include both spatial and attribute queries, such as "Where is the nearest yellow cab with regard to my location?" "How many trucks are in and around 5 miles of downtown Boston?" For such queries to be executed in a timely manner, the current locations of moving objects have to be managed efficiently. For this purpose, index structures are necessary. An index structure is an auxiliary data structure that stores search key entries about objects in an ordered fashion to reduce the search time to access the objects themselves (Ramakrishnan and Gehrke 2000). For example, a search key could be the identifier of objects or the value of one or more object attributes such as "Find the car with license plate '816 FLA'" or "Find all cars that are red BMWs." Similarly, an object's geographic location can be the search key. In this case, index structures are called spatial indexes. In an index structure, an entry is represented by a search key value, which holds an object's specific information and a pointer to the storage location of the data record. Indexing is a critical component for any MOD that handles large amounts of data, since it avoids a sequential scan of the

location attribute of all objects in the system, thereby improving update and query processing performance.

Since DBMSs are designed to manage and preserve large amounts of data for a long time, the data records as well as the index structures are stored on hard disks. Furthermore, data records and index structures are updated in the scope of transaction, that is, any changed data have to be written safely to the disk before a transaction commits so that the persistence of changes can be guaranteed. Active database systems were designed to automate database operations using procedures called triggers (Dayal et al 1988). Triggers involve an event (such as updating a tuple in database) that activates an action based on a condition. Triggers in active database systems enable the database administrator to execute an action before or after a database operation, but do not necessarily improve the performance speed of database operations within the scope of a transaction. Hence, disk-based storage of index structures and trigger based active database systems are ill fitted and inefficient for managing MO data with constantly changing locations. The rate at which an object's location is updated in MODs pushes the boundaries of efficiency of disk-based spatial index structures with regard to reorganizing frequently at a rate that becomes unacceptable, considering the number of write operations to be performed.

1.2 Research Objectives

This thesis is aimed at designing an efficient and highly scalable spatial index structure for managing the current locations of very large numbers of moving objects (>100 Million objects), capturing and tracking their point-based location. The index structure

needs to adapt well under uniform and skewed moving object update traffic and provide a high throughput, that is, minimal time for the reorganization of the index structure.

1.3 Research Questions

To design an index structure for moving object traffic, some of the key research questions to be addressed are:

a) How is indexing of moving objects different from indexing traditional spatial vector data?

b) Is a main-memory based spatial index viable for moving object indexing and does it provide sufficient indexing efficiency?

c) Can a distributed main-memory-based spatial index structure scale up sufficiently for concurrent location update rates of very large numbers of moving objects or a very high frequency of object updates, and how can such a distributed spatial index be conceptualized?

1.4 Contributions

The hypothesis of this thesis is that it is sufficient to deploy a distributed, main-memory based spatial index structure to efficiently handle constantly changing moving object data with varying, but very high insert or update rates. We also assume that since main-memory storage space is limited, a distributed main-memory-based spatial index provides sufficient and effective tracking of varying number of moving objects, and scales seamlessly to indexing >100 Million moving objects concurrently.

In this thesis we introduce a novel distributed main-memory spatial index structure for moving object databases that is based on distributed, cooperating spatial

database servers each of which deploys a main-memory based spatial index structure to track moving objects. The spatial database servers cooperate with regard to tracking moving objects and answering queries about those objects. Furthermore, the spatial database servers cooperate to address local and time-based burst in load, that is, in the number of objects to handle in specific areas, the update load of objects at certain times, and the number of queries. The thesis encompasses the implementation and performance analysis of the *yellow tree* prototype, which is a proof-of-concept implementation that contains the investigated concepts. The thesis prototype is named yellow tree, that is, a yellow book to retrieve the current location of all managed moving objects.

1.5 Concepts / Terminology

Before we continue with more details in the subsequent chapters, we provide a brief overview of concepts and terminology to understand the research problem better and the solution.

1.5.1 Positioning Technologies

Positioning technologies rely on the principle that if distance and time could be measured precisely from known points of reference whose position is expressed as a function of time, then the location of any point in space can be determined (Leick 2004). Measured distance between the points of reference (that transmit their time through radio signals) and the receiver can be calculated as the product of signal travel time and speed of light. The accuracy of the determined location or the quality of positioning depends on the signal strength, since the layers in the atmosphere have varying distorting effects on radio signals. Different positioning strategies are required to position objects based on their

environment. Objects in outdoor environment typically use Global Positioning Systems (GPS) technologies, while indoor objects use infrared positioning technology.

1.5.2 Base Station

A base station is an access point in a wireless communication network that transmits and receives radio signals for every participating client within the network. Global System for Mobile communications (GSM) base stations are equipped with a tower, radio transmitters, and receivers and cover an area of about 10 sq. miles (Brian and Tyson 2006). Base stations are also typically equipped with highly accurate positioning information about their own location, which make them good reference points or control points. Wireless base stations provide standard 802.11g, a data rate of up to 54 Mbps. They are also used in home networks for laptop computers or personal digital assistants (Marks 2003).

Base stations can be distinguished based on different allocated frequencies depending on their range of operation. Radio base stations for FM receivers, mobile phone base stations, and wireless base stations for laptops operate at 30 – 300 MHz, 0.3 – 3 GHz and 3 – 30 GHz, respectively (National Telecommunications and Information Administration, 2006).

1.5.3 Memory

Memory in computing terms is a storage device for data and instructions. Storage devices can be volatile (main memory) or persistent (hard disks, CD-ROMs, and tapes). Data in volatile memory is stored and maintained when the computer system is powered on, but during a system crash, or when the system is powered off, the data is lost. The most

significant performance difference between volatile memory and persistent storage is the type of data access (direct access is approximately 1000 times faster than seek time) and the data read and write time.

1.5.3.1 Hard Disk

A hard disk is a round-shaped, flat circular metal plate that is a magnetic coated device to store data. The unit of storage on a disk is called disk blocks. Disk blocks are contiguous sequences of bytes that are written to or read from the disk. Time to access data from the hard disk is the sum of *seek time*, *rotational delay,* and *data transfer time*. Typical seek time is of the order of 6ms, while average rotational delay time is around 4ms and data transfer time is 1ms. The cost of a hard disk for desktop computers in 2006 is around $0.5/GB (AnandTech 2006).

Figure 1.1 Dell 80 GB Hard Disk (Brain 2006)

1.5.3.2 Main Memory

Main Memory or Random Access Memory (RAM) is an electronic storage device that holds the currently executing program and its working data. When stored data are accessed, the time to identify and get to the memory location or address is called the

access time. RAMs in desktop computers come with access times as low as five nanoseconds. Each address in memory is randomly accessible, thus the access time is relatively constant when compared to the access time of hard disks, where the access time depends on the actual physical location of data on the hard disk and the current position of the read-write head of the disk device. Cost of a memory for desktop computers in 2006 is around $80/GB (AnandTech 2006).

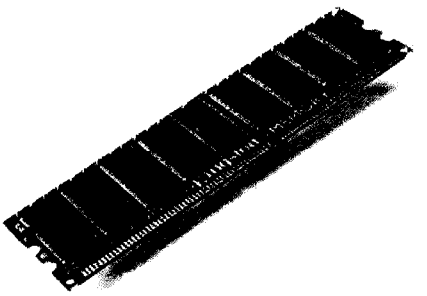


Figure 1.2 Kingston 1 GB RAM (Epinions 2006)

1.5.3.3 Cache

Cache is the fastest accessible portion of memory that is built in the processor, called primary or L1 cache. Sometimes an extended cache is also provided in the motherboard called secondary or L2 cache that is used as buffer.

1.6 Audience

The intended audiences of this thesis are computer scientists or spatial database system developers who are exploring the possibility of designing or implementing an index structure for moving point objects. In a more general sense, this thesis might also interest any geographer, transportation engineer, or city planner.

1.7 Organization of Thesis

This chapter has provided a general outline on location-based applications, and the objectives of this research. Further, a selection of relevant concepts in location positioning technology and computer hardware were introduced. Chapter 2 presents the problem definition. It first introduces location modeling of moving objects, location update protocols, typical queries on moving objects, and arrives at a set of requirements for indexing moving objects. Chapter 3 reviews existing literature with regard to spatial indexing structures for moving objects. Chapter 4 provides an architectural overview of MOD in which we envision the yellow tree index structure to be deployed. Chapter 5 presents the *yellow tree*, a distributed main-memory based spatial index structure, a new approach to indexing moving objects. Chapter 6 evaluates the performance of the yellow tree with a set of experiments to confirm that a distributed main-memory based spatial index structure is required to dynamically adapt to continuously changing moving object traffic and efficiently answer spatial range queries on moving objects. Chapter 7 summarizes the findings as conclusions and suggests future work.

CHAPTER 2

CONTINUOUSLY MOVING OBJECTS IN GEOGRAPHIC SPACE

At their core, location-based applications deal with users moving in geographic space, querying an underlying information system and receiving location-based services at users' requests or users' preferences. This chapter formally defines the problem of indexing moving objects in geographic space. The chapter also introduces different interaction models between a moving object and an MOD with regard to location updates, investigates the types of queries that we are interested in answering, and introduces classical indexing structures. Finally, we derive the requirements for a spatial index structure for efficient handling of the type of queries that are addressed in this problem setting.

2.1 Problem Definition

This section gives a formal definition of the problem of spatially indexing large numbers of moving objects that update their location information concurrently and continuously to a moving object database system. We detail the problem, and model space and moving objects in an MOD environment.

2.1.1 Euclidean space

Euclidean space, also called Cartesian space, is mathematically an n-dimensional space that is represented by the set R^n, $\sqrt{((x_1 - y_1)^2 + (x_2 - y_2)^2)}$ where R is a set of real numbers and n is a non-negative integer (Dubrovin et al 1993). Euclidean space defines a distance function between any two points (x_1, y_1) and (x_2, y_2) as and acts as a container for

all points. We model space as a Euclidean plane region in dimensionality 2 (R^2) with a closed boundary. A minimum bounding box defined by (x_{min}, y_{min}) and (x_{max}, y_{max}) represents the minimum and maximum extents respectively of the underlying space.

Geographic objects contain non-spatial attributes and a spatial attribute, which describes the object's geometry. Geographic space can also be considered as a geographic object itself. In theory, it embeds an infinite number of zero-dimensional points, but to be represented in the database the geographic space will embed only a finite number of points for practical purposes. For simplicity, let geographic space be a 2-dimensional object and its shape be a rectangular box with extent M in x direction and N in y direction, then the area of the object is the product of M and N. We represent each point object in space with a pair of coordinates (x, y), where x and y represent abscissa and ordinate respectively.

Movement of an object in geographic space can be constrained, unconstrained or network movement (Pfoser 2002). Network movement is generated by a network that restricts the movement of objects in a certain direction. Road and rail networks are some examples of networks that constrain object movement. Constrained movement is allowed movement in certain regions of space, like pedestrians in a park, or soldiers in a battlefield. Constrained movement implies regions within space that are inaccessible for moving objects. Unconstrained movement is possibility with regard to movement to any point in geographic space, like vessels in sea and all possible movements happen within defined space with a closed boundary. In this thesis, we consider geographic space as space without constraints and moving objects with all types of movements. Constraints imposed by the underlying space and carrier influence the type of movement that is

allowed for a moving object. For example, pedestrians carrying cell phones follow constrained movement within accessible regions in the park, while postal packages carried by trucks follow network movement. Another concept of interest is time, since properties of objects are expected to change over time. We model time as one-dimensional attribute to a moving object.

2.1.2 Moving Objects

An object is an entity in the real world. We define a moving object as an *object whose location changes continuously over time* (Guting et al 2000). Some examples of moving objects that can be tracked with location sensors are cars, people, pets, or postal packages. Moving objects such as a tornado or a forest fire can also change their shape and location continuously and evolve with time, but these types of features are beyond the scope of this research. A unified framework for spatio-temporal data is presented by Worboys (1994) and Guting et al (2000). We consider moving objects as objects with a well-defined boundary, and abstract object's shape to a point.

Since objects are moving, the location of an object *o* can only be defined at a specific instant of time, t as $o_{(t)} \in R^n$, where n is usually 2. The current location of a moving object is expressed as a function of time, velocity and the heading direction as $o(t_i) = o(t_i)+v*(t_i-t_{i-1})$. An object's movement in space and time is modeled by Miller (1991), and Hornsby and Egenhofer (2002). Miller models movement in space and time as a lifeline thread, which is a linear approximation of a path between any two points in space. Movement has an associated temporal duration and spatial extent. Lifeline model also illustrates that the set of all possible locations that an object could have visited between any two points in space-time traveling at a certain speed is a *bead.* Lifeline

beads represent a coarse granular view of movement, but if additional space-time sample points become available, the bead's geometry can be altered to a series of beads forming a necklace representing a finer view of movement history. Geospatial lifelines are viewed in different granularities depending on the level of detail required and the consequence of shifting among granularities in spatio-temporal knowledge representation as is discussed in Hornsby and Egenhofer (2002).

We model history of moving object in space and time as a lifeline thread, but our research focuses on efficiently managing the current locations of moving objects, thus for indexing purposes, we maintain only the last updated location of every moving object. We also assume that moving objects are equipped with (1) a location sensor that records the object's location and (2) a wireless network connection that enables the object to communicate with a database system. MOD environments allow moving objects to be created, registered, and communicate using a wireless network and maintain a list of its historical states. Moving objects such as cars, trucks, and trains often travel at constrained speeds that their underlying network imposes. There are many types of moving objects that travel at different speeds and varying update frequencies. In this research, we do not restrict ourselves to specific types of moving objects based on speed or underlying network.

In defining a data structure for moving objects, we identified some of the common properties to all types of moving objects such as a unique identifier, speed, position, time, and a direction vector. Unique identifiers distinguish different objects from each other. Cars and trucks have a vehicle identification number, postal packages are designated a tracking number and mobile phones have a phone number that are all unique. Moving

objects have a speed and a heading direction that are dependent on the underlying route. Speed is expressed in miles per hour (mph) and heading direction in *azimuth*, the angle of deviation from a reference direction, usually north (Bowditch 1995). Location attribute of moving object is also associated with an uncertainty measure, since the location value represented in the database is not necessarily accurate.

2.2 Uncertainty

Uncertainty is ambiguity in terms of information about something and the factors about the data that result in uncertainty are inaccuracies, vagueness, incompleteness, inconsistency, and imprecision (Worboys 1998). *Spatial uncertainty* refers to indecisiveness about the location of a moving object. In MOD environments, the location that is represented dynamically as a function of the last updated location, the last updated time, the speed and the heading direction, is predicted and not necessarily exact (Wolfson 1998). The current location being derived implies that a deviation from the actual location can exist, since speed at which the object travels is dependent on the underlying network and presence of other objects in the network. The *linear deviation*, which is the distance between an object's derived location in the database to its actual location, can occur in all directions, thus geometrically the spatial uncertainty in location is a circle with linear deviation as a radius. If the moving object's location between time t_1 and t_2 is defined by end points P_1 and P_2 respectively, the spatial uncertainty across time can be represented as an ellipse with foci at P_1 and P_2 (Pfoser and Jensen 1999).

In another scenario, where objects move in a predefined route, such as delivery trucks or city metro buses, conditions in the route can cause delays resulting in a

deviation from the estimated travel time and the actual travel time, which is referred to as the *temporal uncertainty*.

2.3 Location Update Streams

Moving objects record their spatial locations periodically using location sensors such as GPS. The minimum time interval to compute an object's location is defined by the location device's ability to generate consecutive location updates. The cost of updating a moving object's location to an MOD is, therefore, a function of (1) the location generation cost, (2) the communication cost, and (3) the database system processing cost. The bandwidth of wireless base stations that are Wi-Fi (complying with IEEE 802.11b standard) enabled is 11 Mbps and the network availability for location updates of moving objects is limited by network bandwidth, network traffic, and congestion (Wolfson et al 1998). Although MOD users are interested in the accurate position of moving objects of interest, it is not effective to update the database system at every instance while an object is moving, since each object would generate a very large number of location updates. This would be aggravated if an MOD keeps track of millions of moving objects; however, updating the database too rarely also becomes a problem, since the objects' location data in the database become outdated and therefore, inaccurate. Consequently, a location update protocol between the moving object and the MOD is employed to achieve a feasible update ratio with acceptable location accuracy.

The location update protocol is an agreement between the moving object and the MOD that defines how often and what information is communicated (Trajcevski et al 2002). Moving objects update their new location, time of update and any other parameter

that is being monitored. Location update protocols can be based on either time, distance or both.

2.3.1 Distance-Based Location Update Protocol

A *distance-based* location update protocol requires an update from the moving object to the MOD whenever the moving object deviates from its last updated location by *d* units of distance; therefore, *d* is threshold distance. Distance-based location update protocols help predict the moving objects' locations within a circular region of uncertainty, with *d* being the radius of the circle. Spatial uncertainty in predicted location for moving objects that follow a distance-based location update protocol can be expressed as πd^2 as illustrated in Figure 2.1.

