FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

AN INDEXING STRUCTURE AND APPLICATION MODEL FOR VEHICLES MOVING ON ROAD NETWORKS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Xiangyu Ye

2004

To: Dean R. Bruce Dunlap
College of Arts and Sciences

This dissertation, written by Xiangyu Ye, and entitled An Indexing Structure and Application Model for Vehicles Moving on Road Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

_____
Geoffrey Smith

_____
Ouri Wolfson

_____
Nagarajan Prabakar

_____
Shu-Ching Chen

_____
Naphtali Rishe, Major Professor

Date of Defense: July 13, 2004

The dissertation of Xiangyu Ye is approved.

_____
Dean R. Bruce Dunlap
College of Arts and Sciences

_____
Dean Douglas Wartzok
University Graduate School

Florida International University, 2004

DEDICATION

I dedicate this dissertation to my two grandmothers and my son Michael.

ACKNOWLEDGMENTS

I thank my major professor Naphtali Rishe for his academic advice and guidance. I am grateful for his offers throughout the past seven years. From time to time, it was these offers altogether that made it possible for me to go back to school and eventually reach this point.

I am indebted to Geoffrey Smith, from whom I learn to be a good researcher. At the hardest time of this dissertation research, Dr. Smith helped me out with his invaluable advice and encouragement. In the late stage, he carefully reviewed and pointed out great many errors in the dissertation.

Maxim Chekmasov has not only spent time providing directions and discussions at the early stage of the dissertation research, but also carefully read and corrected my early versions of writings. Nagarajan Prabakar has spent time discussing, directing and correcting any new ideas I had. Shu-Ching Chen has provided valuable critiques on my early work and constructive suggestions on the final results. In the classes they teach, Xudong He and Giri Narasimhan have showed me the beauty of algorithms from different angles. I thank the above FIU professors for their time, knowledge and support.

I wish to thank Ouri Wolfson and express my special appreciation to him for his contributions to Moving Objects Databases research area. It was Dr. Wolfson's work that attracted me to this area of research, and the information and advice he provided in the final stage has resulted in the improvement of the completeness of this dissertation.

Several of my friends have helped me in various ways. Fang Xie and Jiacun Wang helped me greatly in taking care of my son when I started taking classes at FIU. Natalia Terekhova, Li Yang and Olena Zhyzhkevych helped me with registration issues and

various of forms during my physical absence from school. School of Computer Science secretary Maria Monteagudo thoughtfully did all the required paper work during the whole course of my graduate study at FIU, and handled all the final forms for me. Gaolin Zheng and I had a very informative discussion on a topic in Computational Geometry. I thank them and appreciate their help and time.

The last, but not least, thanks go to my family. My aunt Xiu not only encouraged me to finish the degree, but also supported me through the toughness of living in a foreign country. My mother in law Hui encouraged and comforted me on my fear of technical writing. Even though being desperately waiting for the first “doctor” in the family, my father and my uncle Yan didn’t ask me too often about when I was going to graduate in the last couple of years. My husband Charles and our son Michael provided many constructive suggestions on the research and application of the dissertation. Charles set up and maintained a fully equipped research lab for me in our California home, where the key solutions on this research topic were formed and the dissertation was finished. Many of our discussions on systems and hardware also turned out to be very valuable to the quality of this dissertation. Charles also provided full financial support during the last year of my PhD research. I thank all my family members for their love and patience.

ABSTRACT OF THE DISSERTATION

AN INDEXING STRUCTURE AND APPLICATION MODEL

FOR VEHICLES MOVING ON ROAD NETWORKS

by

Xiangyu Ye

Florida International University, 2004

Miami, Florida

Professor Naphtali Rishe, Major Professor

Moving objects database systems are the most challenging sub-category among Spatio-Temporal database systems. A database system that updates in real-time the location information of GPS-equipped moving vehicles has to meet even stricter requirements. Currently existing data storage models and indexing mechanisms work well only when the number of moving objects in the system is relatively small. This dissertation research aims at the real-time tracking and history retrieval of massive numbers of vehicles moving on road networks. A total solution is provided for the real-time update of the vehicles' location and motion information, range queries on current and history data, and prediction of vehicles' movement in the near future.

To achieve these goals, a new approach called Segmented Time Associated to Partitioned Space (STAPS) is first proposed in this dissertation for building and manipulating the indexing structures for moving objects databases.

Applying the STAPS approach, an indexing structure of associating a time interval tree to each road segment is developed for real-time database systems of vehicles moving on road networks. The indexing structure uses affordable storage to support real-time

data updates and efficient query processing. The data update and query processing performance it provides is consistent without restrictions such as a time window or assuming linear moving trajectories.

An application system design based on distributed system architecture with centralized organization is developed to maximally support the proposed data and indexing structures. The suggested system architecture is highly scalable and flexible. Finally, based on a real-world application model of vehicles moving in region-wide, main issues on the implementation of such a system are addressed.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

1 INTRODUCTION

The Global Positioning System (GPS) has been widely adopted in location based applications and services. The current GPS technology makes it possible to automatically track the position of an object in real-time [DOD01]. Some GPS devices in the market can achieve an accuracy of about 3 meters [Garm]. That is to say, we can locate the location of vehicles equipped with such devices to as accurate as ± 1 lane when they are running on road networks.

On the one hand, it is not uncommon nowadays to see application systems that monitor, retrieve, and manage transportation systems by equipping the vehicles with GPS devices and having them wirelessly connected to a processing server located in a data center. The owner of a private car can install a small GPS enabled device as shown in [Garm, Geod, Nova] and subscribe to services from a service provider to keep track of the current location as well as to retrieve the past driving routes. Transportation services such as Taxi companies and car shipping firms monitor and schedule their GPS-equipped taxicabs or shipping carriers. Public transportation services extend such systems even further so as to allow commuters to check real-time bus status.

On the other hand, researchers in the (road networks) traffic control and Intelligent Transportation Systems (ITS) areas have been spending great amount of efforts studying the patterns of vehicles' movement on roads, and developing laboratory traffic simulation systems.

Is it possible to improve, if not replace, the traffic simulation systems with the real-world allocation of vehicles location information and their movements from the existing

vehicles location information tracking systems? To do so, we must first integrate the data in these separate vehicles tracking systems into one database system. Besides providing the patterns of vehicles' movements for traffic simulation, an integrated vehicles location information system can also provide more other useful information, such as the real-time road condition, to the users; And it costs less in system maintenance than running individual tracking systems separately.

However, the security and privacy protection are one kind of issues and the technical difficulties in designing and implementing such an integrated database system are another issue that we need to resolve before this idea can be carried out and put in use. Even though the first kind of issues will not be completely omitted in this dissertation, our focus is on the technical issues in designing and implementing a real-time location information tracking and history retrieval system of massive numbers of vehicles.