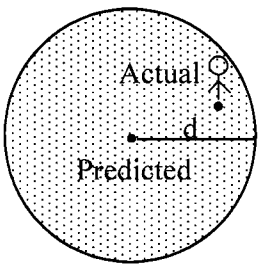


Figure 2.1 Spatial Uncertainty in Distance-Based Location Update Protocol

2.3.2 Time-Based Location Update Protocol

A *time-based* location update protocol requires an allocation update to the server every *t* units of time, where *t* is threshold time. Time-based location update protocols can only predict the moving objects' locations based on the speed and the time elapsed since the last update. If t_l is the last updated time and t_c is current time, then the elapsed time since last location update t_e is $(t_c - t_l)$. For an object traveling with maximum speed of v_{max}, the

maximum deviation from its predicted location is ($t_e * v_{max}$), and the spatial uncertainty is a circle with area $\pi (t_e * v_{max})^2$.

2.3.3 Hybrid Location Update Protocol

A hybrid location update protocol prompts an update to an MOD if the moving object deviates from its last updated location by *d* units of distance or θ angular units from its heading direction, where *d* is the threshold distance and θ is the angular deviation threshold (Gowrisankar and Nittel 2002). This protocol is only applicable with objects moving in constrained networks and on predefined routes. For an object traveling in a predefined route, with a threshold distance *d*, the object is only allowed to deviate by d units of distance in any direction along the route. If an angular threshold of θ is additionally applied, then the object movement is constrained to two directions, either forward or behind, but along the predefined route. Therefore, an object's location can be predicted within twice the distance along the route, and including the width *w* of the route, the spatial uncertainty is a rectangle with *2d* as length and *w* as width.

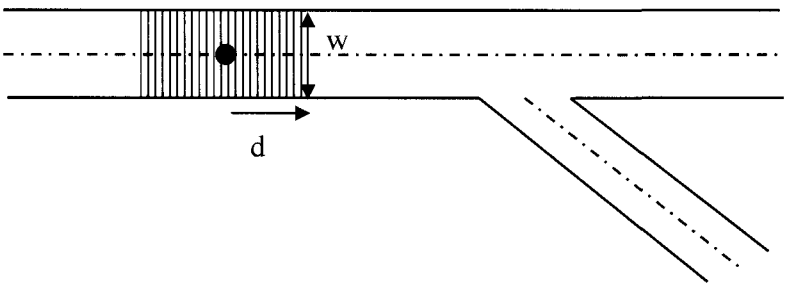


Figure 2.2 Spatial Uncertainty in Hybrid-Location Update Protocol

2.3.4 Time-Distance Location Update Protocol

An effective location update protocol uses both time and distance; such that an update is prompted when the moving object deviates by the threshold distance or by the threshold

time, whichever happens first. By including both parameters, we effectively track the object better based on distance as well as time, since the spatial uncertainty is minimum of distance location update protocol (πd^2) or time location update protocol ($\pi(t_e * v_{max})^2$). Any object that has not updated its location even after the threshold time is labeled inactive, since an idle object does not necessarily mean the object has moved out of the service area, but could have been disconnected from the network.

2.4 Distributed Architecture

Objects participating in an MOD application use a *distributed architecture* in the background. A distributed architecture is a logically connected network with well-defined components: a set of nodes participating in the network, a communication protocol between the nodes, a storage replication policy, and load balancing between the nodes (Sadoski 1997). Clients of an MOD can be static or mobile themselves. Moving objects by virtue of being mobile are bound to be spatially distributed. To manage location updates efficiently from large numbers of spatially distributed objects, it is useful to deploy *directory servers* at vantage points (Denny et al 2003). Each moving object would thus communicate to a local database server or directory server depending on its location. The cell engineering approach used to support cell phones (Layton et al 2006) and spatial discovery services such as Mobiscope (Denny et al 2003) follow a spatially distributed architecture to manage moving objects.

2.5 Queries

Queries allow the assessment of the current state or history of objects in a database. They define or manipulate (*retrieve, insert, delete, and modify*) data. Queries for spatial data have been given considerable attention by commercial DBMSs like Oracle 9i Spatial, and

IBM's DB2 spatial blade. A similar approach has been made to manipulate spatio-temporal data using STSQL (Bohlen et al 1998) and STQL (Erwig and Schneider 1999). The following subsection introduces the relevant queries in an MOD environment.

2.5.1 Object Queries

Object-based queries are queries based on object identifiers. The information queried about the object could be spatial as in "Where is object X?" or non-spatial as in, "Which color is the car with license plate: 914 XEP?"

2.5.2 Spatial Queries

Spatial queries are queries either on the spatial attribute of an object, on objects located within a certain region, or are objects that are co-located. Processing spatial queries involve one or more spatial operators such as intersect, touch, overlap, cross, within, and contain (Egenhofer and Franzosa 1991). The spatial attribute of interest is the current geographic location of an object; however, we assume that the objects of interest are moving, changing their positions continuously, compared to static spatial objects, such as land parcels, dam locations or roads. Queries like, "Retrieve all yellow cabs within a 5 mile radius from the airport," "Find the nearest gas station for object X," "What is traffic volume at 50^{th} and Oak street intersection," and "Find all coffee shops in the mall" involve algorithms like point-in-polygon or polyline-polygon-intersection.

2.5.3 Spatio-Temporal Queries

Spatio-temporal queries are queries with spatial predicates that also have a temporal selection. Spatio-temporal queries like "Did object A meet with object B in the last hour?" and "What is the total number of trucks that have passed through the city

yesterday?" involve processing of spatial relations over the given period of temporal selection. An elaborate study on spatio-historical queries can be found in Griffiths et al (2001). We focus only on spatial and object queries that involve the current location of moving objects as shown in Table 2.1.

Querying Object	Queried Object	Query Type	Yellow Tree Support
Static	Static	Object	No
Static	Static	Spatial	No
Static	Moving	Object	Yes
Static	Moving	Spatial	Yes
Moving	Static	Object	No
Moving	Static	Spatial	No
Moving	Moving	Object	Yes
Moving	Moving	Spatial	Yes

Table 2.1 Query Support in Yellow Tree

2.6 Indexing Moving Objects

Index structures are auxiliary data structures that are aimed at accelerating database transactions, such as insert, delete, update, and query processing by organizing and grouping objects in a specific way (Ramakrishnan and Gehrke 2000). Index structures for spatial data use *space filling curves* (e.g., Hilbert curve) internally to organize multidimensional data in computers to preserve proximity, since data stored as memory blocks have no sense of directionality (Samet 1990). In traditional disk-based database systems, the dominant cost in query processing is the processing time to load or write a data page in memory, commonly referred to as the input/output (I/O) time. Thus, the primary goal of disk-based index structures is to minimize the number of I/O requests.

Trends in increasing main-memory capacity and the possibility of 64 bit addressing in processors have led to commercial systems with *gigabytes* of main-memory (Bernstein et al 1998). In main-memory DBMS, the I/O bottleneck is no longer relevant if the data of interest fit completely into main memory; hence, the primary goal of a main-memory index structure is to reduce the overall computation time while searching for objects with specific attribute values.

For MOD systems, a mix of disk-based and main memory-based techniques are required. Moving objects consist of spatial attributes that are continuously changing such as location and non-spatial attributes that do not change very often such as the type of moving object. Continuously changing attributes of a moving object need to be stored in main-memory to cope with rate of change, while relatively static attributes can be stored in disk memory. However, if an MOD application is developed for the New England region in United States, to track moving objects of interest such as cars, trucks, buses, postal packages, pedestrians, and pets, the moving object load is expected to be around 100 million. If 100 million moving objects update their locations about every 10 minutes, generating approximately 170,000 updates per second to an MOD application, main-memory techniques are required to index the current location of moving objects. We expect directory servers participating in an MOD environment have at least 1 GB of main-memory and allocate 800 MB exclusively for MOD application, the rest being used for operating system and network services. For a directory server to manage moving object entries with location attribute (together occupying about 128 bytes), a dense index of 100 MB allows indexing of 800K moving objects, hence to scale to millions of moving objects we need a network of PCs that coordinate and participate in distributed index

structures with regard to indexing. With increasing trends in the main-memory available to adapt to continuous location updates dynamically, we conclude that storage of moving objects' location attribute values as well as spatial index structures are more effectively handled by main-memory techniques.

Spatial index structures are index structures that are built on keys with spatial attributes, which is the geometry of the object. A spatial index is built primarily to reduce the search time in spatial query processing (Frank 1981). This is achieved by preserving spatial proximity, such that objects co located in space are grouped together in memory as well. Search key values for spatial indexes are only a representation of the actual object, since storing the complete spatial geometry, as a key is infeasible due to the complex nature of geometries. Hence, geometry is typically abstracted to a *minimum bounding box (MBB)* to be represented in the index.

Abstracting an object's geometry into an MBB prompts spatial range query processing to be done in two steps. The first step is to check whether the indexed objects' MBBs intersect with the geometry of the spatial query and arrive at a set of candidate objects. The second step is to check whether the objects' actual geometry intersects with the spatial range, eliminating *false hits* (Brinkhoff et al 1994). False hits are objects whose MBB intersects with the spatial range but their actual geometry does not intersect.

2.7 Index Requirements

We pose the following requirements to an indexing structure.

2.7.1 Adaptability to High-rate Updates

To track moving objects that update their positions continuously, a spatial index structure needs to adapt to high rates of continuous location updates. The adaptability of the index structure includes the insert, update, and delete of moving objects. The index structure also needs to reorganize in near real time.

2.7.2 Scalability

MODs handle very large datasets in terms of spatial extent and numbers of moving objects; therefore, a spatial index structure needs to scale well to fit large spatial extents and increasing numbers of moving objects of different types.

2.7.3 Ability to Handle Batch Loading Of Data

While loading data into the database server, it is desirable to bulk-load the database and then build the index structure, rather than building the index structure while inserting objects. Batch loading of data becomes necessary in the event of a crash of a directory server; therefore, construction of index structures should be independent of the order of insertion of objects (DeWitt et al 1994).

2.7.4 Ability to Handle Skewed Moving Object Update Traffic

Skewed moving object traffic refers to an uneven distribution of moving objects, in a specific geographic area leading to an uneven distribution of updates. An indexing scheme needs be able to handle *spatially skewed traffic* (more updates from a specific area) and *temporally skewed traffic* (that is, a burst of updates at a specific time).

2.7.5 Flexibility to Accommodate Different Location Update Protocols

Often, a moving object database will have to handle different *types* of moving objects like cars, buses, or people that might have different location update protocols, that is, minimal constraints to update their location to an MOD. An indexing scheme needs to be flexible enough to accommodate different types of location update protocols.

2.7.6 High Throughput

Throughput in general refers to the effective amount of work done within a given period of time (Jain 1991). We define throughput of an index structure as the percentage of time that the index structure is available to handle inserts, updates and queries. Throughput is particularly important in an MOD environment since the database server is expected to be highly available to handle millions of objects with location updates rates as low as 30 seconds.

2.7.7 Index Deterioration

As indexes are built on a steady state of data, with a continuously changing data distribution in an MOD, the search performance of an index structure degrades over time. The deterioration is more pronounced in skewed data, and it should be avoided to support search queries efficiently. A performance study of index structures for MOD applications with a focus on index deterioration is presented in Myllymaki and Kaufmann (2003).

2.7.8 Effective Space Utilization

Space utilization refers to the ratio of the total number of objects being indexed to the maximum number of objects that could be indexed. The indexing scheme should be able

to maintain a high space utilization ratio taking spatial skew of input data into consideration.

2.7.9 Garbage Collection

Indexing scheme should be able to cleanup objects that are no longer active to accommodate currently active objects and process queries efficiently.

2.8 Summary

This chapter defined the problem of indexing moving objects. It led to a set of requirements for an index structure, specifically addressing the types of queries that we are interested in supporting. The next chapter reviews spatial index structures and their relevance to indexing moving objects.

CHAPTER 3

SPATIAL INDEX STRUCTURES

With Chapter 2 providing a definition of the problem and arriving at a set of requirements for indexing structures, this chapter surveys relevant research work in indexing moving objects. While evaluating existing approaches for merits and demerits, it is important to consider the problem that each approach was trying to solve. We will analyze spatial index structures for their support in point/spatial range queries, dynamic adaptability, and space utilization. Dynamic adaptability is the ability of the index structure to support insert/delete of objects and gracefully adapt to growth or shrinking of the dataset in near real time. The rest of the chapter surveys spatial index structures on partitioning schemes, memory residence, and temporal support. Since most of the spatial index structures analyzed here are tree based, an overview of hierarchy and tree traversal is also provided.

3.1 Data Order

Index structures can either be built to preserve the natural order of data or they can follow a random order while indexing objects. Order varies with the type of data, for example indexes on text attributes group data alphabetically, while an index on location data is grouped on spatial proximity. Hash-based index structures randomize the order and group data into units of storage, called *buckets*, using a *hashing function* (Lehman and Carey 1986). A hashing function determines the bucket address for an object based on its key so that the time taken to access any object is constant for equality searches. Since hashing randomizes the order of data, it is inefficient for range queries and hence inappropriate for preserving spatial proximity.

3.2 Tree-based Index Structures

Tree-based index structures such as the B tree (Bayer 1971), the Quadtree (Bentley 1979), and the R-tree (Guttman 1984) preserve the natural order of data and follow a hierarchy in indexing data. Tree-based index structures are composed of different types of nodes, such as *root*, *non-leaf,* and *leaf* nodes. The tree hierarchy starts with the root node on top, non-leaf nodes in the middle, and leaf nodes at the bottom. The root node serves as an entry point to the tree structure. Non-leaf nodes contain pointers that point to other non-leaf or leaf nodes. Leaf nodes, which are in the last level of tree hierarchy, contain pointers that point to data. Non-leaf and leaf nodes share a parent-child relationship, where non-leaf nodes create leaf nodes, thus referred to as parent nodes and leaf nodes as child nodes. The process of linking root, non-leaf and leaf nodes with each other through pointers is called redirection. The height of a tree is the number of redirects from the root node to leaf node. Given a set of N objects in the index, each redirect from the root to non-leaf and subsequently to leaf nodes, is aimed at filtering data to a subset of N. For index structures to be efficient in search operations, the subset of data in the leaf nodes should be minimal, while for insert/delete operations the levels of redirection or height of the tree should be minimal. The trees are height-balanced, if all leaf nodes fall in same level, implying a constant time to reach any leaf node in the tree. Height-imbalanced trees have leaf nodes in different levels, implying the time to reach different leaf nodes is variable.

In tree-based spatial index structures, the root node encompasses the entire spatial extent of the dataset. Each leaf node is expected to group a subset of data from the entire dataset, and as the subset of data grows, the search performance of the index structure

becomes inefficient. Split is a technique that partitions data in leaf nodes, by creating new leaf nodes and distributing data among them. A node that is split becomes a non-leaf node, introducing another level of redirection from root to leaf node, thereby increasing the height of the tree. The decision by the index structure to split also involves a trade-off between search and insert/update performance. Splitting often results in too many leaf nodes, each having a small subset of data, thus accelerating some search queries, but also introduces non-leaf nodes between root and leaf nodes, thereby slowing down insert/update performance.

3.3 Partitioning Scheme

Partitioning involves dividing of space or data based on a condition, such that objects could be filtered and grouped together. Two types of index structures are distinguished: those that partition the underlying space and those that partition data. The quadtree family falls under the space partitioning category, while the R-tree (Guttman 1984) and its variants fall under the data partitioning category.

3.3.1 Space-driven Partitioning

Space-driven partitioning divides the underlying 2D space into rectangular cells to achieve balance in data distribution. Since partitioning is independent of data distribution, balance in load may not be achieved with one level of division, thus the space-driven partitioning algorithm continues recursively to balance the load.

The quadtree is a height-imbalanced spatial index structure with a space-driven partitioning scheme. It was originally developed for indexing images in main memory (Bentley 1979). Quadtree traversal follows a top-down approach, from the root to non-

leaf nodes and then to leaf nodes that point to data. The tree traversal has a maximum depth beyond which the tree cannot be partitioned to achieve a balance between insert/update and search query processing performance.

3.3.1.1 Bucket Quadtree

Bucket Quadtree (BQT) is a quadtree that uses a hash table to group data within the leaf nodes (Samet 1990). Each node has a unique identifier (*NodeID*) within the index structure, based on its depth and a spatial extent or bounding box (*BoundingBox*) within which all of the objects reside.