Moving Vehicles are a subcategory of moving objects in general. Researchers in several areas, such as databases, motion planning, and pattern analysis, have been studying the properties and/or management of continuously moving objects. The main challenges of this topic for researchers in database area stem from the continuity of the objects' motion, which makes setting up a precise data model and building an efficient indexing structure difficult. To track up-to-date motion of GPS-equipped vehicles within a database system, real-time constraints need to be satisfied for update and query operations. This fact introduces even stricter requirements to the system performance in terms of I/O costs of these types of data accesses. Other than indexing structures that work well with past/historical data, new indexing structures that are efficient in performing real-time data accesses need to be found or designed.

In the recent years, extensive research efforts have been put in the field of Moving Objects (Database) Systems. Most of these works are either experimental or theoretical. A systematic solution has not yet been found. As described in [ES02, Jen02], the remaining research challenges in this area are mainly in data modeling, efficient indexing approaches, and Query Languages Standardization.

Targeted at building an application system to track the current location information and past movement of massive numbers of vehicles moving in region wide, the main efforts of this dissertation are put in finding an efficient indexing scheme for the real-time tracking and history retrieval of moving vehicles on road networks. A new approach, Segmented Time Associated to Partitioned Space (STAPS), is proposed for the building and manipulating of the indexing structures for continuously moving objects in general. Applying this general moving objects indexing approach, an efficient indexing structure is developed for moving vehicles on road networks, and will be used in the design and implementation of a vehicles location information database system.

1.1 Thesis Overview

The objects of our main concern in this dissertation are vehicles moving on road networks. The final goal will be to develop an integrated real-time vehicles' location information tracking system. The indexing structures to be developed should be able to efficiently process range queries and data updates, and meet the following criteria: (1). Vehicles location information is updated in real-time, (2). The past moving trajectories of the vehicles can be efficiently retrieved, and (3). Vehicles' near future movement can be predicted, provide known information like a vehicle destination.

The vehicles are equipped with GPS receivers and they periodically transmit their location information in the form of geo-location, speed, and the time when the data set is sampled. And the road networks are represented by street and highway road segments. The road segments information can be obtained in the form of geo-coordinated polylines, along with other properties, such as the speed limits [Nav].

The most frequently appearing queries to a moving objects database are range queries. The types of queries of our main concern include:

Q_1. Display the motion of all or a given sub-set of vehicles in real-time.

Q_2. Replay the motion of a given set of vehicles from past time t_1 to time t_2, t_2 can be past, now or near future time.

Q_3. Find the vehicles that have been on a given road segment between time t_1 and t_2, and display their movements during $[t_1, t_2]$.

Q_4. Retrieve the moving trajectories of the vehicles that have been in a given rectangular area R, between time t_1 and t_2.

Q_5. Return the vehicles that can reach a given point p, within time Δt.

Q_6. Predict the motion of given set of vehicles in the near future from t_1 to t_2.

In this dissertation research, a commonly used data model is adopted. Vehicles are represented by points. Their shapes and sizes are ignored. The moving trajectory of a vehicle is kept as a sequence of sample points of the form of (t, R_{id}, d, v). Where t is the time at which the vehicle is at this location, R_{id} is the number of the road segment that the vehicle is on, d is the current distance the vehicle is from the starting point of this road segment, and v the speed the car is running at this point. Road networks are represented

by a set of non-inner-intersecting one-directional road segments, each assigned a road segment number. A two-way road segment is treated as two one-way segments. The attributes maintained with each road segment include: the geo-coordinated layout of the road segment, its starting and ending points, total length, speed limit, and the road segments that are next to it at the end.

The core innovations of this dissertation research are in the indexing structures. In the proposed STAPS approach, time and space are treated differently. Space is divided into non-overlapping regions and indexed using appropriate space indexing structures. The time intervals during which objects moving inside each region of space are collected, indexed, and associated to this region.

Applying the STAPS approach to moving vehicles databases, the space is divided into road segments. The road segments are indexed using segment tree structure [Bent??]. And the sets of time intervals generated by vehicles running on road segments are indexed using balanced interval trees. A moving vehicles database system adopting this indexing scheme is flexible and highly scalable, since road segments, time intervals, and their indexing structures can easily be distributed to and processed by different processors, any levels of memories, and servers, when such distribution is needed to improve system performance.

An application system design is also developed in this dissertation, based on the proposed data and indexing structure. In the system design, a distributed system architecture with centralized organization is adopted. Issues such as the storage management, transaction and concurrency control mechanisms, and security are carefully balanced to maximize system performance, meet real-time requirements, and ensure

system reliability. The final data structure and implementations can be embedded into Sem-ODB [Rish92] or other existing Database Management Systems (DBMS) in the future.

The rest of this thesis is structured as follows. Chapter 2 addresses related work in the context of moving object and spatio-temporal database systems. Chapter 3 presents the data model, data structure, and indexing scheme that this dissertation proposes. Chapter 4 discusses the system architecture and design. Chapter 5 shows an example implementation, which applies the design into a system managing vehicles moving in the greater Miami area. Chapter 6 analyses the possibility of extending the data structure and the application. Topics in this chapter include extending the indexing structure to objects moving in higher dimensions and/or with fewer constraints, data warehousing the system, and embedding the system design as a data blade to Sem-ODB and other object-oriented or relational database management systems. Chapter 7 concludes the thesis and points out future work.

2 RELATED WORK

The storage and retrieval of moving objects are concerns of several research areas, such as spatio-temporal databases, moving objects databases, and motion planning, etc. Various proposals have been published, each addressing one or more sub-topics related to the whole problem. In the application aspect, there are several laboratory implementations, along with a big number of industrial real-time tracking systems covering single or enterprise wide vehicles.

The emphasis of this chapter will be put in reviewing the currently existing indexing schemes for moving objects. The current status and trends of other key issues in this area of research, which include the data modeling and data structures, query languages and application systems development, will also be briefly mentioned.

2.1 Data Model and Data Structures

Since the mid 90's when the continuously moving objects caught serious attention of researchers in the Database area, possible data models and data structures for managing this category of data have been almost exhaustively explored.

In Worboys' approach [Wor94], spatio-temporal objects are defined as spatio-bitemporal complexes. Their spatial features are described by simplicial complexes, and their temporal features are given by bitemporal elements attached to all components of simplicial complexes.

In [TSPM98], Theodoridis *et al.* proposed a discrete snapshot model. A spatio-temporal object *o* is represented by a time-evolving spatial object. Its evolution is

represented by a set of triples (o_id, s_i, t_i), where o_id is the object identifier of o and s_i is the location of o at time t_i.