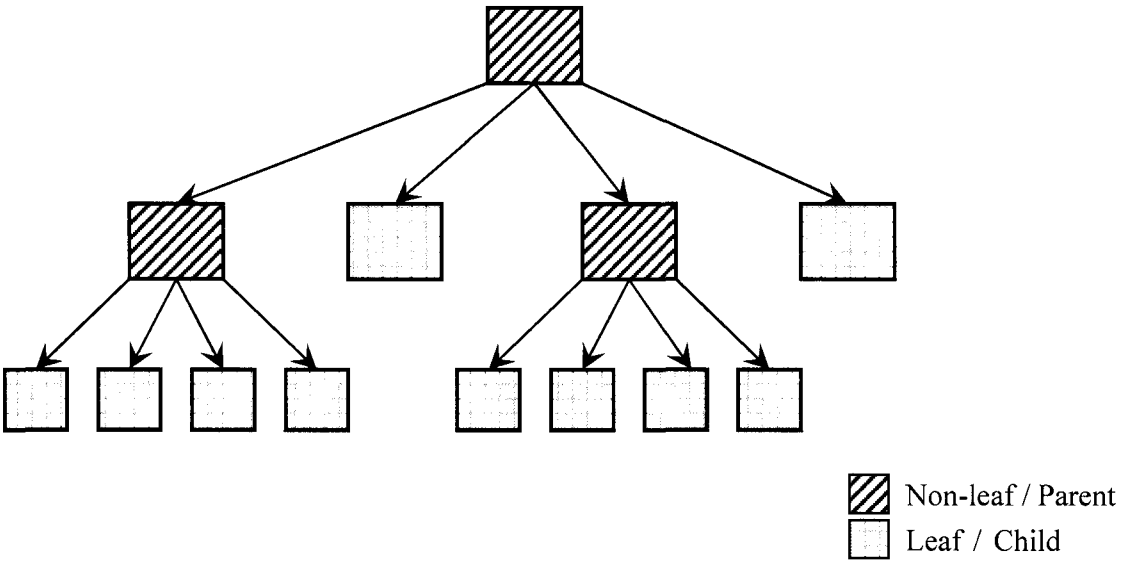


Figure 3.1 Bucket Quadtree

BQT nodes are of the form (*NodeID, BoundingBox, HashTable, and ChildNodesList*). A hash table stores all moving objects within the bounding box and uses moving objects' unique identifier (*MovingObjectID*) as its key. BQT, by using hash tables for storage has an advantage of answering ID based queries with maximum

efficiency. BQT also achieves better memory space utilization, since it groups objects in hash buckets instead of indexing each object into a leaf node. Key requirements of an index structure are its ability to build index, insert new objects, update/delete existing objects, and reorganize.

- **Build Index**

Building index is the process of creating the data structure in memory from the root node to leaf nodes such that every object is accessible through the index structure. The index starts with the root node, and as the object count increases, the root node splits and becomes a parent node. For any object to be inserted, the root node is the entry point to traverse the tree.

- **Insert**

Insert is a function by which an object is added to index structure, such that the index structure maintains a reference to the object. Objects are inserted into the leaf nodes of the tree based on their spatial containment and boundary intersection relationship with the node.

Insert begins with root node and continues routing until an appropriate child node that spatially contains the object is found. Routing is the process of identifying the child node for an object based on spatial containment. In BQT, child nodes' bounding boxes do not overlap, thus each point object shares a containment or boundary intersection relationship with only one child node.

- **Reorganize**

Reorganize is a procedure by which the BQT adapts by splitting or merging dynamically to effectively manage indexing objects. Split and merge achieve load balancing and memory space utilization respectively.

- **Split**

Split is the procedure by which a node partitions its space to form four new nodes, when the object count reaches 90% of node's maximum capacity. The splitting node becomes the *parent* node and the newly formed nodes become the *child* nodes. Since the split procedure follows a space partitioning algorithm, some of newly formed child nodes may be empty, implying a skewed distribution of data with respect to its underlying space. Split procedure continues to partition space recursively until the underlying data is distributed enough to be handled by the newly formed child nodes. If M represents the maximum number of objects that a node's hash table can hold, a split is initiated if the object count exceeds the maximum allowed capacity.

If the spatial extent to be partitioned is defined by the extreme points $B_{(Xmin,\ Ymin)}$ and $B_{(Xmax,\ Ymax)}$, space is partitioned right down the mid point $C_{(x,\ y)}$ such that $C_x = ((B_{Xmin} + B_{Xmax}) / 2)$ and $C_y = ((B_{Ymin} + B_{Ymax}) / 2)$ resulting in four new nodes, one in each SW, SE, NW and NE quadrant. The SW quadrant is defined by the end points $B_{(Xmin,\ Ymin)}$ and $C_{(x,\ y)}$, SE by end points (C_x, B_{Ymin}) and (B_{Xmax}, C_y), NW by end points (B_{Xmin}, C_y) and (C_x, B_{Ymax}), and NE by end points $C_{(x,\ y)}$ and $B_{(Xmax,\ Ymax)}$. If the total spatial extent to be indexed is a rectangle with minimum side length *s* and *if* d_{min} is the minimum distance beyond which two points in space cannot be differentiated, then the length of side after i^{th} split would be $s/2^i$.

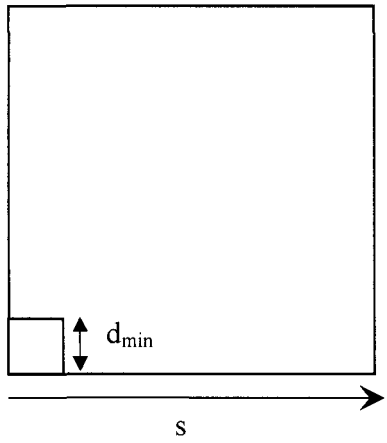


Figure 3.2 Maximum depth of Quadtree

The maximum distance allowed between two points in square B_{min} is given by the diagonal distance = $\sqrt{((s^2/2^i)^2 + (s^2/2^i)^2)}$ that can be simplified to $s\sqrt{2/2^i}$. Hence d_{min} should be less than or equal to (B_{min}), for space to be partitioned further.

$d_{min} <= s\sqrt{2/2^i}$ implies, $i <= \log(s/d_{min}) + ½$.

Thus the maximum depth of a quadtree, h_{max} is reached when i equals $(\log(s/d_{min}) + ½ + 1)$, to account for the root node at level zero.

- **Merge**

Merge is the procedure by which a parent node takes back all moving objects from its child nodes. The merging node becomes the *child* node and its child nodes become empty. Merge is initiated when the sum total of moving objects under all of its child nodes is less than 50% of maximum capacity. Merging is important to main memory index structures like BQT, since it frees up under utilized nodes thus achieving better utilization of memory space. A parent node can only initiate merging, if none of its child nodes is a parent node.

3.3.1.2 Quadtree Variants

Alternate versions of quadtree like Point-Region (PR) Quadtree (Samet 1984), Kinetic PR Quadtree (Winder 2000) have also been used to index point and region data by mapping each point object into a quadrant or mapping region data into a set of quadrants. While indexing moving objects, the size of the dataset could be of the order of millions, so mapping each point into a quadrant would become very expensive in terms of memory space utilization. Kinetic PR quadtree indexes moving point objects and is very effective in determining collision and visibility among objects. Although conceptually Kinetic PR quadtrees are ideal for indexing moving objects, the index structure cannot realistically change for each movement of an object. Continuous reorganization of kinetic PR quadtree makes it impractical to handle large volumes of inserts/updates from moving objects, since most of the time would be spent in reorganizing the index structure leaving very little time to handle inserts and updates.

3.3.2 Data-driven Partitioning

Data-driven partitioning divides the data associated with a node to achieve balance in data distribution. We will focus on data-driven partitioning for objects with spatial attributes. Given a set of N objects, data-driven partitioning is based on incrementally distributing data by increasing dead space and comparing for spatial containment relationship. Dead space is the space in total spatial extent that is not contained in any of the leaf nodes (Guttman 1984). If B_T represents bounding box of the total spatial extent and B_L represents bounding box of leaf node, then dead space d_s is (B_T - Σ $B_{L(i)}$ i = 1 … n).

The R-tree (Guttman 1984) is a height balanced spatial index structure with data-driven partitioning scheme for indexing spatial data. R trees have root node, non-leaf nodes and leaf nodes. R-trees use a minimum bounding box to abstract shape of the spatial object and index entries in leaf nodes have a spatial containment relationship with leaf node's bounding box. Maximum depth of R tree is logarithmic with respect to total number of objects to be indexed, since R tree is equivalent of B+ tree for spatial (multidimensional) data. If m is the number of objects in a node and N is the total number of objects to be indexed, the maximum depth of R tree is at most $[(\log_m N) - 1]$, since root node is considered to be level 0. R trees split to evenly distribute data when $m > M$, using a quadratic split algorithm, which involves identifying two objects called *seed* objects that would maximize the dead space and continue to grow regions around seed objects. R trees are efficient in processing spatial range queries and being height balanced, search and insert/update queries on any object is processed in constant logarithmic time.

Variations of R tree, the R* tree (Beckman 1990), Lazy Update R (LUR) tree (Kwon et al 2002), Bottom-up approach for supporting frequent updates in R trees (Lee et al 2003) were proposed to improve the update performance of R tree. R* tree suggested a simplified split algorithm, where chosen best axis split such that the sum of perimeters of the bounding boxes is minimal instead of a quadratic split. Lazy updates to R trees proposed expanding a leaf node's bounding box such that the object's new location will be contained in the bounding box and delay an update. Bottom up approach to updates provided an alternative to tree traversal from root to leaves, by maintaining a summary hash table with object identifier as key, pointing to leaf node that contains the object. Performance evaluation of main-memory R trees (Hwang et al 2003) concludes

that the Hilbert R tree and its cache conscious version (Kamel and Faloutsos 1994) enhance update performance, but the complex split algorithms are only good enough for static state of data with less number of inserts/updates. Comparing space-driven and data-driven partitioning schemes in the context of indexing moving objects, we arrive at following conclusions.

Space-driven partitioning schemes:

- Employ a deliberately simple split and merge algorithm that makes it more adaptable to continuously changing location data,
- Provide necessary space discrimination that improves filtering on uniformly distributed data for efficient update and search query processing performance, and
- Use recursive split algorithms and overflowing leaf nodes for skewed distribution of data, making it inefficient for update and search queries.

Data-driven partitioning schemes:

- Achieve data distribution with each level of partition for spatially uniform or skewed data,
- Provide necessary space discrimination on uniform or skewed data for efficient update and search query processing performance, and
- Use complex split algorithm that makes it less adaptable for indexing continuously changing location data.

3.4 Traditional Spatio-Temporal Index Structures

Spatio-temporal (ST) applications such as traffic monitoring, natural resources management, and fleet management involve queries on spatial object over a period of time. ST applications require index structures that index and provide necessary data filtering in spatial and temporal dimensions. ST index structures can be classified based on the type of queries supported. ST queries can be either *historical* that query the past positions, *now* that query the current position or *futuristic* that query the predicted future positions of moving objects (Mokbel et al 2003). Historical ST queries require indexing the past positions of moving objects, while now ST queries require indexing the current positions. Futuristic ST queries involve indexing the current and future position (predicted) and are based on parameters such as speed, heading direction and underlying network of moving object. Index structures such as quadtrees for moving object trajectories (Tayeb et al 1998), Parametric space indexing (Porkaew et al 2001), Spatio-temporal self adjusting R (STAR) tree (Procopiuc et al 2002), time parameterized R (TPR) trees (Saltenis et al 2000), and TPR* tree (Tao et al 2003) were proposed to index current and future positions and are beyond the scope of our research.

3.5 Indexing Moving Object Trajectories

Three approaches to indexing spatio-temporal data are identified in (Mokbel at al 2003). First approach is to treat time as an extra dimension and build one multidimensional spatial index structure. RT tree (Xu et al 1990), 3D R tree (Theodoridis et al 1996) and STR tree (Pfoser et al 2000) fall under this category, where space and time are combined in one spatial index structure providing no temporal discrimination for ST queries. RT tree introduces a start time and end time for leaf nodes in R tree. 3D R tree adds time as

third dimension in R tree and STR tree introduces a *trajectory preservation* parameter, which is the number of states of moving object stored together to achieve a balance between maintaining spatial proximity and temporal closeness. This approach answers spatial range queries efficiently, but does not work well for time interval queries.

Second approach is to deal with spatial and temporal dimensions separately to provide space and time discrimination for ST queries. Indexing temporal objects based on versioning was attempted in time split B (TSB) tree (Lomet and Salzberg 1989), relational interval (RI) trees (Kriegel et al 2002) and Multi-Version B trees (Becker at al 1993) but they were not suitable for ST data. Index structures such as MR tree (Xu et al 1990), HR tree (Nascimento et al 1998), HR+ tree (Tao and Papadias 2001), and MV3R tree (Tao and Papadias 2001) where the goal is to keep all spatial data together using an R tree built over many time ranges, provide necessary space and time discrimination. MR tree builds an R tree for each time stamp and employs an overlapping B tree (Burton et al 1990) to manage time slice queries. HR tree and HR+ trees also employ an overlapping B tree and R tree, but objects in leaf nodes of R trees are allowed different time stamps to avoid excessive storage. MV3R tree builds a MVR tree to process time slice queries and 3D R tree for long time range queries. This approach works well for spatial and time range queries, but has high storage requirements. Specialized data structures based on versioning of data, grouped as multi-version index structures have been proposed to index temporal objects. Indexes developed for temporal objects primarily focused on indexing time dependent attribute data. Partially persistent index structures persistently store the past states of the object, but allow updates only to the current state of the object (Kollios et al 2001).

Third approach to handling temporally changing attribute data was attempted in trajectory-bundle (TB) tree (Pfoser et al 2000), SETI (Chakka et al 2003), and start/end timestamp B (SEB) tree (Song and Roussopoulos 2003) where updates belonging to the same trajectory are stored in the same node irrespective of their spatial location. This approach gives up space discrimination for trajectory preservation and is only efficient for time range queries.

3.6 Indexing Moving Object Positions

To answer now spatio-temporal (ST) queries efficiently, indexing on current moving object positions becomes necessary. Index structures such as 2+3 R tree (Nascimento et al 1999), 2-3 TR tree (Abdelguerfi et al 2002), lazy update R (LUR) tree (Kwon et al 2002), bottom up updates in R trees (Lee et al 2003), Q+R tree (Xia and Prabhakar 2003) and hashing moving objects (Song and Roussopoulos 2001) were developed to handle continuous updates and index the current positions of moving objects. The 2+3 R tree employs two R trees, one for 2D current data and another for 3D historical ST trajectories, whereas the 2-3 TR tree employs a TB tree for trajectory queries and an R tree for 2D current data. Lazy update R Tree stores only current positions and no historical data, as objects update their location, the old entry is deleted or the MBR is adjusted to accommodate the new location of moving object thus delay updating the index. Bottom up updates to R trees avoids the top-down tree traversal by maintaining a main-memory hash table that stores associated leaf nodes of each moving object. The Q+R tree uses a synergistic combination of main-memory quadtree for current updates on moving objects and builds an R tree for historical ST data. Hashing moving objects partitions the underlying space into fixed rectangles or zones that may overlap and delays

updates to the database until a moving object leaves the zone. This approach of indexing current position works well to answer now ST queries, but delaying updates with overlapping bounding boxes to leaf nodes leads to multiple paths for search queries, thus reducing efficiency.

3.7 Summary

Review of relevant literature has identified some common issues to be handled in designing an index scheme for moving objects. Clearly, there is not one index structure that is efficient for insert/update, delete, object, and spatio-temporal range queries. The index structures that have been proposed also did not explore the possibility of sharing load across nodes in a network like in the case of distributed architecture and an even lesser number have addressed dynamic allocation of resources. The identified missing links in all these index structures gives us a better understanding on the new approaches that we are going to experiment with the yellow tree.

CHAPTER 4

A DISTRIBUTED MOVING OBJECT DATABASE ENVIRONMENT

Chapter 3 explained the importance of index structures and addressed some of the issues in designing an index structure for moving objects. Before exploring the yellow tree index structure, we provide a general overview of a distributed architecture and its services to support MOD applications. This chapter also introduces different computing systems available in the context of a moving object database environment.

4.1 High-Performance Computing

High-performance computing uses large numbers of computers that are connected virtually to accomplish a task (Foster and Kesselman 1996). MOD applications involve database operations, whose performance can be enhanced by employing a network of computers because of the enormity of data to be stored and complexity of computations to be processed.

4.1.1 Parallel Computing

Parallel computing systems use multiple processors that are interconnected and work simultaneously to solve a problem (Devitt and Gray 1991). Parallel computing works by dividing a problem into independent sub tasks and dedicating a processor to each subtask. Teradata (Teradata 2006) and Bubba (Boral et al 1990) are systems that use parallelism in database management.

4.1.2 Distributed Computing

Distributed computing uses a network of independent computers to solve a problem using parallelism (Attiya and Welch 2004). Each computer that participates in the network is called a node. Nodes are well distributed geographically and are open to participate in other tasks, thus making distributed computing systems open and scalable. Climate prediction (Climate Prediction 2006), Digipede (Digipede 2006) and Parabon (Parabon 2006) are some examples of systems that have implemented distributed computing.

4.1.3 Cluster Computing

Cluster computing is a type of distributed computing that uses a tightly coupled network of computers to complete a group task (Abbas 2003). The nodes in the cluster are exclusively used for the group task and are not open to participate in other networks. Scyld Beowulf clusters (Scyld Beowulf 2006) and Secondlife (SecondLife 2006) are some examples of cluster computing systems.

4.1.4 Grid Computing

Grid computing uses a geographically distributed network of independent computers dynamically at run time to accomplish a task (Abbas 2003). Grid computing relies on using a network like the internet as a computer. A unique aspect of grid computing systems is their ability to utilize unused processing capabilities of its participating nodes dynamically. Some of the commercially available grid computing systems include DataSynapse GridServer (DataSynapse 2006), IBM Grid Computing (IBM 2006), and Oracle Grid (Oracle 2006).

4.1.5 Peer-to-Peer

Peer-to-Peer (P2P) systems use a decentralized network of nodes called *peers*, in which each peer is a client as well as a server. File sharing systems such as Napster (Napster 2006), Gnutella (Gnutella 2006) and Kazaa (Kazaa 2006) adopt P2P architecture. P2P systems share data and network bandwidth; location-based P2P systems aim to share local context information between peers. Querying for a data item in P2P involves locating peers or nodes that contain the data item. Systems like Chord (Stoica et al 2001) use a distributed hash table to locate data items with peer addresses. Efficient distribution of data to achieve load balancing, dynamically adapting to online and offline nodes, and decentralization of data to avoid single point of failure are some of the challenges in P2P systems (Ganesan et al 2004).

Moving objects tend to be spatially distributed and, therefore, we envision MOD environments to follow a distributed approach, but the yellow tree's indexing component was developed independent of the underlying architecture.

4.2 Distributed MOD Environment

We expect MOD environments to follow a distributed architecture, since distribution provides necessary decentralization of data for moving objects. In contrast to peers in P2P, a large percentage of users in MODs just want to query the environment rather than share data and network bandwidth and be a part of the network. Hence, we foresee an MOD environment to be distributed using powerful base stations and offering services to register, query and manage moving objects, such as *Root Servers* (RS), *Directory Servers* (DS), and the *Query Clients* (QC).

4.3 Root Server

A *Root Server* (RS) in an MOD acts as a registration server to monitor individual directory nodes (or servers) and serves as an entry point, metadata placeholder, and directory node locator. A directory server manages a set of moving objects in a well-defined region of the overall observation area. Multiple directory servers cover the entire study area. Each DS manages a set of moving objects within a small portion of the study area. In order to scale to large numbers of moving objects, directory nodes rely on the RS for information about availability of underutilized nodes to perform load balancing. The communication protocol between the RS and directory nodes ensure that the RS remains up to date about changes in the spatial extent (represented as minimum bounding box) of the directory nodes throughout their lifetimes. The RS uses the metadata information about spatial extents (represented as minimum bounding box) of participating directory nodes to route spatial range queries. A typical RS implements a node information listener to communicate with the directory nodes and a query router that interfaces with query clients is shown in Figure 4.1.

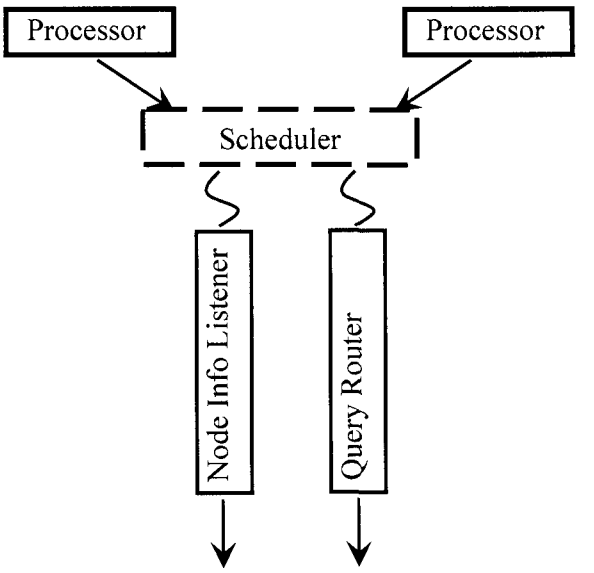


Figure 4.1 Root Server

4.4 Directory Servers

Directory Servers (DS) in MODs are data management nodes primarily concerned with storing and indexing of moving objects. DS nodes typically start their active life cycle by registering with the RS, signaling to accept location updates from local moving objects. Each DS manages a section of data and network load by limiting itself to a defined local spatial extent. Space partitioning in MODs allows for overlapping spatial extents as in a shared architecture or non-overlapping as in a shared-nothing architecture. Systems such as Google (Google 2006) and Terradata (Terradata 2006) use share-nothing distributed data management. In some cases, DS nodes keep track of their sibling nodes and in almost all cases, they maintain metadata about their children nodes. A complete explanation of our version of data management node, the yellow tree node, is given in chapter 5.

4.5 Query Client

Query clients are the users in MODs that pose queries about moving objects. Examples of query clients are a mobile computer, a personal digital assistant (PDA), a mobile phone, or a blackberry device that can connect to the distributed environment at will. Typical queries are spatial range and object-based queries. Query clients pose queries to the RS or directly to the DS nodes to eliminate a single point of entry for queries and achieve better system robustness by distributing the network traffic. When clients direct spatial range queries at the RS, it forwards the query to appropriate DS nodes whose spatial extents intersect with the spatial region that is part of the spatial query predicate. The query processing environment is independent of the type of network to accommodate different types of query clients.

4.6 Query Router

The query router is a service implemented by an RS and DSs to act as a gateway for all types of spatial range queries. The query router service in the RS can route a spatial range query to the appropriate directory servers in the network. The query router in a DS node merely redirects a query to its child nodes based on the intersection with the query predicate. Routing spatial range queries only to intersected DS nodes achieves distributed processing.

Algorithm QueryRouting (Q : Query predicate)

```
begin
   if (Q is spatial) then
      for each (Node N in NodeLookupTable)
         begin
            if (N.mbb intersects Q.mbb) then
```

```
                ROUTE (Q, N) // Communicates to node N, a query
                             request with Q as query
                             predicate.
            end if
        end
    end if
end
```

4.7 Summary

This chapter provided an overview of a typical MOD environment and its services. The communication protocol between the query client, the RS, and the DS nodes have been devised to be independent of the type of network connection to achieve a flexible test environment. The structure of the yellow tree node including the spatial index employed to manage moving objects is discussed in the following chapter.

CHAPTER 5

ANATOMY OF THE YELLOW TREE

Following the overview of distributed MOD environment in Chapter 4, this chapter details the anatomy of the *yellow tree* (YT). The YT services, algorithms in load balancing, and main-memory strategies applied to handle continuous location updates from moving objects are studied. This chapter also details the structure of a typical moving object handled in a yellow server (YS) environment.

5.1 Moving Object

A moving object in the YT is an object whose location changes continuously with time. The spatial attribute of a moving object is a point geometry, defined by (x, y) values represented in a Cartesian coordinate system. The YT supports different location update protocols for moving objects and, therefore, only the last updated location is maintained and indexed in main memory. The current location is derived using a dynamic function of last updated location, last updated time, speed, and heading direction of moving object. Moving objects can also choose not to maintain a location update protocol, in which case the last updated location is the current location. In the YT, moving objects of interest are cars, trucks, pedestrians, and postal packages transported through land whose traveling speeds range between 0 to 80 miles per hour.

5.2 Yellow Tree

The yellow tree (YT) spatial index structure is the most important component of the yellow tree environment, which is responsible for organizing, storing, and query processing of moving object data. At the core, the YT implements the following services:

the data manager and the query processor. The data manager is responsible for insert, update and delete of moving objects. The query processor accepts moving object queries, and uses the YT to answer them efficiently. The YT consists of nodes that have a unique identifier within the yellow tree environment during the nodes' entire lifetimes and are marked by start/end times. YT nodes are created on demand based on the current load in the system. Internally, they all employ a main memory based spatial index structure to index the location updates of moving objects. Each node is responsible for a subset of moving objects that is managed by the YT. To ensure a balanced moving object load and an efficient utilization of main memory, a limit on the maximum and minimum number of moving objects per YT node is enforced. The challenging aspect in the design of the YT is the strategy with which YT nodes cooperate to (1) balance the load amongst each other, and (2) index all moving objects and participate in efficient query answering.

The YT deploys a hierarchical scheme for load balancing. If the number of location updates to a YT node increases, then the node splits into a set of sub nodes to distribute the original load. The split node becomes a parent node and is *passive* with regard to updates and query processing of moving objects. Parent nodes process queries by redirecting them to appropriate active child nodes, and communicate with their child nodes to manage load. The newly formed sub nodes are called child nodes. Child nodes register with the RS directly and are *active* in accepting location updates and query processing. Active child nodes that belong to the same parent are sibling nodes. If the total load managed by the siblings can be managed by the parent node itself, the parent node calls back all moving object load from its children. Active child nodes that have just submitted all moving object load to its parent enter into a *passive* state. In the YT, child

nodes continue to remain passive for some time so if a child node decides to split due to dynamic increase in load, the passive child nodes will be listening and node creation need not be done over the network again, saving time and network traffic. Parent nodes have a time-out parameter to decide on lifetime of child node in passive state.

The YT spatial index structure is made up of a set of YT nodes. Each YT node internally employs a main memory-based variant of the *bucket quadtree* (BQT section 3.3.1.1) to perform spatial indexing of moving objects. The YT supports a top-down approach in tree traversal for spatial range queries as well as a direct approach for object queries. In addition to the BQT, each YT node also maintains a summary hash table of all child nodes that are stored in the BQT, but instead of spatially indexing these objects they are indexed by the object identifier directly. The hash table maintained in each BQT leaf node is particularly important for incoming location updates, which are based on the object ID. The hash table serves as a lookup table for the object and its last stored location. In order to update the old location in the spatial index structure, the old entry has to be removed or updated, and inserted in the new correct BQT node.

5.3 Data Manager

The data manager service in the YT handles inserting, updating, organizing, and garbage collecting of moving objects.

5.3.1 Data Listener

The data listener is the gateway for moving objects to update their location information to the YT. A data listener can be in one of the three states: *active, idle,* and *destroy.* An active data listener gets all the location updates, which are subsequently redirected to the

BQT contained in the YT node. The data listener turns to an idle state during the YT reorganization to indicate that the YT node is not accepting location updates. During idle time, location updates are queued up; afterwards they are routed to the appropriate child node if the YT node splits during reorganization. The data listener operates in idle state to add flexibility and robustness to the YT, since re-routing ensures that location updates are not lost. The destroy state is the state when the data listener stops to listen to location updates and queries.

5.3.2 Reorganize

Reorganizing is the process by which YT nodes adapt to dynamically changing moving object update load through splitting into new nodes or merging existing nodes with low load into a well-balanced node. The splitting and merging algorithms in the YT node are based on a static space partitioning scheme, similar to BQT, with variations to handle the distributed nature of the YT.

5.3.2.1 Split

A YT node splits when the total number of moving objects exceeds 90% of the maximum allowed capacity to achieve better load balancing on YT nodes as well as balance network traffic. The 90% cutoff is chosen to make maximum use of main-memory and provide the child node enough time to split, since during the split process, the remaining 10% of allocated main-memory is used to queue location updates from moving objects. To scale to large numbers of moving objects, a novel feature in design of YT nodes supports the creation of child nodes on remote machines, because each machine has limited main memory and YT nodes need to have access to a large portion of the main memory to accommodate moving objects. Thus, new nodes have to be created on remote

machines with available main memory. The YT node split process starts with memory allocation for four new child nodes to be created and communication with RS to allocate more resources in remote machines if needed. Dynamic allocation of resources in a distributed network ensures that the YT will never run out of resources as long as there is a remote machine willing to share the load. Parent nodes push all moving objects to their child nodes and occupy only minimum space in memory.

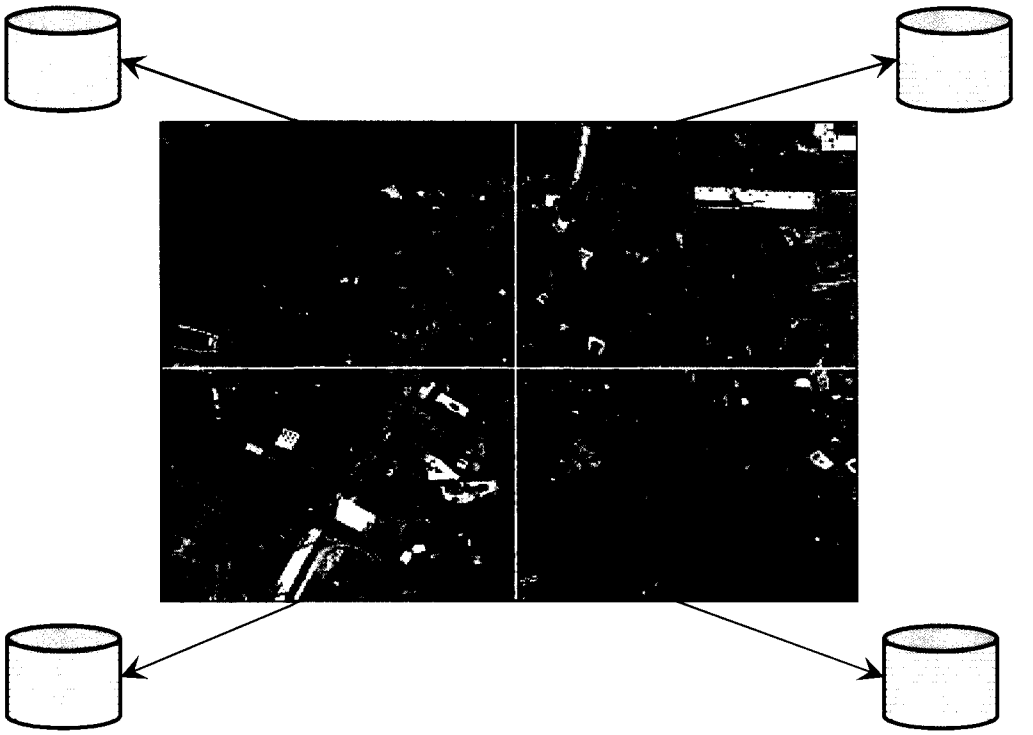


Figure 5.1 YT Reorganization by Split

In principle, the original parent node is split into four sub nodes similar to the bucket quadtree. Thus, the original region of indexing is split into four equal-sized sub regions and a new YT node for each of those areas is created. Each child node is initialized and populated with all the moving objects of a sub region of the original node.

Algorithm Split (N : Node)

```
begin
   NodeInfoList ← LocalHost.FreeNodesAvailable
   // Check for resources to create free nodes in local
   host, which is the physical machine where Node N resides.
   if (NodeInfoList.Count < 4) then
      requestNodeCount ← (4 - NodeInfoList.Count)
      NodeInfoList+= RequestRootServer(requestNodeCount)
      // requests root server for additional resources to
      create new nodes.
   end if
   for each (NodeInfo I in NodeInfoList)
   begin
      ChildNode C ← YTNode (I)
   end
   Route (N.MovingObjects)
   // Routes moving objects to newly formed child nodes
   where they are inserted.
   N.Status ← Parent // Child node becomes a parent
end
```

5.3.2.2 Merge

Merge is an operation initiated by a parent YT node when the sum total of moving objects in all of its child nodes is less than 50% of the allowed maximum capacity of the parent node. The minimum memory usage of 50% is chosen to prevent under-utilization of allocated memory in child nodes. Merge achieves better space utilization in memory, computing power and underlines the adaptability of the index structure to dynamically changing load. Merge ensures that not all sibling nodes under a parent are under utilized. Since some of the child nodes could be remote, the merge may not happen instantaneously when the total moving object count is less than 50%, rather the parent

node has to rely on the child node to communicate its moving object count and then decide on merging, and thus the merge is a *relaxed merge*.

Algorithm Merge (N : Node)

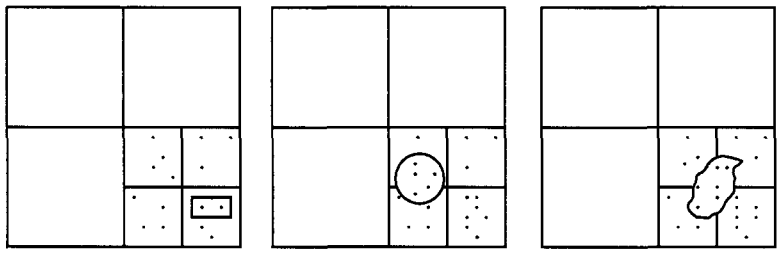
```
begin
   for each (Child C in ChildNodesList)
      begin
         N.MovingObjects += GetMovingObjects (C)
         // gets moving objects from child C to parent N
         ClearMovingObjects (C)
         // Clears memory references to moving objects.
      end
      N.Status ← Child // Node N becomes a child again.
   end
end
```

5.3.3 Garbage Collection

Garbage collection (*GC*) is a mechanism by which the YT cleans its idle moving objects periodically from main-memory. *Idle* refers to moving objects that have not updated their location according to their location update protocol or objects that move out of their area and start updating a new YT node. The YT periodically writes the current state of idle moving objects to a log file. To balance the tradeoff between frequency of garbage collection and available main-memory, YT nodes time garbage collection when the main memory usage on index structure exceeds a threshold percentage of the maximum allocated memory for the node. The threshold is configurable by the user that administers the YT performance. Garbage collection reduces the risk of nodes running out of main memory.

5.4 Query Processor

This section details two types of queries typically supported in the YT: (1) spatial and (2) object-based (section 2.5). The query processor uses index structures and an optimal query execution plan to process both types of queries. Spatial queries supported in the YT are based on the spatial containment relationship with the query predicate or shape. The query shape could be a box, a circle or a polygon as shown in Figure 5.2.