Attribute time stamped models were proposed by Gadia and Nair in [GN93], Segev and Shoshani in [SS93], and Clifford, Crocker, and Tuzhilin in [CCT93]. This group of data models aim at gathering information about an object in one tuple and allow complex attribute values. These complex values incorporate the temporal dimension and are frequently modeled as functions from time into a value domain.

Another approach is to use linear constraints for modeling spatio-temporal data [GRS98]. This model supports efficiently representing and manipulating infinite point sets in arbitrary dimension. Time and geometry are treated as different and independent categories of data.

The approach presented in [ES02] by Erwig and Schneider supports an integrated view of space and time. A temporal version of an object of type α is represented by a function from *time* to α. A straightforward and instructive view of spatio-temporal objects is to visualize their temporal evolution as purely geometric, 1+*geometric*-dimensional objects. The spatio-temporal objects in 2-d space are hence taken as 3 dimensional geometric objects.

Erwig and Schneider also developed a complete set of spatio-temporal predicates to describe the spatio-temporal relationships between objects, viewing time as another spatial dimension [ES02]. The sptio-temporal predicates they proposed take into account temporal logic, point set theory and point set topology. This approach was moved forward by Forlizzi *et al.* in [FGNS00] and by Cotelo *et al.* in [CFGN01] with data structures and algorithms for the representing and implementation of basic types,

predicates, and operators. In [GBEJ01], Güting *et al*. presented how to incorporate this model with object-relational databases, which include how to put these spatial and temporal data types into the columns of a database, and how to form different types of queries in the format that an object-relational DBMS accepts.

In [WSXZ99, SWCD97] Wolfson, Sistla and their group introduced the MOST data model. In their data model, the concept of dynamic attributes to objects is introduced. The continuous change of an object's location is represented by a function of time. A dynamic attribute *A* is represented by three sub-attributes: *A.updatevalue*, *A.updatetime*, and *A.function*. The value of a dynamic attribute depends on time, and it is defined as: at time *A.updatetime* the value of *A* is *A.updatevalue*, and until the next update the value of *A* at time *A.updatetime*+ t_0 is given by *A.updatevalue*+ *A.function*(t_0). An explicit update of a dynamic attribute may change its value sub-attribute, or its function sub-attribute, or both sub-attributes. For an object moving in *2-d* space, its location attribute *L* can be modeled by two dynamic attributes *L.x* and *L.y*, each with its own update value, function and update time. They also modeled the uncertainty property of the data [SWCD98].

Vazirgiannis and Wolfson in [VW01-2] proposed a data model for moving objects on road networks, which is similar to the approach this dissertation adopts. Roads are represented by polylines. The geo-information along with the speed limit and other attributes regarding each segment of the road are associated with the road *id*'s. The trajectory of a moving object can then be represented by a polyline along space and time dimensions.

2.2 Indexing Structures

The most commonly used indexing schemes for temporal databases are multi-version and persistent tree structures. In the other hand, R-tree, quadtree and their variants are widely adopted in indexing spatial data. In general temporal database systems, the change or movement of data along time is discrete. And pure spatial data do not involve time. When the objects' movement is continuous, neither the general temporal nor the general spatial indexing mechanisms can be directly adopted for efficiently indexing the continuously changing spatio-temporal data.

Becker and colleagues in [BGOS93, BGOS96] proposed multi-version B-tree approach for indexing the data discretely changing as time evolves. A multi-version B-Tree keeps an index structure for the versions and another structure for the data within each version. Whenever an update -- either an insert, delete, or modify -- happens, a new version is generated. These two structures can use the same scheme, or different trees. Each node in the version index structure points to the root of the data entries for the corresponding version. A data entry inside one version can point to a node in another version if itself and all its descendants are the same in both versions. Varman and Verma in their work [VV99] presented their multi-version structures and showed how to apply the multiversion technique to database systems.

Multi-version trees and persistent data structures have been discussed in the work of other researchers. The main differences between various proposed schemes are in how the versions and data indexing are organized and maintained, how the overflow and underflow of a node are treated, etc. These techniques are appropriate for temporal

databases where the data change is discrete. But they are not suitable for indexing continuously moving objects.

Time Parameterized R-Tree or TPR-tree, was originally described in [ŠJLL00] by Šaltenis *et al.* Šaltenis and Jensen later on optimized this structure in [ŠJ02] and named it R^{EXP}-tree. The TPR-tree or R^{EXP}-tree is a balanced, multiway tree with the basic structure of an R*-tree. In this approach, Minimum Bounding Boxes (MBB) are time-parameterized. MBB changes dynamically as time evolves. This structure supports only queries falling inside time frame between *now* and *near future*. The term "*near future*" is defined by a querying window. MBB's are revisited upon updates when reaching update-time bounding rectangles. The concept of object's expiration was introduced. Queries on the past and future data that beyond the defined querying window are not supported.

TPR*-tree is proposed in Tao *et al*'s recent paper [TPS03]. It optimizes the original TPR-tree by employing a new set of insertion and deletion algorithms to minimize the query time cost, especially for the static point interval query q, whose (i) MBR has length $| q_{Ri} | = 0$ on each axis, (ii)VBR(Velocity Bounding Box)={0,0,0,0}, and (iii)query interval Q_T={0, H} where H is the horizon parameter.

In [HKTG02], Hadjieleftheriou *et al.* proposed partially persistent R-tree (*PPR-tree*) for indexing moving points, rectangles and objects of other shape. Emphasis is given to range queries for historical time t or a short period around t. Typical queries their approach supports are of form "find all the objects that appear in area S during time t." They view the plane where the objects move and the time dimension as a 3-dimensional volume. And objects are split along time dimension to reduce empty space and hence

reduce the size of MBR's. It can handle non-linearly moving objects as efficiently as those moving linearly.

In their paper ["PKGT02"], Papadopoulos *et al.* proposed a scheme for indexing linearly moving objects on the plane using dual transform. Objects with small velocity magnitude are transformed to Hough-X, which is (v, a) space. And objects with large velocity magnitude are transformed to Hough-Y, which has (n, b) coordinates. This way the MBR won't be too unreasonable for any objects, and hence the searching performance is efficient. Inside a *Horizon*, this scheme performs approximately the same as a TPR-tree. But outperforms the later when temporal part is out of the horizon.

Chon, Agrawal and Abbadi in [CAA02] gave a scheme to index the objects for queries within time period ΔT. It divided the time into n adjacent intervals, and the space into m segments. As time evolves, they shift the time domain forward. The trajectory of a moving point is represented by a polyline using a linked list. The headers of these linked lists are indexed with a hash table. The space-time grid is implemented using an array. Inside each cell (grid), only the identifier of intersected points is stored. They also provided performance experiments with different parameters including the size and thresholds of a grid.

In [BGZ97, BGH99, Bas99], Basch *et al.* proposed a kinetic data structure for indexing linearly moving points, which is based on the observation that the topological relationship between moving points changes only at some event (time) points even though their motion is continuous. Similar to the plane sweep technique which is widely adopted in Computational Geometry community, two structures are kept at any time for the indexing purpose. One is the order of the moving points, the other is an event queue.

Agarwal et al later proposed external kinetic B-Tree to handle the linear movement of points [AEG98].

Agarwal has contributed greatly to the indexing of linearly moving points, among which are the time-oblivious algorithms and time-responsive solutions. In his time-oblivious algorithms [AAE00], the core is the adoption of dual transform and the partition tree. The motion in 2-d space along time is viewed as the combination of motion in *x*- along time and that in *y*- along time. With both *xt*- and *yt*-, dual transform is applied and the trajectory of each moving point, which is a line, is hence transformed to a point in the dual space. For points moving in 2-dimentional space, build a 2-level partition tree on the points dual transformed from the trajectories of the moving points as follow: first level partition tree on the *xt*-space, with second-level on *yt*-space associated to nodes in some levels chosen by a certain mechanism. The total number of levels chosen is in $O(1)$, so the size of secondary tree is $O(n)$. With a set *S* of *N* linearly moving points in either 1-*d* or *2-d* space, an index on *S*, using *n* blocks of storage with each block of size B, can be built in $O(N\log_B n)$ expected *I/O*s such that the range queries can be answered in $O(n^{1/2+\varepsilon}+k)$ I/Os. Points can be inserted or deleted in amortized cost of $O(\log_B{}^2 n)$ expected *I/O*s. Where ε can be a very small but positive real number.

Another scheme proposed by Agarwal combines the kinetic range tree [AAV01] developed by Basch and an external range tree [ASV99] by Arge. Kinetic external range tree combines the kinetic B-Tree as a secondary tree to some levels of an external range tree. It answers range queries for given point of time on points moving in $\mathbb{R}^2$ in

$O(N^{1+\varepsilon}/\sqrt{\Delta}+k)$ *I/O*s using $O(n\log_B n/\log_B\log_B n)$ space. Where Δ is a parameter, $BN \le \Delta \le N^2$.

Agarwal's *time-responsive* algorithm for indexing N points moving in 1-dimension is to divide the space along horizontal (t-) axis into $\log_B N$ slabs. It is claimed that with high probability each slab contains O(N) vertices. Build index structure for each slab using the concept of *arrangement* [AAV01]. Complexities achieved for each slab is claimed as: Using O(N/B) space and O($N\log_2 N\log_B N$) I/Os to build the index, a *point* query can be answered in $O(B^{i-1}+\log_B N+K/B)$ *I/O*s. Where $1\le i\le\log_B N$ is the order of the slab from left to right.

The indexing schemes we have so far talked about are mainly for range queries. Nearest and reverse nearest neighbor queries frequently appear in some application as well. In [BJKS02], Benetis *et al.* proposed an algorithm for answering RNN and one for NN queries for continuously moving points in the plane. The algorithm returns the RNN/NN of a moving point for any time duration T, in the form of a set $\{NN_i, T_i\}$ / $\{RNN_i, T_i\}$, where $\bigcap_i T_i = \emptyset$, $\bigcup_i T_i = T$. The query point can be a member in or outside the set. A TPR-tree is used as underlying index structure. The RNN algorithm is developed from the observation by Stanoi *et al.* in [SAA00] that in each one of the six regions divided by three lines intersecting at the query point p there are at most two RNN points of p, and the total number of RNN points is not more than 6.

On objects moving in constrained space, there are several indexing schemes proposed as well. Kollios *et al.* in [KGT99-2] mentioned NN search where objects moving in road networks**.** They define this as 1.5 dimensions. Road networks are indexed using any traditional static index scheme (Spatial Access Method) for 2-d space. And the motion of objects is then 1-d. But no detail was given. In [PJ01], Pfoser and Jensen proposed an indexing scheme for objects moving in constrained environment. In such environments, some areas would never have any object appear. This scheme segments the query window based on the infrastructure of background where the objects are moving on, query the index on only those segmented sub-windows that possibly have objects appear at any time, and then evaluate the joint of result sets.

Pfoser and Jensen later developed an indexing scheme for objects moving on 2-dimensional constraining networks in [PJ03]. They reduced the 2-d space to one-dimensional, and built indexing structures on the reduced 1-d space and 1-d time. In this scheme, three kinds of mapping – all reduce the space dimensions from 2 to 1 – are required: the mapping of the fixed networks, objects' moving trajectories, and the queries. Instead of one 3-d indexing structure, the networks in (x, y) space and objects moving trajectories, which are reduced to (x, t) space, are indexed using separate indexing structures. Both indexing structures, for networks and for trajectories, are R-tree based. This indexing scheme supports only queries on past data.

Frentzos in his work [Fren03] proposed an indexing structure, called FNR-tree, to index the past trajectories of objects moving on fixed networks, which looks very similar to the indexing structures developed in this dissertation. The networks are indexed using an R-tree. Attached to each leaf node of this R-tree is a set of time intervals during which

objects moving over the space range defined by this leaf node. And each set of such time intervals are indexed using a 1-d R-tree.

The MON-Tree proposed by Almeida and Güting [AG04] also uses two-components to index the past trajectories of moving objects on networks. Routes, each defined by a polyline, are indexed using an R-tree, and another R-tree indexes the moving trajectories of the objects along each route.

To summarize the currently existing indexing schemes for moving objects, we can mainly classify them into four categories: (1) Temporal approaches, which can manipulate only discrete change of data; (2) The static R-tree based schemes, which require pre-processing and only support queries on past data; (3) The time parameterized R-trees, which are for queries on near future provide assuming objects moving linearly; and (4) The proposals from Computational Geometry community, such as the kinetic-based approaches, the dual transform scheme, the time oblivious and responsive approaches, are for future motion of linearly moving points.

The known existing indexing structures for objects moving in networks constrained space are all R-tree based, and they support past data only. To meet our application system requirements, which include efficient retrieval of past data, prediction of objects' near future position, and real-time data updates, none of these existing approaches well fits for the data indexing purpose.

2.3 Handling of Uncertainty and Imprecision

Wolfson *et al*. have systematically analyzed in their work [WCDJ98, SWCD98, WSXZ99, Wol02] the uncertainty and imprecision problem in moving object management systems. The location of a continuously moving object is inherently imprecise because, regardless of the policy used to update the database location of the object, the database location cannot always be identical to the actual location of the object. This inherent uncertainty has various implications on database modeling, querying, and indexing. For example, for range queries there can be two different kinds of answers, i.e. the set of objects that "may" satisfy the query, and the set that "must" satisfy the query. Thus, different semantics should be provided for queries. Another approach would be to compute the probability that an object satisfies the query. Although uncertainty in databases has been studied extensively, the new modeling and spatio-temporal capabilities needed for moving objects introduce the need to revisit existing solutions.