(a) Box (b) Circle (c) Polygon

Figure 5.2 Query Predicates

5.4.1 Spatial Query Processing

Spatial query processing of YT nodes involves two major steps. First, the query-processing algorithm finds child nodes in the spatial index whose bounding box intersects with the query predicate geometry. Since the YT is hierarchically structured and height-imbalanced, a top-down, depth-first recursive tree traversal is adopted. Second, moving objects in such intersected child nodes are candidates to share a containment relationship with the query shape. Every moving object that is contained within the query shape is then added to the query results.

Algorithm: ExecuteSpatialQuery (Q : Query predicate) : List(MovingObjects)

```
begin
```

```
   Node N ← Root // start from root node and continue with
                 top down approach.
      while (N.Status is parent)
           ChildNode C = GetChildNode (N)
           // recursively looks for child nodes
           if (C.mbb intersects Q.mbb) then
                for each (MovingObject M in C.MovingObjects)
                     if (Q.geometry Contains M) then
                          List += M
                     end if
                end
           end if
      end while
end
```

5.4.2 Object Query Processing

Object query processing involves identifying the child node in the index structure that contains the object and getting the latest version of the object. The YT maintains a summary table of all child nodes in the index structure. Object query processing involves searching for the object in existing child nodes' hash tables. MODs use hash tables for equality searches, since the search time is constant. Bottom-up approaches also employ a summary hash table that maps each object identifier with a child node that contains the object. A shared-nothing space partitioning scheme implies that each moving object resides in only one child node.

Algorithm: ExecuteObjectQuery (O : Object ID) : Moving Object, M

```
begin
     for each (Child Node C in ChildNodesTable)
          if(C contains object key O) then
```

```
                M ← C.GetCurrent (O)
                // gets the current version object with O as
                                object key.
            end for
        end if
        end
end
```

5.5 Handshake Protocols to Handle Cooperation between YT Components

The YT nodes communicate with each other and with the RS to handle the load balancing and overall strategy of indexing all moving objects in the system. Each communication is comparable to a handshake. The YT node handshake protocol determines timing and information communicated between any two YT nodes. The *child node to parent* handshake protocol forces a child node to report its status and moving object count, which helps the parent node in monitoring all of its child nodes and decide on merging as needed. The *YT node and RS handshake protocol* enforces all YT nodes to register with the RS to become a part of the YT environment, report their status, metadata, and connection details to the RS. The query router in the RS uses the node's spatial extent to route appropriate spatial range queries. The YT nodes also request the RS for additional resources while splitting as needed to create child nodes in remote machines.

5.6 Crash Recovery

The YT employs a logging mechanism during garbage collection to log the current state of moving objects. Data logging is time stamped and temporally distributed, such that a new log is created for each garbage collection cycle. In the event of a system crash, the

latest time stamped log file is loaded in to memory with minimal loss of data and system availability.

5.7 Summary

This chapter elaborated on the data structure, data management, and query processing capabilities of the YT. A main-memory BQT was employed as a spatial index structure to reduce the complexity and provide dynamic adaptability to changing load. Active load balancing and effective utilization of main-memory was demonstrated through split and merge operations, respectively. The chapter also outlined handshake protocols between various nodes in the YT environment. Merging and garbage collection algorithms demonstrate efficient utilization of main-memory; while re-routing and retaining passive child nodes signify the robustness and flexibility in the YT.

CHAPTER 6

PERFORMANCE TESTS AND RESULTS

This chapter performs a practical evaluation of the YT system and provides experimental results derived from testing the simulated MOD environment. We discuss the validity of our hypothesis after analyzing experimental results. We begin by analyzing the YT algorithms for space and time complexity to provide a better understanding on the logic behind the design of the YT index structure. The chapter also details about different types of simulated moving object load generated to update the YT.

6.1 Algorithm Complexity

Algorithms are abstractions of a program or a sequence of steps used to solve a problem. Algorithm complexity is a generic way to evaluate efficiency of an algorithm, since there is a direct negative correlation between an algorithm's complexity and its efficiency (Cormen et al 1990). Design of algorithms typically involves three major steps, namely the designing of a computation model, a pseudo code or language to express the algorithm, and performance evaluation. We use a *random access machine* (RAM) computation model for the YT, where time to access each block of memory is assumed constant. Arithmetic and comparison operators are used to model the algorithm and pseudo-language is used to explain the algorithm in steps, using loops and conditional statements. We measure the performance of an algorithm as the total time involved in executing all processing operations of the algorithm, neglecting data transfer time across the network, since it is dependent on hardware. Of all the parameters involved in performance evaluation, input size is the most important parameter that affects the time

complexity of the algorithm, since the time to execute an algorithm increases with input size. We evaluate the performance of YT for its worst-case complexity.

6.1.1 Worst-Case Complexity

In the worst-case complexity, the total number of operations or running time of the algorithm for any given input size will be less than or in some cases equal to the *upper bound* (Skiena 1997). Upper bound is the maximum number of operations that are executed by the algorithm to solve the problem with a constant input size.

6.2 Time Complexity

This section analyses the YT's spatial indexing scheme, BQT with its variations for time complexity and presents the results for worst-case. Section 6.5 presents the values from experimental results on BQT operations.

6.2.1 Insert

Insert assumes that the moving object is entered into BQT index for the first time. For a moving object to be inserted based on its location into a height imbalanced BQT, the worst case is to traverse the maximum height of BQT. Since the leaf node of BQT has a hash table to store moving objects, inserting a moving object is executed in constant time.

6.2.2 Update

Updating a moving object is done in two steps. First, the object is inserted into the BQT and then the leaf node is checked for any older versions of the same object. Checking for previous instances of an object in hash bucket involves searching through the list of overflow buckets.

6.2.3 Delete

Delete operation on a moving object, involves locating the object based on its ID. In a BQT where the leaf nodes store moving objects in hash tables, the worst case in locating an object based on its ID is of the order of total number of child nodes. Once the appropriate child node that stores the moving object is identified, deleting a moving object takes constant time.

6.2.4 Space Partitioning

In BQT, since the space always splits down the middle into four new quadrants, irrespective of the distribution of moving objects, the worst-case space partitioning is executed in constant time.

6.2.5 Object Queries

Object queries involve identifying an object based on its identifier. The worst case for an object query is to search all child nodes, and retrieve the object from the child node's hash table. Search is of the order of total number of child nodes and retrieving the object from hash table is executed in constant time.

6.2.6 Spatial Range Queries

Spatial range queries involve identification of all the child nodes that intersect with the bounding box of the query predicate, which could be a box, a circle or a polygon. The worst case for a spatial range query could be intersection of all the child nodes, which is of the order of total number of child nodes.

If N is the total number of moving objects, S is the length of the smaller side of spatial extent, d_{min} is the minimum distance allowed between two moving objects to be

placed in the same leaf node of BQT, then H_t, the maximum allowed theoretical height for BQT, is $(\log(s/d_{min}) + 1.5)$ as derived in section 3.3.1.1.4. The maximum number of child nodes C_{max} for a tree structure that splits into four new nodes at each level is $4^{(H-1)}$.

For the implementation of the BQT another parameter of importance is H_u, the user defined maximum height, which is set as six (including the root node). Let us assume B_o as the number of overflow buckets. The following table gives an account of the worst-case complexity and its impact on the number of comparisons BQT algorithms use to perform various operations.

If S, the length of the smaller side of spatial extent is 500 miles (2640000 feet) and d_{min}, the minimum distance allowed between two moving objects to be placed in the same leaf node is 100 feet, and the user defined maximum height for BQT, H_u is 6, then Table 6.1 lists the worst case complexity and the number of comparisons required to execute the algorithm.

Algorithm	**Input**	**Worst Case Complexity**	**Theoretical comparisons**	**Practical comparisons**
Insert	N	$O(H_t)$	15	6
Update	N	$O(H_t) + O(B_o)$	15	6
Delete	N	$O(C_{max})$	268435456	1024
Space Partitioning	N	$O(1)$	0	0
Object Queries	N	$O(C_{max})$	268435456	1024
Spatial Range Queries	N	$O(C_{max})$	268435456	1024

Table 6.1 Algorithm complexity of YT operations

6.3 System Tests and Simulation environment

The YT was developed using a hardware environment that could support main-memory execution of applications and network communication through sockets. We used seven Dell Optiplex GX260 PCs, each equipped with a 2.8 GHz Intel Pentium IV processor, 8KB L1 (data cache), 512KB L2 Cache, and 1 GB DDR RAM (266 MHz). Dell Latitude laptops were used to simulate query client and root server. The YT nodes in the network, query client and root server were connected by a Netgear GS508T GigaSwitch. Two Optiplex machines were configured as load generators and five machines were running as YT nodes.

The YT was programmed in Java, which is an object-oriented programming language that offers standard *application programming interfaces* (API) for main-memory data structures like the arrays, lists and hash tables (Java 2006). Java also provides a graphics API that implements spatial algorithms such as point-in-polygon and spatial intersection. Sockets are used to communicate between YT nodes and the query client on a different machine. The YT is main-memory based, so no portions of hard disk were used in paging to memory.

The Java programs run under a Java virtual machine (JVM) an application that operates on top of existing operating systems. JVM's heap and garbage collection options need to be tweaked to achieve better utilization of main-memory and throughput (Java Tuning 2006). Java was chosen to develop the YT for its support for multithreading and the ease of use in communicating objects between nodes in the network. Main-memory objects are converted into serialized objects before being communicated across nodes. All components described in Chapter 5 were implemented in the prototype. The YT

maintains threads such as data listener, node info listener, and query processor in a single instance. Threads in java run under different priorities, but we have set the default priority for each thread and this allows the JVM to choose the best thread to run at a particular instant to support concurrency.

6.4 Data Generator

The moving object data generator generates moving objects and communicates location updates to corresponding YT nodes. The spatial extent within which objects were allowed to move was set as a rectangular area (0, 0) to (1000, 500) in Cartesian coordinate system. Total number of moving objects that were actively sending location updates varied from 10K to 1M. To simulate a valid test scenario, it was important to understand typical behavior of moving objects. Theodoridis and Nascimento (2000), Saglio and Moreira (2001), Brinkhoff (2000), and Brinkhoff (2002) have proposed to generate spatio-temporal moving objects that move in a free and constrained network taking into consideration external impacts, minimum and maximum speed and underlying network edge's maximum capacity. Some of the factors we considered while generating moving objects are its location update protocol, maximum speed, and distribution of data.

6.4.1 Maximum Speed

Moving objects typically have a maximum speed with which they travel. This is typical of objects moving in constrained networks such as the road network that impose a speed limit on vehicles. Maximum speed does not depend only on the underlying network, but also on the moving object itself, therefore, we assume that the moving objects themselves know a maximum speed with which they could travel. We have set the minimum speed

of moving objects to be zero, since the object can be stationary for some period of time, but continue to update its location according to some predefined location update protocol.

6.4.2 Data Distribution

Data distribution refers to the relative position of a moving object with other objects within the given spatial extent. Distribution of other objects in spatial proximity can affect the speed of moving objects. For instance, during peak hour traffic, the average travel time could be high because of congestion. In addition, there could be an uneven distribution of data when there is some special event like a baseball game that attracts many objects. Our data generator accounts for two kinds of skew, spatial and temporal. Spatial skew is uneven distribution of moving objects in an area. Temporal skew refers to the uneven number of updates that moving objects generate within a short time interval. Moving objects can have distance or time based location update protocols with the YT. In the YT, the least and most time interval between location updates is set as 1 minute and 30 minutes respectively for time based protocols.

Moving objects were created continuously up to a simulated steady state of one million objects with different update frequencies for each object. The load generator created 50K object updates according to uniform or skew load specifications; these updates were pushed to the corresponding directory servers using sockets. The maximum capacity of a YT node was fixed at 400K objects. Garbage collection was timed be executed every 5 minutes or 50K updates depending on which ever happened first.

6.4.2.1 Uniform Load Generator

The uniform load generator aims to generate moving objects that are spatially well distributed and that follow a normal distribution. To generate a uniform pattern of moving objects over a period of time as shown in Figure 6.1, we followed an approach that would ensure that at any given point of time, the total number of moving objects within each quadrant would not differ by more than 5%. Since the YT space partitioning scheme divides space into quadrants, we started with the center of each quadrant and placed an equal number of moving objects in each to start. Objects were allowed to move only within their own quadrant to ensure uniform load across quadrants. Periodically after every few cycles, we increased the load in each quadrant by a constant amount, to maintain uniform increase in load. Increasing load should prompt the YT node to split and dispersing load or objects moving out of area should force the YT nodes to merge. Figure 6.1 shows the simulated uniform load generator that we have used for the experiments.

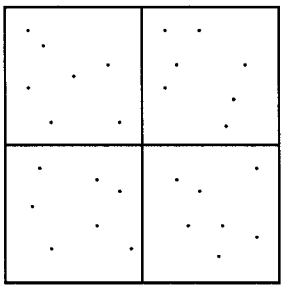


Figure 6.1 Uniform Load Generator

6.4.2.2 Skewed Load Generator

Skewed load represents a spatially uneven distribution of load. To generate a skewed load, we predefined a route for all the moving objects from each quadrant, to converge at

the skewed area identified by two points (700, 350) and (800, 400) as shown in Figure 6.2. Objects in SW would travel NE to get to the skew center and then generate location updates while in the skewed area and travel back to their original positions. We drew this analogy from a real-life situation, where objects from the suburb travel to the city for work and return to the suburb at the end of the day. The pattern of increasing the load in one area and then dispersing helps us to evaluate the index structure for its adaptability.

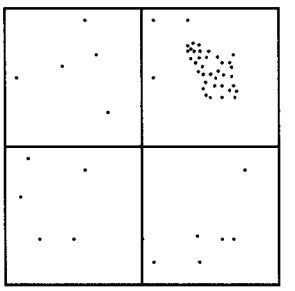


Figure 6.2 Skewed Load Generator

6.5 Performance Results

One of the main virtues of the YT is its ability to handle updates of a very large number of moving objects in near real-time. The performance of a distributed spatial main memory index is characterized by its main memory storage utilization, its capacity to handle large update rates of moving objects, and to answer queries efficiently. In general, a spatial access method can be evaluated with respect to both time and space complexity. To prove our hypothesis, we developed two versions of the YT, one with dynamic space partitioning and another with static space partitioning scheme. Both the dynamic and static space partitioning schemes employed a BQT for spatial indexing in the background. The test environment for both the static and dynamic YT was the same.

Each static partitioning node communicates only to the root server, but cannot dynamically adapt to or branch out to child nodes.

6.5.1 Uniform Load Query Processing

This section presents the results of query processing of box, circle, and polygon query predicates under uniform load. In uniform query processing, query boxes of different sizes were tried (1%, 5% and 10% of total area). The processing times were expected to increase linearly with load as well as with the size of query box. For every 100K increase in load, the slope increase in query processing time for box, circle, and polygon predicates is linear for SP, while for the YT the slope increases steeply until 400K for all query predicates. At 400K the query processing performance of the YT is comparably weak against SP and suffers a delay as high as 43%. In our experiments, the maximum number of objects that an instance of YT handles is fixed at 400K, hence YT reorganizes by splitting when the moving object load reaches the limit. The peak value in query processing times is caused by reorganization of YT as depicted in Figures 6.3, 6.4 and 6.5. Reorganization locks access to YT index causing a delay in query processing requests but after the reorganization process completes, the query response time reduces as load is distributed after 400K. On the other hand, query processing time for SP increases linearly with higher load and query processing time is peak at 1M objects. After distributing load at 400K, the YT is well balanced to handle load and achieves maximum performance gain, which could be as high as 57%, similar to circle query processing.

Thus, for a relatively small number of moving object updates, SP outperforms the YT in most cases, but for very large number of updates the YT outperforms SP. The gains recorded in Table 6.2 for SP are for loads less than 400K and average gain for the

YT occurs at loads greater than 400K. Upon analyzing SP and YT it can be inferred that YT utilizes more resources to run handshake threads (to communicate among YT nodes) compared to SP. Handshake threads and constant updating of status act as overheads to the YT, thus adding up to processing time for small loads. Split threshold (400K) is the worst case performance for the YT and 1M is the best case, and fairly balanced query processing time from 500K to 1M indicates that the YT scales up better to increasing load and has the ability to handle virtually unlimited number of objects with greater ease than SP.