Additionally, existing approaches to deal with uncertainty assume that some uncertainty information is associated with the raw data stored in the database. How is this initial uncertainty obtained? For moving object database applications the question becomes how to quantify the location uncertainty, how to quantify the trade-off between the updating overhead and the uncertainty/imprecision penalty, how frequently should a moving object update its location, and how to handle the possibility that a moving object becomes disconnected and cannot send location updates?

In [WCDJ98, SWCD98, WSXZ99, Wol02] the authors extended their data model, query language, and indexing method to address the uncertainty problem. The data model

was extended by enabling the provision of an uncertainty interval in the dynamic attribute. More specifically, at any point in time the location of a moving object is a point in some uncertainty interval, and this interval is computable by the DBMS. Thus, the DBMS replies to a query requesting the location of a moving object *m* with the following answer A: "*m* is on route 698 at location (*x*,*y*), with an error (or deviation) of at most 2 miles". The bound *b* on the deviation (2 miles in the above answer) is provided by the moving object, i.e. the object commits to send a location update when the deviation reaches the bound. The FTL language is also extended. Two kinds of semantics, namely *may* and *must* semantics, are incorporated, and the processing algorithms are adapted for these semantics. The indexing method is also extended to enable the retrieval of both, moving objects that "must be" in a particular region, and moving objects that "may be" in it.

They also addressed the question of determining the uncertainty associated with a dynamic attribute, i.e. the bound *b* mentioned above. They proposed a cost based approach, which captures the tradeoff between the update overhead and the imprecision. The location imprecision encompasses two related but different concepts, namely deviation and uncertainty. The deviation of a moving object *m* at a particular point in time *t* is the distance between *m*'s actual location at time *t*, and its database location at time *t*. For the answer A above, the deviation is the distance between the actual location of *m* and (*x*,*y*). On the other hand, the uncertainty of a moving object *m* at a particular point in time *t* is the size of the interval in which the object can possibly be. For the answer A above, the uncertainty is 4 miles. The deviation has a cost (or penalty) in terms of incorrect decision making, and so does the uncertainty. The deviation (uncertainty)

cost is proportional to the size of the deviation (uncertainty). The tradeoff between imprecision and update overhead is captured by the relative costs of an uncertainty-unit, a deviation-unit, and an update-overhead unit. Using the cost model we propose update policies that establish the uncertainty bound *b* in a way that minimizes the expected total cost. Furthermore, we propose an update policy that detects disconnection of the moving object at no additional cost.

2.4 Query Languages

Generally, a query in Moving Objects Database applications involves spatial objects and temporal constraints. Traditional query languages such as SQL are inadequate for expressing such queries.

Sistla *et al*. in [SWCD97] introduced a temporal query language called Future Temporal Logic (FTL) for query and trigger specifications in moving objects databases. The language is natural and intuitive to use in formulating MOD queries, and it uses both spatial operators (e.g. object INSIDE polygon) and temporal operators (e.g. UNTIL, EVENTUALLY in the future).

In [MSI02], Mokhtar *et al.* investigated the appropriateness and efficiency of regular query languages on moving objects databases. They argued that traditional constraint query evaluation techniques are suitable for past queries, but neither for "continuing" nor the future ones. They also think that plane sweep technique can evaluate spatio-temporal queries efficiently.

In [VW01-2], Vazirgiannis and Wolfson also formalized the queries for moving objects on road networks.

2.5 Implementation and Applications

In the implementation and application aspects, both inside and outside the research community have various finished or ongoing projects on management of moving objects.

Pfoser and Theodoridis in [PT00] presented a trajectory generator, which extends their earlier work named GSTD, based on constraint environment (e.g. areas with buildings where objects are prevented from entering). Oporto, developed by Saglio and Moreira, is another scenario generator for fishing boats.

Tripod, as talked about in Griffiths *et al.*'s work [GFPH01] is a spatio-temporal database management system with full functions of a DBMS: data update, storage and retrieval, query processing, programming language access support, etc.

There are several real-time vehicles tracking systems, such as the ones described in [Garmin, Geod, Nova]. These applications are for personal, or enterprise uses. Either the numbers of vehicles are small, or the query involved relationship are concerned only of inside subsets. One of the most widely application area is for intelligent transportation systems, such as [Wash, Geog, Trim].

3 DATA MODEL AND DATA STRUCTURE

As observed and analyzed earlier, even though the motion of the vehicles is continuous, we can only keep the location and speed information for some discrete sampling points. So it is necessary to have a mechanism, which should be not too complicated and not too difficult to implement, to approximate the location and motion of a vehicle between the discrete sampling points.

Now the problem becomes: how do we decide the sampling points, along time dimension of course? The GPS enabled devices in our system update their location and speed data periodically. That is to say, the most accurate data we could get on the vehicles' motion is a set of data at discrete sampling points. So do we keep each and every data update from a vehicle in our database?

We know that there is a tradeoff between the data precision and system load. The shorter the sampling interval, the more accurate the motion of a vehicle is kept, and more data is kept in the database. Our goal is to use least possible memory space and simplest possible computation expense to achieve the best accuracy.

Even though the GPS enabled devices send vehicles' motion data periodically, the sampling time interval in our tracking system need not be uniform. Depending on the road situation where a vehicle is running on, the speed, and the change of speed, some motion updates from the vehicles may be omitted. Only those key points which affect the calculation of the motion in their nearby region are recorded and put into the database.

In the following sections of this chapter, we talk about the data modeling, data structure and indexing scheme that will be used in our system design and implementation.

3.1 Data Model

The various proposals on the data modeling sub-topic of the moving objects database area may look quite different from one another. The differences are mainly in the degree of formalization of these models. Beneath the variety of the appearance of these data models, almost all of the known proposals are basically similar in terms of a mathematical representation. Considering the maturity and reliability of these already approved and widely accepted existing approaches, there is no reason for us not to adopt a similar one.

The rest of this section will discuss the data model for road networks, vehicles and their movement, and time granularity.

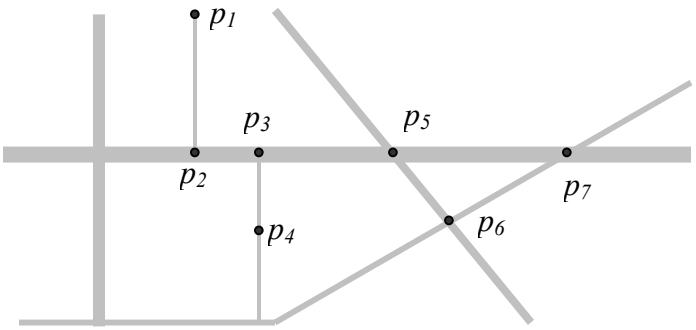
The road networks are represented by a set of directed and connected road segments. A *road segment* in the context of this writing is defined as a segment on the road such that:

(1) No intersection with other road segments exists inside the segment, and

(2) The ends, namely the start point and the end points, are either the end of a road segment or the intersecting point of two or more road segments.

A two-way road segment is treated as two separate one-way road segments. Figure3-1 explains the definition of the road segments in the context of this writing.

Each road segment is defined by a unique integer number R_{id}, the geo-location of the start point S_e and S_n, geo-location of the end point E_e and E_n, the total length of this road segment L, the speed limit SP, and the road segments that are geographically next to it.



$\overrightarrow{p_1p_2}$, $\overrightarrow{p_2p_3}$, $\overrightarrow{p_3p_5}$, $\overrightarrow{p_5p_7}$, $\overrightarrow{p_7p_5}$ are road segments, but $\overrightarrow{p_3p_4}$, $\overrightarrow{p_2p_5}$, $\overrightarrow{p_2p_6}$ are not.

Figure 3-1 The Road Segments

It is needed to mention that the road networks are not static. Road segments may look static when we look them in a short time interval. But they actually are not. Even though, compare to the movement of vehicles, changes to road networks are far less frequent, such changes do happen. A road segment can be removed due to reasons like construction. Roads can change their routes and new roads can be built as well. But such changes are discrete. So the time period at which a road segment is live needs to be attached, and a record for a road segment is defined as:

```
road_segment {
        Road_id :          integer,
        Start_easting :    real,
        Start_northing :   real,
        End_easting :      real,
        End_northing :     real,
        Length :           real,
        Speed_limit:       real,
        Time_of_birth:     time,
        Time_of_death:     time,
        Next:              *road_segment
  }
```

Using such a format, not only the road map is described by a complete set of road segments, but also the topological relationships among them.

The vehicles' attributes can be classified to three sub-categories: the static attributes, the discretely changing attributes, and the continuously changing attributes. These three categories of attributes are modeled respectively as follows.

(1) **The static attributes**. Since vehicles in our system are independent entities, each of them is assigned a unique integer ID, V_{id}. A vehicle's static properties, such as the make, model, size, etc., are included in this category. A complete set of such attributes and their data types is defined as:

vehicle_attributes_ static {
    V_{id}: *integer*,
    make: *string*,
    model: *string*,
    year: *integer*
}

(2) **The discretely changing attributes**. Vehicle' color, ownership and use could change over time, even though such changes are extremely infrequent. But the planned destination of a vehicle changes probably several times a day. It is not mandatory for a vehicle to update the data center on its change of destination. However, if a vehicle is a taxi cab, a shipping truck, or a bus, it is necessary to keep this information up-to-date at any time, for the sake of efficient scheduling. This set of attributes are:

vehicle_attributes_ discrete {
    ownership: {$owner_i$: *string*, t_i: *time*},
    color: {$color_i$: *string*, t_i: *time*},
    use: {use_i: *string*, t_i: *time*},
    destination: {$dest_i$: *location*, t_i: *time*}
}

Here location is a compound data type defined as:

location {
    zone: integer, //UTM zone
    easting: real,
    northing: real,
    Address: string
}

(3) **The continuously changing attributes**. Or called the dynamic attributes. All the attributes that together define vehicle's motion belong to this category. They are formally represented as:

vehicle_attributes_ dynamic {
    $Road_{id}$: *integer*,
    time: *time*,
    velocity: *real*,
    distance: *real*,
    easting: *real*,
    northing: *real*
}

Below are the analysis and explanation about the dynamic attributes.

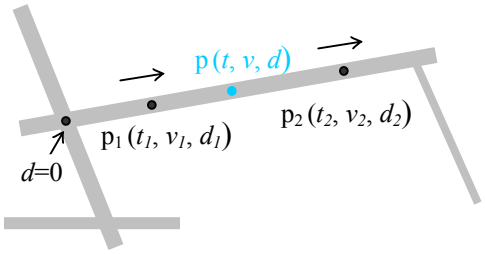


Figure 3-2 Approximate the Motion Between Two Sampling Points

The motion of a vehicle is defined by its location, direction and speed at the sampling points. And these values are recorded upon each update. The motion between two sampling points, as shown in Figure3-2, is approximated as follow.

To achieve a little better precision than using simply a linear function, we take the acceleration a into consideration. At any time t between t_1 and t_2, which are the time when the vehicle is at point p_1 and p_2 respectively, the speed v and distance d from the starting point of this road segment can be calculated as:

$$a = \frac{v_2 - v_1}{t_2 - t_1}$$

$$v = a(t\text{-}t_1) + v_1$$

$$d = v \bullet (t - t_1) + \frac{1}{2} \bullet a \bullet (t - t_1)^2$$

Taking the acceleration into consideration, on the other hand, reduces the size of data have to be stored persistently. This happens because we can drop those sampling points at which the acceleration is the same as their succeeding sampling point, ignoring if there is a speed change.

A vehicle sends its location and motion information in a certain frequency to the central station where such information is collected. But not every update from vehicle side is recorded into the database. If both of the following criteria are true, the update is ignored:

(1) The vehicle is on the same road segment as last update;

(2) The speed change since last update is within $\pm \delta_1 \times SP_r$, or the acceleration is within $\pm \delta_2$ comparing to its next update.

Where SP_r is the speed limit of this road segment, δ_1 and δ_2 are predefined small real numbers depending on the accuracy requirement of the system. In one case, a sampling point needs to be inserted between two updates from the vehicle. If two consecutive updates from the vehicle indicate that the vehicle has moved out of the one road segment and enter another, an approximated sampling point at the intersection of these two road segments needs to be calculated and added. The algorithm used to calculate the attributes are the similar to which is used to approximate the motion between two known sampling points on the same road segment.

Besides time *t*, speed *v*, distance *d* from the starting point of the road segment, the exact geo-location in the form of (*UTM-zone*, *easting*, *northing*) is stored with each record as well.