Query Predicate	**Average Gain %, SP**	**Average Gain %, YT**	**Split Delay %**	**YT Peak Gain %**
Box, Uniform	12.96	19.81	36.11	37.36
Circle, Uniform	13.43	33.22	38.06	56.84
Polygon, Uniform	14.93	34.57	43.74	57.75

Table 6.2 Uniform Load Query Processing

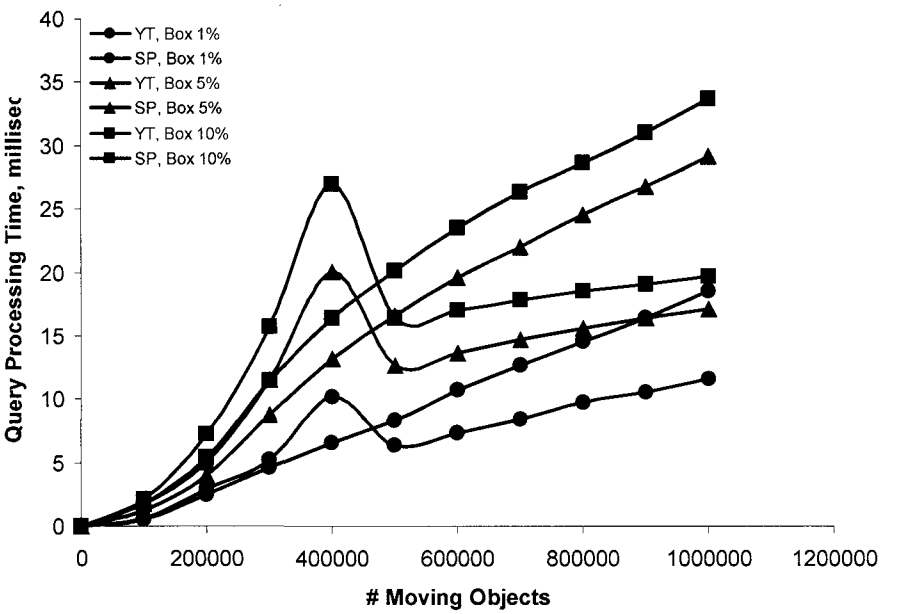


Figure 6.3 Box Query Processing, Uniform Load

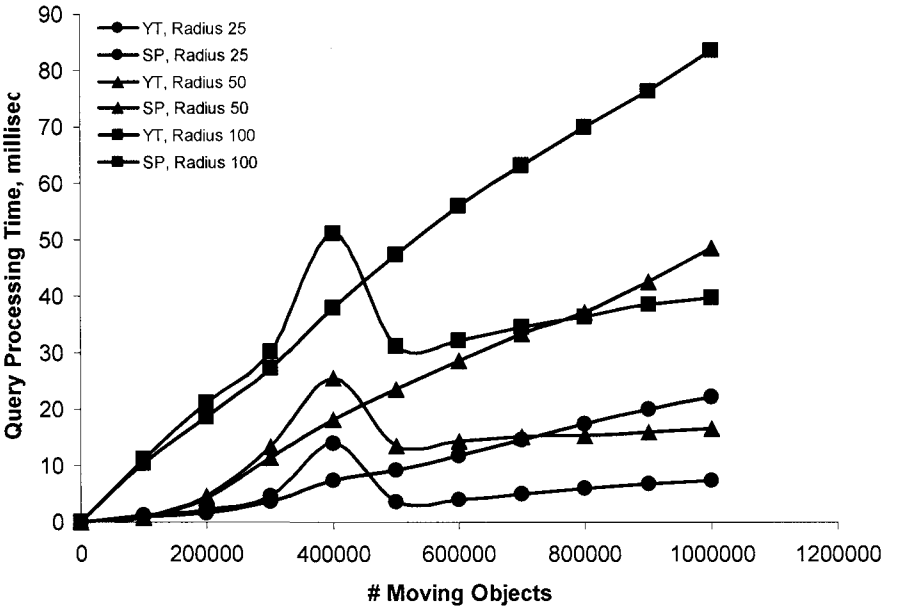


Figure 6.4 Circle Query Processing, Uniform Load

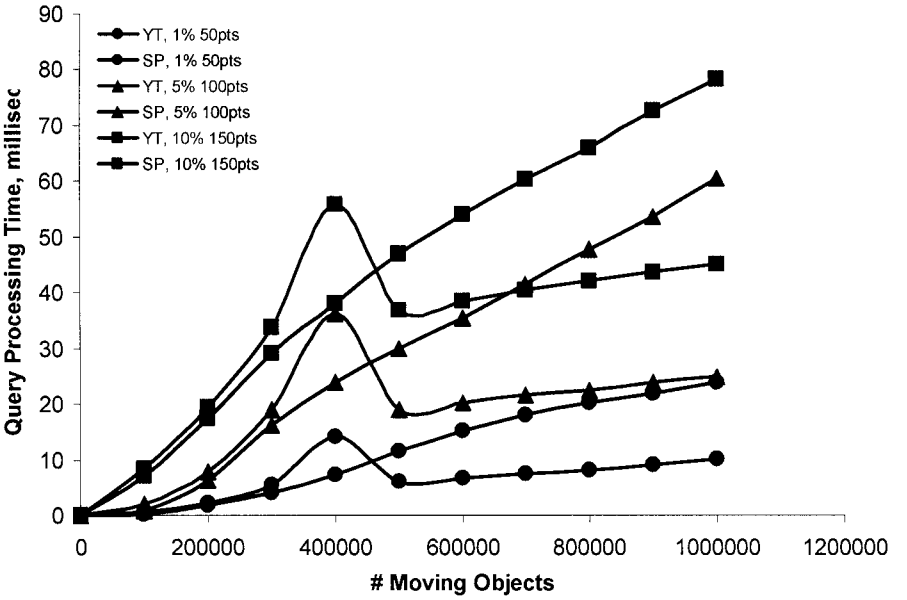


Figure 6.5 Polygon Query Processing, Uniform Load

6.5.2 Skewed Load Query Processing

Query processing of skewed load follows the same trend as that of uniform load, as illustrated in Figures 6.6, 6.7, and 6.8. Another aspect to be noted here is the average gain in the YT after 400K is considerably high at 48% for polygon query processing compared to 21% in SP. Also the peak gain at 1M objects is 77% denoted in Table 6.3 and the worst case performance of the YT occurs at 400K due to split. Split delay for skewed load is also higher 53% compared to 43% in uniform load, since the split procedure is bound to take more time for recursive split that typically happens with skewed load, rather than a one level split for uniform load.

For both skewed and uniform query processing the time taken to process queries increase linearly with increases in query predicate percentage from 1% to 5% and then to 10%. Complexity of the query shape also has an effect on query processing time, which

can be inferred from the fact that polygon queries take more time than circle queries, which in turn requires more processing time than box queries.

Query Predicate	Average Gain %, SP	Average Gain %, YT	Split Delay %	YT Peak Gain %
Box, Skew	20.24	47.92	46.84	75.96
Circle, Skew	19.68	44.21	46.32	77.79
Polygon, Skew	21.92	48.51	53.41	71.82

Table 6.3 Skewed Load Query Processing

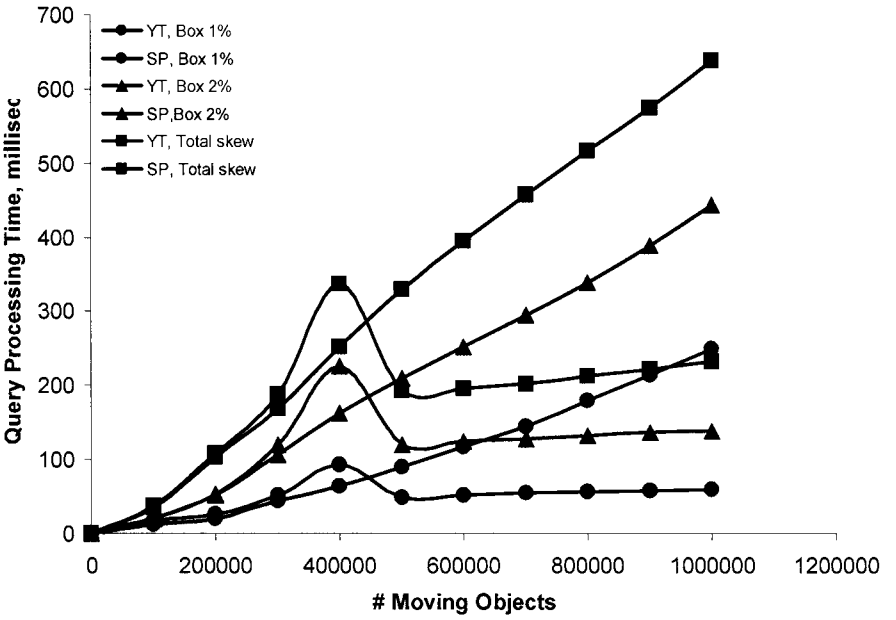


Figure 6.6 Box Query Processing, Skewed load

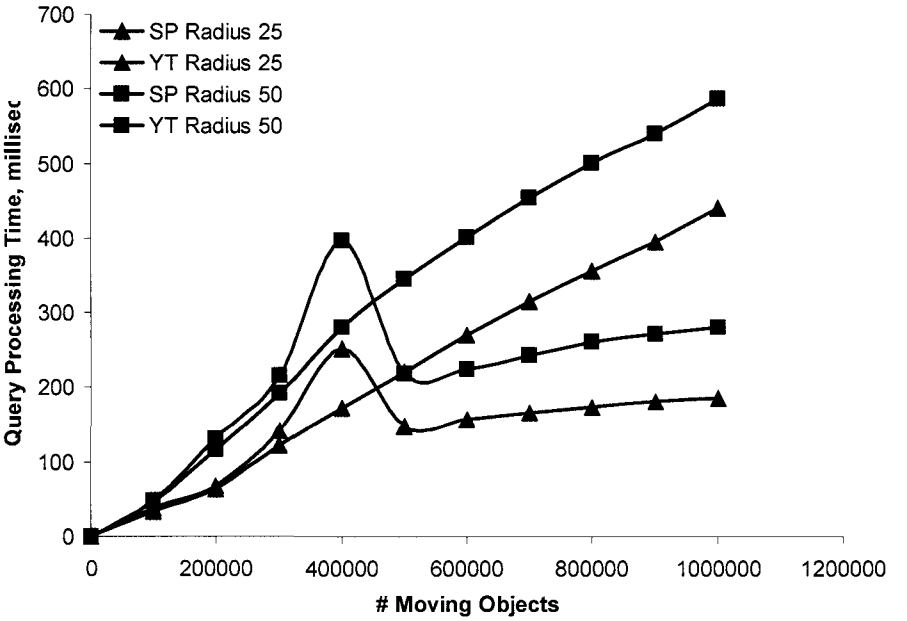


Figure 6.7 Circle Query Processing, Skewed load

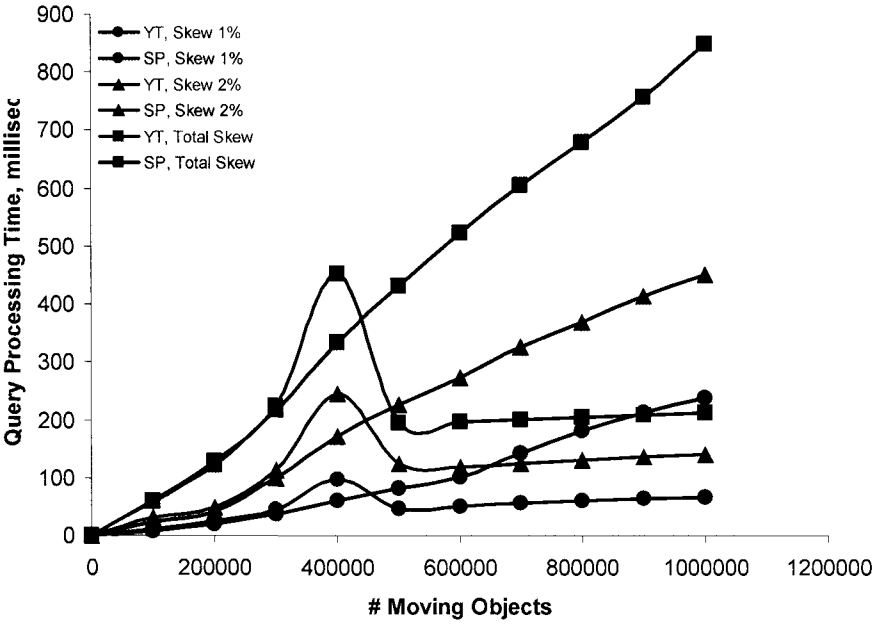


Figure 6.8 Polygon Query Processing, Skewed load

Spatial query processing of uniform and skewed load supports our hypothesis, since it emphasizes the need for a distributed nature of deploying a spatial index structure. This is evident from the fact that YT records maximum gain in query processing performance against its static counterpart, at higher loads by a percentage of 57% for uniform load and 77% for skewed load.

6.5.3 Inserts and Updates

Inserts and update results shown in Figure 6.9 indicate a linearly increasing trend in the insert and update time with increase in load. Both insert and update algorithms involve a tree traversal, which is dependent on the height of BQT. After analyzing insert and update processing times, it may be noted that updates are on an average 56% more time consuming, since updates involve inserting the current version and changing the reference for any previous instance of the same moving object in overflow buckets.

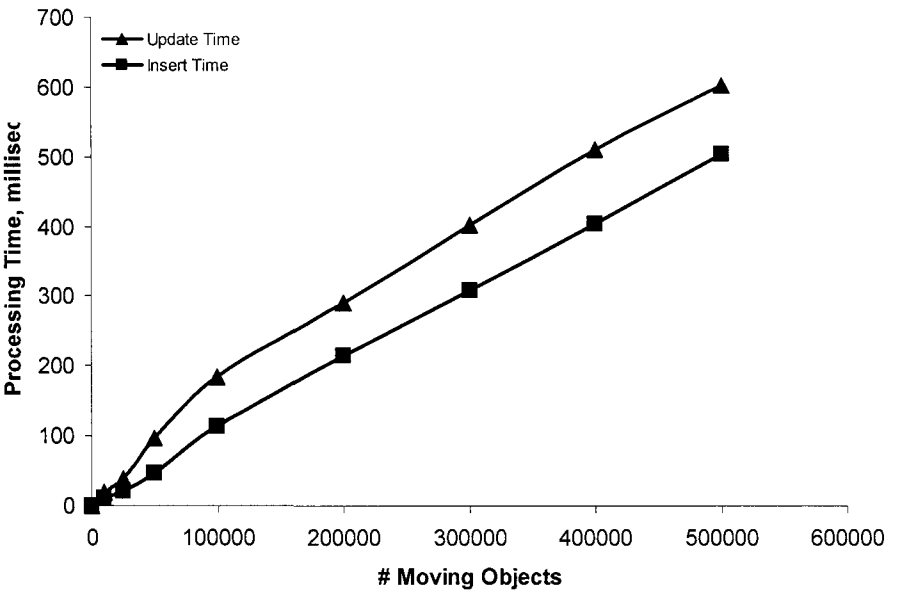


Figure 6.9 Bucket Quad Tree, Inserts Updates

Insert and update processing times, were both reported different versions of YT which were all main-memory based, since the feasibility of an index structure to manage millions of moving objects is already discussed in section 2.6.

6.5.4 Communication Time

Communication time as shown in Figure 6.10 is the amount of time taken to communicate N objects over a network with T1 connection. An average of 11 moving objects per millisecond in de-serialized extensible markup language (XML) format can be communicated across YT nodes in a dedicated network.

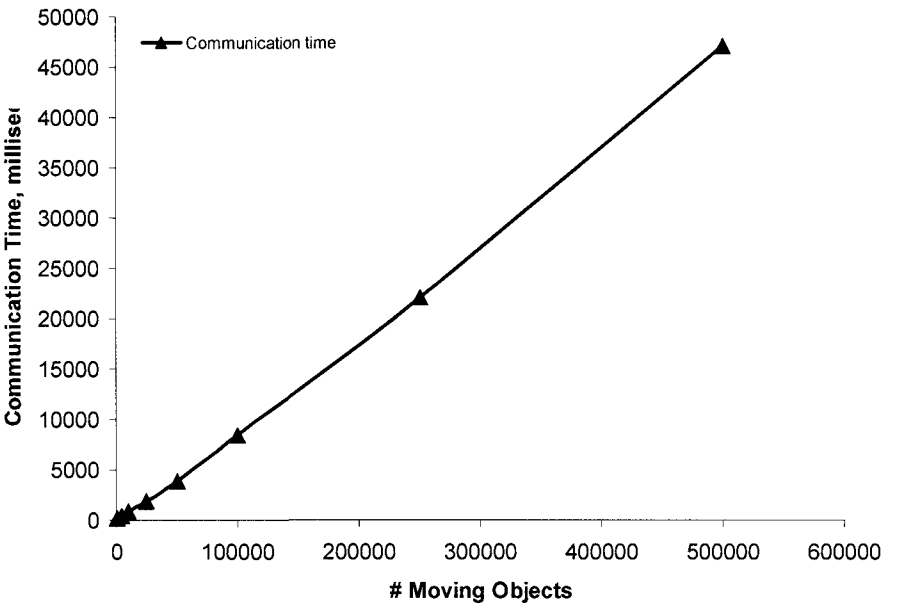


Figure 6.10 Communication Time

6.5.5 Space Requirements

Space requirements of bucket quadtree represent the total number of child nodes required to store moving objects. Skewed load requires more child nodes, and because of the uneven distribution among sibling nodes, child nodes may be under utilized. Inverse of

space requirement is space utilization, which is an indicator of compactness of index structure. Figure 6.11 identifies that skewed load uses on an average 58% more child nodes to index same number of moving objects as in uniform load.