A system parameter *chron* , which is the smallest time granularity, is predefined. It can be small as $\frac{1}{10}$ or $\frac{1}{100}$ second, depending on the system accuracy requirement. But for a vehicle tracking system, it should not exceed 1 second. Considering a car moving at 40*mph*, the distance it moves in 1 second is about 17 meters. The average length of a road segment is in the order of 100 meters, and 17 meters per update means only about 5 updates per car per road segment. This potentially could introduce data inaccuracy. On the other hand, it is a waste of resource if we set it too small: if data updates come in every $\frac{1}{10}$ second and the distance between two consecutive updates would be 1.7 meters, which is far less than the standard GPS error which is about 3 meters, and hence is not necessary. Due to the introduction of the term chron into our system, time *t* can be stored

in and treated as type integer. It will greatly reduce the complexity of maintaining the indexing structure.

3.1.1 Handling of Imprecision

The imprecision in our system is mainly caused by two factors. One is the difference between our data model and the real world. The other is caused by the GPS or communication system (between the vehicles and the data center). Among the errors caused by the GPS or communication system, there could be systematic errors and random errors. For the systematic errors, we can correct them by measuring the difference between the correct data and the data we get from the GPS system, and adjusting the number accordingly.

For the random errors, the method we correct them with is as follows. At each sampling point, we compare the location and speed with the data at its preceding and its succeeding sampling point. If it is reasonable we take it; we drop it otherwise.

So far we have talked about the basic data types and the handling of imprecision in our system. A complete schema on the classes of data in our system is shown in Appendix A. In the next two sections we are going to detail this data model with data structures and indexing scheme.

3.2 Data Structure

We observed that the data in our system consist of continuously changing data, discretely changing data, and static data. To make our system efficient, they should be manipulated differently.

Recall that a road segment is defined as a tuple: ($Road_{id}$, $Start_{easting}$, $Start_{northing}$, $End_{easting}$, $End_{northing}$, Length, Speed_limit, *Time_of_birth*, *Time_of_death*, *Next*). Every attribute in such a record is required and their size is uniform, except for the last one. The last attribute defines the road segments next to the current road segment according to their geographical layout. In most of the cases, there are 3 or 4 such road segments: go straight, turn left, turn right, and sometimes U-turn. It is reasonable to set 4 cells for the *Next* attribute, so that the size for every record is the same hence we can use an array to store this set of data. As for the special cases where there are fewer or more than 4 next segments, assume the number is x, rules are given as follow:

(1) when x is smaller than 4, fill in the (x+1)th through 4th cells with 0;

(2) when x is larger than 4, the first two cells are the same as in normal case, set the 3rd cell to 0, and the 4th cell the pointer to the address where the 3rd through xth are stored and there NULL is put at the end.

Two road segments located on the same road but reversed in direction share all the information except for attribute *Next*. Redundancy is introduced if we store all the information for each directed road segment. Instead we can use two separate arrays to store attribute *Next* and the rest attributes. Now a cell x in the attribute array corresponds to two cells $2x$ and $2x+1$ in the *Next* array. And the *Start* and *End* attributes in cell x of the attribute array are for cell $2x$ in *Next* array. They should be reversed when applying to cell $2x+1$.

The road networks information is needed to locate the road segment for a given point or street address. It is also needed to find paths, answering range queries in some cases, and to display vehicles' motion. We choose a segment tree as the indexing structure for

road segments, which will be discussed in detail in following section. Data representing road networks incur discrete changes. A temporal data management scheme needs to apply to this set of data for indexing purpose. A multi-version tree structure, as described in section 2.2, is adopted to keep track of the old versions as well as the most current layout of the road segments.

The static and the up-to-date discretely changing data on vehicles can be stored using a normal table in a relational DBMS or as instances of the same category in an OO or semantic DBMS. The history of the discretely changing data can be manipulated similarly in most of the cases. If there are special needs, we can use a multi-version tree structure for the indexing of these data, the same way as we handle the data on road segments.

Our main concern for the vehicles related data is on their motion. Figure 3-3 shows a file keeping the motion of a vehicle during the time frame *time_span*. It can be divided into three parts: the header, meta data or index, and the content data. The header is of a uniform size for all vehicles. It starts with the Vehicle's ID number. The second attribute, *time_span*, is in practice composed of two attributes: *start_time* and *end_time*. The total size assigned to keep meta data and the total size of the file is kept in the header as well. Meta data contains a set of records in ascending order of time for indexing purpose. It keeps a mapping from time frame to the offset inside this file where the motion records for this time frame are located. And the content is a set of equal sized records keeping the data for this vehicle at all the sampling points.

The reason that each vehicle's motion history is kept separately is that frequent queries are of the form "re-play the motion of one or several vehicles during time t_1 to t_2".

Even for the range queries on a rectangle area or a set of road segments for a time frame, the motion of each included vehicle during that time frame needs to be retrieved altogether.

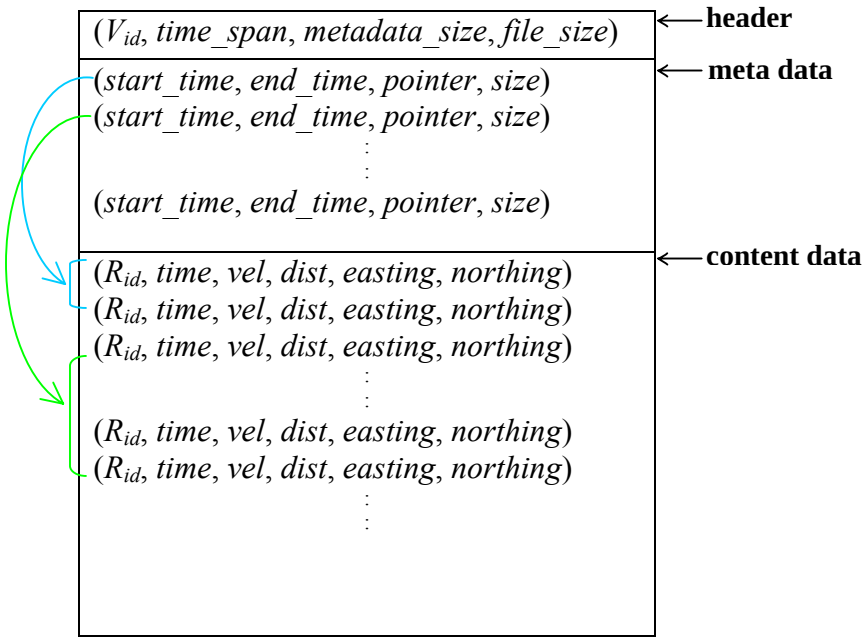


Figure 3-3 A File Keeping a Vehicle's Motion

It seems that vehicle's motion data is well organized for searching and query purposes, since an indexing scheme already exists inside the storage of the moving trajectory of each vehicle. Even though it is helpful for locating a vehicle's position within given time frame, such an data organization within data on separate vehicles is not enough to efficiently answer all types of queries, especially for those range queries based on space and/or time span. Efficient indexing schemes are necessary for both the road segments and vehicles' motion data. In the next section we will discuss these indexing structures.