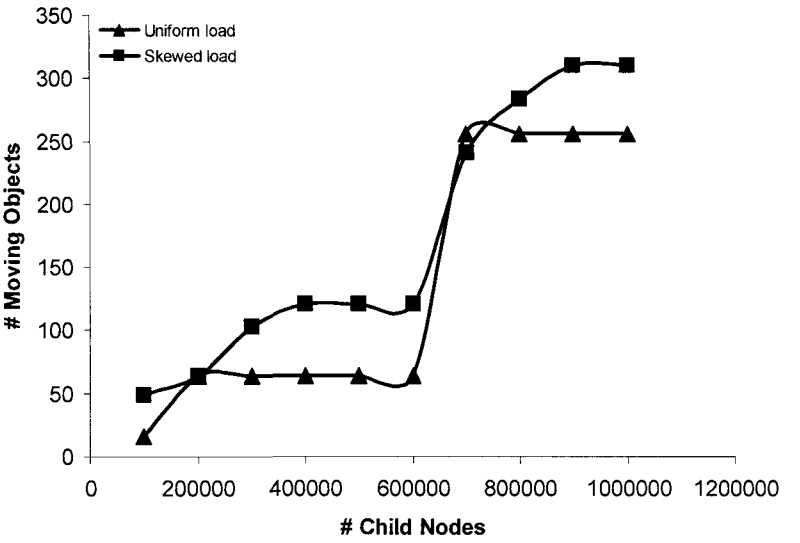


Figure 6.11 Space Requirements of Bucket Quad Tree

6.5.6 Overlap Query Processing

Overlap query processing is an attempt to study the performance of the YT for its ability to process queries in a distributed environment. Overlap queries are spatial range queries, where the predicate overlaps more than one YT node. Ability to process queries distributed over multiple YT nodes is another novel feature of the YT. Overlap query processing time is calculated as the maximum time taken to process the query among all distributed YT nodes. Distributing the query shape typically means each YT node needs to process only a portion of the shape, thus bringing down the processing time. Figures 6.12 and 6.13 illustrate that queries distributed over multiple YT take less processing

time than executing the same query shape on a single YT node. Distributing load across YT nodes achieves an average of 37% increase in query processing performance for box query predicates and 71% performance gain for circle query predicates.

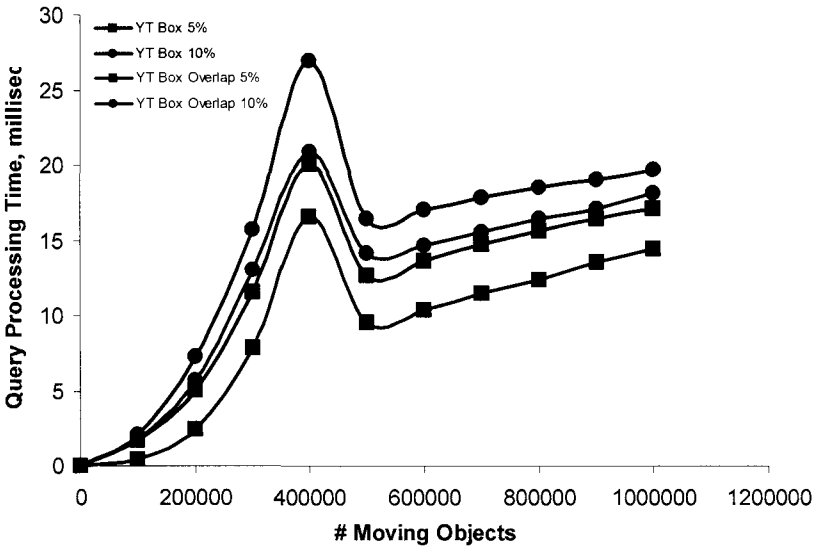


Figure 6.12 Box Overlap Query Processing

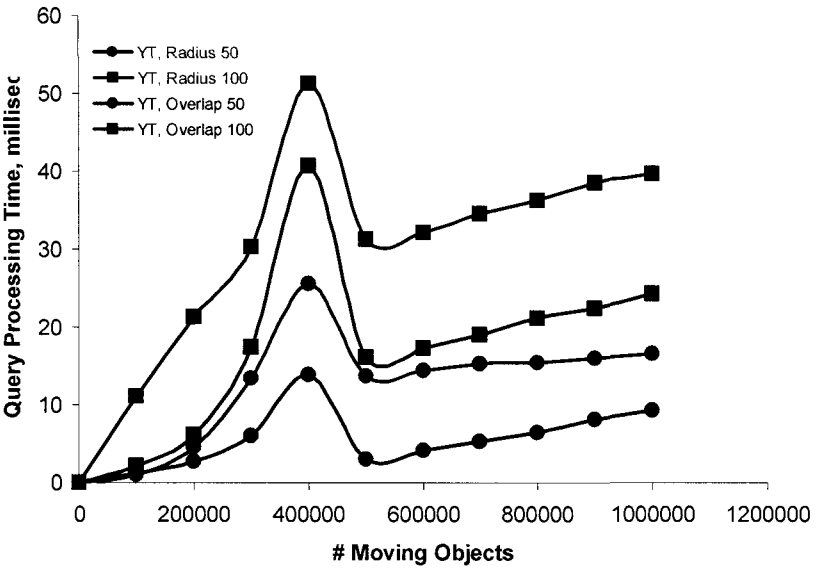


Figure 6.13 Circle Overlap Query Processing

6.5.7 Object Queries

Object queries are queries for retrieving an object based on its identifier (ID). In BQT, object ID is used as key for hash table, and thus object query processing involves traversing to all the leaf nodes and checking if the object is contained in hash table. Analyzing the results in Figure 6.14, we could infer that time taken for object queries increases with a slope of almost 1.17 for every 100K increase in load.

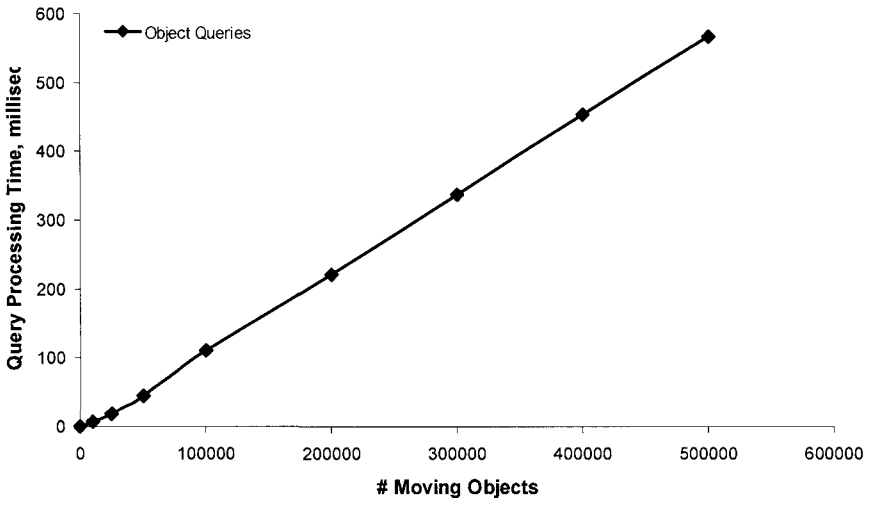


Figure 6.14 Object Queries in BQT

6.6 Summary

The chapter provided a thorough analysis of various YT algorithms and performance results to support the hypothesis that a distributed main memory spatial indexing scheme to index moving objects. Communication time is neglected for analysis, since it is constant between YT nodes and dependent on the client's network card capabilities. Factors influencing query processing costs were identified and significance of deploying a simple space partitioning scheme was highlighted. Results indicate that SP fares well for spatial range query processing over small loads, but dynamic YT performs considerably well for heavy moving object updates and scales up gracefully for increasing load.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Location based services, mobile workforce management, wireless emergency services, and fleet management are some applications in which the user is mobile. Location based applications rely on databases to support large number of users with continuously changing location. Traditional databases with their index structures have not been able to handle large moving object data, since the assumption that data is static until it is explicitly updated, is invalid. A database that supports dynamic attributes for location, such as moving object databases is required to handle location based applications. Moving object databases need to be equipped with spatial index structures to speed up inserts/updates and query processing on moving point objects.

This research work presented is a novel approach to solve the problem of indexing moving objects in a moving object database environment. Our hypothesis suggested that a distributed main-memory based spatial indexing scheme with simple space partitioning and hashing can be used to answer spatial range and object based queries for moving objects efficiently. To scale to large numbers of moving objects and dynamically adapt to varying moving object load, the distributed and collaborative network of nodes is required. We evaluated the performance of the proposed index structure and presented experimental results on concurrent location updates and spatial range queries that demonstrated the feasibility and scalability of the approach. The unique aspect of the YT spatial index is its ability to scale up to virtually unlimited

number of objects and to dynamically request and allocate resources to achieve maximum adaptability.

7.1 Conclusions

The performance tests on spatial range query processing and inserts on YT highlights some of the important considerations in the design of an index structure for moving objects and arrives at a set of conclusions.

7.1.1 Distributed Index

- Deploying a distributed index structure makes the YT scalable to large spatial extents and extremely high concurrent location updates of moving objects.
- Distributed architecture also makes the index structure open, such that it is flexible to grow and shrink, depending on load.
- Hash table that maintains metadata about all distributed nodes in the network allows constant lookup time to locate any node in the network.

7.1.2 Main-Memory

- Main-memory based index structure is required to keep up with the frequency of location updates from moving objects, since the access time to retrieve an object in main-memory can be as low as 5 nanoseconds.
- Main-memory is limited; hence a centralized architecture to deploy an index structure for moving objects becomes infeasible, suggesting a distributed architecture.

- Main-memory is best suited to handle the dynamic increase in the number of location updates to the moving object database.

7.1.3 Handshake protocols

- The YT architecture provides a well-defined set of protocols that support the collaboration of distributed YT nodes to balance location updates and query load dynamically, with respect to uniform and skewed load.
- Collaboration between the YT and the RS ensures that dynamic allocation of resources is possible on remote nodes. The ability to dynamically allocate resources allows the YT to index virtually unlimited number of moving objects if a remote node is willing to share the load.

7.1.4 Space Partitioning

- The YT employs a quadtree based space partitioning scheme that is deliberately simple to promptly split or merge the index structure to account for varying moving object traffic.
- Partitioning of space in each level is achieved in constant time to dynamically adapt to increase in the number of moving objects.

7.1.5 Garbage Collection

- Garbage collection ensures efficient use of main-memory by clearing unchanged or timed out objects based on their location update protocol.

- Garbage collection is an automatically configured process, to prevent under utilization of main-memory as well as avoid clogging of the central processing unit (CPU) cycles.

7.1.6 Redirection

- Redirection of location updates and queries between parent and child nodes ensures that location updates and queries are not lost during index structure reorganization.
- Redirection mechanism consumes memory and computing resources to queue requests and location updates, but is only active during reorganization of an index structure.

7.2 Future Work

Although the YT has addressed a number of problems related to developing an index structure for moving objects, there are several areas for future direction in this line of research.

7.2.1 Cache Conscious

Yellow tree as a main-memory index structure assumes that main-memory data access time is constant, which may not be valid in all cases. The difference in data access times is significant depending on the processor speed and main-memory bus speed. Main-memory index structures need to be cache sensitive, and focus on maintaining as many relevant data items in cache as possible to avoid accessing main-memory, since cache memory in the processor is faster than main-memory.

7.2.2 Partitioning

Space partitioning for YT has worked well to provide dynamic adaptability to increasing load, but the algorithm is recursive and does not guarantee uniform data distribution. Recursive algorithms can be altered to stop after a predefined number of iterations, but may not be efficient for specific data skew such as traffic jams on long 12-lane freeways; thus, the data is highly spatially skewed and might overlap with a split direction of the Quadtree. Data partitioning algorithms such as R tree are better in capturing such spatially skewed data distribution. However, the typical data re-partitioning algorithms that are employed in R tree using quadratic split are both complex and computationally intensive, and lead to spatial index deterioration over time (Denny et al 2003). A more effective data partitioning algorithm needs to be fairly simplistic to accommodate frequent location updates, yet deal well with aforementioned types of spatially skewed data distribution. Another aspect in partitioning is to either share an overlapping region between split child nodes or share-nothing. The YT employs a share-nothing space partitioning scheme in which a moving object's retrieval involves a unique route from the root of the tree. Split space can overlap resulting in more than one route to retrieve an object but can also minimize the number of updates for some objects that move along the border of partitions.

7.2.3 Uncertainty

For spatial range queries, the search range in the moving object index structure should be modified since the query was based on a continuously changing attribute. Current location data derived from location update protocols is inherently inaccurate. Hence spatial range query processing needs to account for uncertainty in current location and

provide an error percentage with results. This error percentage in spatial range query processing results can serve to determine the confidence associated with the results.

REFERENCES

Abbas, A. (2004). *Grid Computing: A Practical Guide to Technology and Applications*, Charles River Media, Hingham, Massachusetts.

Abdelguerfi, M., Givaudan, J., Shaw, K., and Ladner, R. (2002). The 2-3 TR-tree, A Trajectory-Oriented Index Structure for Fully Evolving Valid-time Spatio-Temporal Datasets. *In Proceedings of the ACM Workshop on Advances in GIS*, ACM GIS, McLean, Virginia. Voisard, A. and Chen, S. (eds): pp. 29-34.

Attiya, H., and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, Second edition, John Wiley and Sons, New Jersey.

Bayer, R. (1971). Binary B-trees for Virtual Memory. *In Proceedings of ACM, SIGFIDET Workshop on Data Description, Access and Control*, San Diego, California, Codd, E. F. and Dean, A. L (eds), ACM: pp. 219-235.

Ballinger, C. (2006) Born To Be Parallel. Available: http://www.teradata.com/t/page/87083/index.html

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. (1993). On Optimal Multiversion Access Structures. *In Proceedings of the Third International Symposium on Large Spatial Databases*, Abel, D. J. and Ooi, B. C. (eds), Lecture Notes in Computer Science, Springer-Verlag, 692: pp. 123-141.

Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990): The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, Garcia-Molina, H. and Jagadish, H. V., ACM Press: pp. 322-331.

Bentley, J.L. (1979). Multidimensional Binary Search Trees in Database Applications. *IEEE Transactions of Software Engineering,* 5(4): 333-340.

Bernstein, P., Brodie, M., Ceri, S., DeWitt, D., Franklin, M., Garcia-Molina, H., Gray, J., Held, J., Hellerstein, J., Jagadish, H., Lesk, M., Maier, D., Naughton, J., Pirahesh, H., Stonebraker, M., and Ullman, J. (1998). The Asilomar Report on Database Research. *ACM SIGMOD Record*, 27(4): 74-80.

Bohlen, M. H., Jensen, C. S., and Skjellaug, B. (1998): Spatio-Temporal Database Support For Legacy Applications. *In Proceedings of the 1998 ACM Symposium on Applied Computing*, Atlanta, Georgia: pp. 226-234.

Boral, H., Alexander, W., Clay, L., Copeland, G. P., Danforth, S., Franklin, M. J., Hart, B. E., Smith, M. G., and Valduriez, P. (1990). Prototyping Bubba: A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering* 2(1): 4-24.

Bowditch, N. (1995). The American Practical Navigator - An Epitome of Navigation, Available: http://www.irbs.com/bowditch/pdf/glossary/gloss-a.pdf, 1995.

Brain, M. (2006). How Hard Disks Work, Available: http://computer.howstuffworks.com / hard-disk.htm

Brain, M. and Tyson, J. (2006). How Cell Phones Work, Available: http://electronics. howstuffworks.com/cell-phone2.htm.

Brinkhoff, T. (2002). A Framework For Generating Network-Based Moving Objects. *Geoinformatica*, 6(2): 153-180.

Brinkhoff, T. (2000). Generating Network-Based Moving Objects. *In Proceedings of the Twelfth International Conference on Scientific and Statistical Database Management,*

Berlin, Germany, Gunther, O. and Lenz, H. J. (eds), IEEE Computer Society: pp. 253-255.

Brinkhoff, T., Kriegel, H. P., Schneider, R., and Seeger, B.: Multi-Step Processing of Spatial Joins. *In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota: pp. 197-208.

Burton, F. W., Kollias, J. G., Matsakis, D. G., and Kollias, V. G. (1990). Implementation of Overlapping B-Trees For Time And Space Efficient Representation of Collections of Similar Files. *The Computer Journal* 33(3): 279-280.

Chakka, V. P., Everspaugh, A.C., and Patel, J.M. (2003). Indexing Large Trajectory Data Sets With SETI. *In Proceedings of the First Biennial Conference on Innovative Data Systems Research*, CIDR, Asilomar, California: pp. 164 -175.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (1990). *Introduction to Algorithms*, Second Edition, MIT Press/ McGraw-Hill Book Company, Boston.

Dayal, U., Blaustein, B. T., Buchmann, A. P., Chakravarthy, U. S., Hsu, M., Ledin, R., McCarty, D. R., Rosenthal, A., Sarin, S., Carey, M. J., Livny, M., and Jauhari, R. (1988). The HiPAC project: Combining Active Databases and Timing Constraints. *ACM Sigmod Record*, ACM Press, 17(1): 51-70.

Denny, M., Franklin, M., Castro, P., and Purakayastha, A. (2003). Mobiscope: A Scalable Spatial Discovery Service for Mobile Network Resources. *In Proceedings of the Fourth International Conference in Mobile Data Management*, Melbourne, Australia, Chen, M., Chrysanthis, P. K., Sloman, M., and Zaslavsky, A. B. (eds), Lecture Notes in Computer Science, Springer, 2574: pp. 307-324.

DeWitt, D. J., and Gray, J. (1992). Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 35(6): 85-98.

DeWitt, D. J., Kabra, N., Luo, J., Patel, J. M., and Yu, J. (1994). Client-Server Paradise. *In Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago de Chile, Chile, Bocca, J. B., and Jarke, M., and Zaniolo, C. (eds), Morgan Kaufmann: pp. 558-569.

Dubrovin, B. A., Fomenko, A. T., and Novikov, S. P. (1993). *Modern Geometry-Methods and Applications, Part I: The Geometry of Surfaces, Transformation Groups and Fields.* Graduate Texts in Mathematics, Second edition, Springer-Verlag, New York.

Egenhofer, M. J., and Franzosa, R. D. (1991). Point-Set Topological Spatial Relations. *International Journal of Geographical Information Systems,* 5(2):161-174.

Epinions Website (2006). Kingston 1GB RAM (KVR333X64C25/1G), Available: http://www.epinions.com

Erwig, M., and Schneider, M. (1999): Developments in Spatio-Temporal Query Languages. *In Proceedings of the Tenth International Workshop on Database & Expert Systems Applications*, Florence, Italy, IEEE Computer Society: pp. 441-449.

Foster, I., and Kesselman, C. (1996). Globus: A Metacomputing Infrastructure Toolkit. In Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France. *The International Journal of Supercomputer Applications and High Performance Computing* 11(2): 115-128.

Frank, A. U. (1981). Application of DBMS to Land Information Systems. *In Proceedings of the Seventh International Conference on Very Large Databases*, Cannes, France, IEEE Computer Society: pp. 448-453.

Ganesan, P., and Yang, B., Garcia-Molina, H. (2004). One Torus to Rule them All: Multidimensional Queries in P2P Systems. *In Proceedings of the Seventh International Workshop on the Web and Databases*, WebDB 2004, Amer-Yahia, S. and Gravano, L. (eds): pp. 19-24.

Gnutella (2006). Gnutella, http://www.gnutella.com/

Gowrisankar, H. and Nittel, S. (2002). Reducing Uncertainty in Location Prediction of Moving Objects in Road Networks. *Second International Conference on Geographic Information Science* (GIScience 2002), Boulder, Colorado.

Griffiths, T., Fernandes, A. A. A., Paton, N.W., Mason, K.T., Huang, B., Worboys, M.F., Johnson, C., Stell, J.G. (2001). Tripod: A Comprehensive System for the Management of Spatial and Aspatial Historical Objects. *In Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems*, Atlanta, Georgia, Aref, W.G. (ed): pp. 118-123.

Güting, R.H., Böhlen, M., Erwig, M., Jensen, C.S., Lorentzos, N., Schneider, M., and Vazirgiannis, M. (2000). A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems* 25(1):1-42.

Guttman, A. (1984). R-trees: A Dynamic Index Structure for Spatial Searching. *In Proceedings of the 1984 ACM International Conference on Management of Data*, New York, Yormark. B (ed), ACM Press: pp. 47-57.

Hjelm, J. (2002). Creating Location Services for the Wireless Web: Professional Developer's Guide, John Wiley & Sons, New York.

Hornsby, K. and Egenhofer, M. J. (2002). Modeling Moving Objects Over Multiple Granularities. *Special issue on Spatial and Temporal Granularity, Annals of Mathematics and Artificial Intelligence*. Springer Netherlands, 36(1-2):177-194.

Hwang, S., Kwon, K., Cha, S., and Lee, B. (2003). Performance Evaluation of Main-Memory R-tree Variants. *Eighth International Symposium on Spatial and Temporal Databases*, Greece, Hadzilacos, T., Manolopoulos, Y., Roddick, J., and Theodoridis, Y. (eds), Lecture Notes in Computer Science, Springer 2750: pp. 10-27.

IBM Grid computing (2006). Available: http://www-1.ibm.com/grid/

Jain, R. (1991). The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons Incorporated, New Jersey.

Java (2006). The Java™ Language: An Overview, Available: http://java.sun.com/docs/overviews/java/java-overview-1.html.

Java Tuning (2006). Java Tuning White Paper, Available: http://java.sun.com/performance/reference/whitepapers/tuning.html#section4.2.4.

Kamel, I., and Faloutsos, C. (1994). Hilbert R-tree - an Improved R-tree Using Fractals. *In Proceedings of the Twentieth VLDB Conference*, Santiago, Chile, Bocca, J. B., Jarke, M., and Zaniolo, C. (eds), Morgan Kaufmann: pp. 500-509.

Kazaa (2006). Kazaa, http://www.kazaa.com/us/index.htm

Kollios, G., Tsotras, V., Gunopulos, D., Delis, A., and Hadjieleftheriou, M. (2001). Indexing Animated Objects Using Spatiotemporal Access Methods. *IEEE Transactions on Knowledge and Data Engineering* 13(5): 758-777.

Kriegel, H., Pfeifle, M., Poetke, M., and Seidl, T. (2002). A Cost Model for Interval Intersection Queries on RI-Trees. *In Proceedings of the Fourteenth International Conference on Scientific and Statistical Database Management*, Scotland, United Kingdom, IEEE Computer Society: pp. 131-141.

Kwon, D., Lee, S., and Lee, S. (2002). Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. *In Proceedings of the Third International Conference on Mobile Data Management*, Singapore, IEEE Computer Society: pp. 113-120.

LaMance, J., DeSalas, J., and Jarvinen, J. (2002). Assisted GPS: A Low-Infrastructure Approach, http://www.gpsworld.com/gpsworld/article/articleDetail.jsp?id =12287.

Layton, J., Brain, M., and Tyson, J. (2006). How Cell Phones Work, Available: http://electronics.howstuffworks.com/cell-phone2.htm.

Lee, M., Hsu, W., Jensen, C., Cui, B. and Teo, K. (2003). Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. *In Proceedings of the Twenty-ninth International Conference on Very Large Data Bases*, Berlin, Germany, Freytag, J. C., Lockemann, P. C., Abiteboul, S., Carey, M. J., Selinger, P. G., and Heuer, A. (eds), Morgan Kaufmann: pp. 608-619.

Lehman, T. and Carey, M. (1986). A Study of Index Structures for Main Memory Database Management Systems. *In Proceedings of the Twelfth International Conference*

On Very Large Databases, Kyoto, Japan, Chu, W. W., Gardarin, G., Ohsuga, S., and Kambayashi, Y. (eds), Morgan Kaufmann: pp. 294-303.

Leick, A. (2004). *GPS Satellite Surveying*, Third edition, John Wiley and Sons, New Jersey.

Lomet, D., and Salzberg, B. (1989). Access Methods for Multiversion Data. *In Proceedings of the ACM SIGMOD Conference on the Management of Data, Portland,* Oregon, Clifford, J., Lindsay, B. G., and Maier, D. (eds), ACM Press: pp. 315-324.

Marks, L. (2003). The 802.11g standard -- IEEE, Available at http://www-128.ibm.com/developerworks/wireless/library/wi-ieee.html.

Miller, H. (1991). Modelling Accessibility Using Space-Time Prism Concepts within Geographical Information Systems. *International Journal of Geographical Information Systems*, 5(3): 287-301.

Mokbel, M., Aref, W., Hambrusch, S., and Prabhakar, S. (2003). Towards Scalable Location-Aware Services: Requirements And Research Issues. *In Proceedings of the Eleventh ACM International Symposium on Advances in Geographic Information Systems*, New Orleans, Louisiana, Hoel, E. and Rigaux, P. (eds): pp. 110-117.

Mokbel, M., Thanaa, M., and Aref, W. (2003). Spatio-Temporal Access Methods, *IEEE Data Engineering Bulletin*, 26(2): 40-49.

Myllymaki, J. and Kaufmann, J. (2003). High-Performance Spatial Indexing for Location-Based Services. *In Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungary, Chen, Y.R., Kovacs, L., and Lawrence, S. (eds) ACM: pp. 112-117.

Napster (2006). Napster, http://www.napster.com/

Nascimento, M.A., and Silva, J. R. O. (1998). Towards Historical R-trees. *In Proceedings of Thirteenth ACM Symposium on Applied Computing*, Atlanta, GA, pp. 235-240.

Nascimento, M.A., Silva, J.R.O., and Theodoridis, Y. (1999). Evaluation of Access Structures for Discretely Moving Points. *In Proceedings of International Workshop on Spatio-Temporal Database Management*, Scotland, UK. Böhlen, M.H., Jensen, C.S., and Scholl, M. (eds), Lecture Notes in Computer Science, Springer, 1678: pp. 171-188.

NAVSTAR GPS Satellite (2006). GNSS Summary, http://cddis.gsfc.nasa.gov/gnss_summary.html.

Pfoser, D., and Jensen, C.S. (1999). Capturing the Uncertainty of Moving-Object Representations. *In Proceedings of the Sixth International Symposium on Advances in Spatial Databases*, Hong Kong, China, Guting, R. H., Papadias, D., and Lochovsky, F. H. (eds), Lecture Notes in Computer Science, Springer, 1651: pp. 111-132.

Pfoser, D., Jensen, C.S., and Theodoridis, Y. (2000). Novel Approaches to the Indexing of Moving Object Trajectories. *In Proceedings of the Twenty-sixth International Conference on Very Large Data Bases*, Cairo, Egypt, Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, H., Kamel, N., Schlageter, G., and Whang, K. (eds), Morgan Kaufmann: pp. 395-406.

Pfoser, D. (2002). Indexing the Trajectories of Moving Objects. *IEEE Data Engineering Bulletin* 25(2): 3-9.

Porkaew, K., Lazaridis, I., and Mehrotra, S. (2001). Querying Mobile Objects in Spatio-Temporal Databases. *In Proceedings of the Seventh International Symposium on Spatial and Temporal Databases*, Los Angeles, California, Jensen, C. S., Schneider, M., Seeger, B., and Tsotras, V. J. (eds), Lecture Notes in Computer Science, Springer, 2121: pp. 59-78.

Procopiuc, C.M., Agarwal, P.K., and Har-Peled, S. (2002). STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. *In Proceedings of the Fourth International Workshop on Algorithm Engineering and Experiments*, Mount, D. M. and Stein, C. (eds), Lecture Notes in Computer Science, Springer, 2409: pp. 178-193.

Ramakrishnan, R. and Gehrke, J. (2000). *Database Management Systems*. McGraw-Hill.

Rigaux, P., Scholl, M., and Voisard, A. (2001). *Spatial Databases with Application to GIS*. Second edition, Morgan Kaufmann, San Francisco, California.

Sadoski, D. (1997). Client/Server Software Architectures - An Overview, Available: http://www.sei.cmu.edu/str/descriptions/clientserver_body.html, 1997.

Saglio, J.M., and Moreira, J. (2001). Oporto: A Realistic Scenario Generator for Moving Objects, *GeoInformatica* 5(1): 71-93 2001.

Saltenis, S., Jensen, C.S., Leutenegger, S.T., and Lopez, M.A. (2000). Indexing the Positions of Continuously Moving Objects. *In Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD, Dallas, Texas, Chen, W., Naughton, J. F., and Bernstein, P. A., ACM: 331-342.

Samet, H. (1984). The Quadtree and Related Hierarchical Data Structures, *ACM Computer Surveys* 16(2):187-260.

Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company.

Skiena, S. S. (1997). *The Algorithm Design Manual*, Springer-Verlag, New York.

Song, Z. and Roussopoulos, N. (2003). SEB-tree: An Approach to Index Continuously Moving Objects. *In Proceedings of the Fourth International Conference in Mobile Data Management*, Melbourne, Australia, Chen, M., Chrysanthis, P. K., Sloman, M., and Zaslavsky, A. B. (eds), Lecture Notes in Computer Science, Springer, 2574: pp. 340-344.

Song, Z. and Roussopoulos, N. (2001). Hashing Moving Objects. *In Proceedings of the Second International Conference on Mobile Data Management*, Hong-Kong, China, Tan, K., Franklin, M. J., and Lui, J. C. S. (eds), Lecture Notes in Computer Science, Springer 1987: pp. 161-172.

Stoica, I., Morris, R., Karger, D., Kaashoek, F.M. and Balakrishnan, H. (2001). Chord: A Scalable Peertopeer Lookup Service for Internet Applications. *In Proceedings of ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Diego, California, pp. 149-160.

Tao, Y. and Papadias, D. (2001). MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *In Proceedings of the Twenty-seventh International Conference on Very Large Data Bases*, VLDB, Roma, Italy, Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanara, K., and Snodgrass, R. T. (eds), Morgan Kaufmann, pp. 431-440.

Tao, Y., Papadias, D., and Sun, J. (2003). The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *In Proceedings of Twenty-ninth International Conference on Very Large Data Bases*, Berlin, Germany, Freytag, J. C., Lockemann, P. C., Abiteboul, S., Carey, M. J., Selinger, P. G., and Heuer, A. (eds), Morgan Kaufmann: pp. 790-801.

Tao,Y. and Papadias, D. (2001). Efficient Historical R-trees. *In Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management*, Virginia, Kerschberg, L. and Kafatos, M., IEEE Computer Society: pp. 223-232.

Tayeb, J., Ulusoy, O., and Wolfson, O. (1998). A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal,* 41(3): 185-200.

Theodoridis, Y. (2003). Ten Benchmark Database Queries for Location-Based-Services. *The Computer Journal*, 46(6):713-725.

Theodoridis, Y., and Nascimento, M. A. (2000). Generating Spatiotemporal Datasets on the WWW. *SIGMOD Record*, 29(3):39-43.

Theodoridis, Y., Vazirgiannis, M., and Sellis, T. (1996). Spatio-Temporal Indexing for Large Multimedia Applications. *In Proceedings of the Third IEEE Conference on Multimedia Computing and Systems:* pp. 441-448.

Trajcevski, G., Wolfson, O., Zhang, F., and Chamberlain, S. (2002). The Geometry of Uncertainty in Moving Objects Databases. *In Proceedings of the Eighth International Conference on Extending Database Technology*, March 25-27, Prague, Czech Republic. Jensen, C.S., Jeffery, K.G., Pokorn, J., Saltenis, S., Bertino, E., Bohm, K., and Jarke, M. (eds), Lecture Notes in Computer Science, Springer 2287: pp. 233 - 250.

WAP (2002). Wireless Application Protocol, Technical white paper, Available: http:// www. wapforum.org/what/WAPWhite_Paper1.pdf, 2002.

Winder, R.K. (2000). The Kinetic PR Quadtree, Available: http://www.cs.umd.edu /~mount/Indep/Ransom/

Wolfson, O., Xu, B., Chamberlain, S., and Jiang, L. (1998). Moving Objects Databases: Issues and Solutions. *In Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, Capri, Italy, Rafanelli, M. and Jarke, M. (eds), IEEE Computer Society: pp. 111-122.

Worboys, M. F. (1994). A Unified Model for Spatial and Temporal Information. *The Computer Journal*, 37(1):26-34.

Worboys, M. (1998). Imprecision in Finite Resolution Spatial Data. *Geoinformatica* 2(3): 257-280.

Xia, Y. and Prabhakar, S. (2003). Q+Rtree: Efficient Indexing for Moving Object Database. *In Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, Kyoto, Japan, Cha, S. K. and Yoshikawa, M. (eds), IEEE Computer Society: pp. 175-182.

Xu, X., Han, J., and Lu, W. (1990). RT-Tree: An Improved R-tree Index Structure for SpatioTemporal Databases. *In Proceedings of the Fourth International Symposium on Spatial Data Handling*, Zurich, Switzerland, Brassel, K. and Kishimoto, H. (eds): pp. 1040-1049.

BIOGRAPHY OF THE AUTHOR

Hariharan Gowrisankar was born in Chennai, India on February 19, 1978. He attended Vailankanni Matriculation higher secondary school in Chennai. He completed his bachelor's degree in GeoInformatics at the College of Engineering, Anna University in 1999. He worked as a Software Analyst for the Modular GIS Environment (MGE) team in Intergraph India from August 1999 to August 2001. Driven by his passion for deeper understanding of spatio-temporal databases he headed for a Masters degree in Spatial Information Science and Engineering department at University of Maine, Orono in fall 2001. Upon completion of coursework, he also completed independent study on spatial indexing schemes at GE Energy in Kansas City through Penpower Consulting from May 2004 to July 2005. Later, he worked with South Florida Water Management District (SFWMD) on developing customized tools for ArcHydro data model. He is currently working on a contract with Sacramento County Sheriff Department to automate GIS data management tasks and deploy web based GIS applications. Hariharan is a candidate for the Master of Science degree in Spatial Information Science and Engineering from The University of Maine in December, 2006.