3.3 Indexing Structure

Time and space dimensions are related in defining and representing objects' motion. But they don't play equivalent role, especially during the processing of queries on the movement of the objects. Previous research showed that better performance can be achieved when time is treated differently from space dimensions. During this dissertation research, the author also observed that demands on time and space are different in most of the frequently appearing queries.

One of the main concerns of the queries to a moving objects database is the behavior and properties of the vehicles in certain territories. To reflect the continuity of the objects' movement and efficiently handle the imprecision and uncertainty between the sampling points, the moving trajectory of an object can be divided into consecutive segments, each representing its movement in the same territory.

Based on the above analysis and observation, the ***Segmented Time Associated to Partitioned Space (STAPS)*** approach is proposed for the building of indexing structures for moving objects databases. Figure 3-4 illustrates how the STAPS approach works: First the space inside which the objects move is partitioned into basic units called space partition. Space partitions are independent of one another semantically. The space partitioning is flexible, and it is mainly decided by the frequencies of the different types queries the database system supports. Then for each object, time is segmented into time intervals, during each of which the object is inside one space partition. And the time intervals are put into different sets, according to the space partition that segments them.

Each set of time intervals is indexed and associated to the corresponding space partition. And the space partitions are indexed separately from the time intervals. The

indexing structure for the space partitions, and that for each set of time intervals are independent of one another; they can be the same, similar or completely different ones.

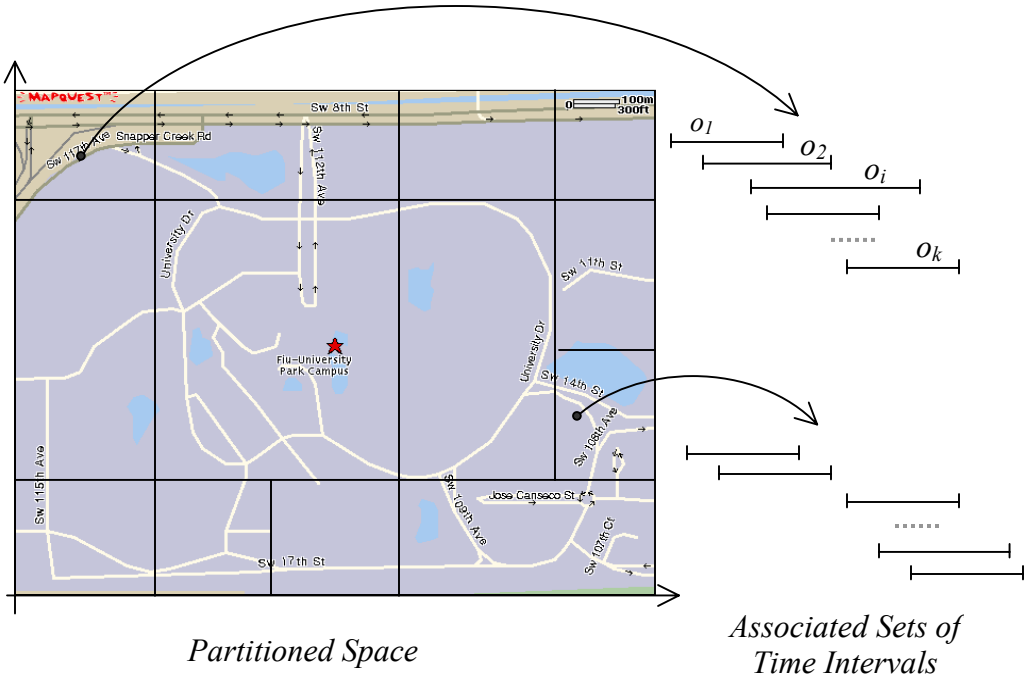


Figure 3-4 The Space Partitions and Time Segments

The advantages the STAPS approach provides include:

- Reflecting and representing the continuity of objects movement.
- No restrictions and presumptions on objects motion.
- Limiting uncertainty and imprecision caused by the data processing procedures.
- Flexibility in adjusting and balancing storage and I/O costs on different data structures and operations to meet each system's specific requirements.

Also notice that there is no restriction on space partition algorithms as long as each point in the space is assign to and only to one partition. The indexing structures for both time intervals and space partitions can be any appropriate kinds.

In the rest of this section, indexing structures and the related algorithms will be discussed in detail. Using STAPS method, time is segmented to intervals during which a vehicle is moving on a road segment. Space is partitioned into single directed road segments. And the indexing structures will include the structure to manipulate the road segments and the time intervals associated to each road segment. The indexing structures for both the time intervals (time segments) and those for the road segments (space partitions) are inspired by and based on the segment tree originally designed by Bently [Bent77] and introduced in [BKOS00]. A balanced interval tree is used to index the time intervals during which the vehicles are moving on the same road segment, and there is a time interval tree associated with each road segment. A modified segment tree is used to index the road segments.

To answer queries involving vehicle's future motion, path finding and computing may be necessary. A* heuristics are used in path computation for the moving vehicles with known destination.

The continuity of the vehicles motion results in inherited uncertainty and inaccuracy for the current computer systems to represent their movement. Uncertainty is hence introduced in answering the queries. Corresponding to this fact, probability is introduced to the results of queries. An extra step other than in the traditional databases case is

necessary to finalize the query results. The details on the step of finalizing the query results will also be addressed.

3.3.1 Introduction to Segment Tree

Segment tree was discovered by Bently [Bent77]. It was originally designed to answer range queries on a set of line segments in 2-dimensional space. In this sub section, we will give the algorithms and do complexity analysis. However, the definition and descriptions of segment tree structure are based on Berg *et al*'s book [BKOS00].

The problem can be described as follow: given a set of n non-intersecting line segments $\{[(x_{11}, y_{11}), (x_{12}, y_{12})], [(x_{21}, y_{21}), (x_{22}, y_{22})], \ldots, [(x_{i1}, y_{i1}), (x_{i2}, y_{i2})], \ldots, [(x_{n1}, y_{n1}), (x_{n2}, y_{n2})]\}$ in 2-d space, to answer range queries of the form "return all the line segments in the given set that intersect the vertical line segment defined by $[(x, y_1), (x, y_2)]$". In this sub-section, we explain what is a segment tree and how it works. The following sub-sections will show how to use the traditional segment tree to index time intervals and road segments and efficiently answer the range queries to our database system.

The way a segment tree works is based on the observation that if we divide the space in parallel to the y-axis into consecutive slabs at the x-value of every end point of all segments, the partial segments inside each slab don't intersect one another and hence they can be ordered. Figure 3-5 shows how this works.

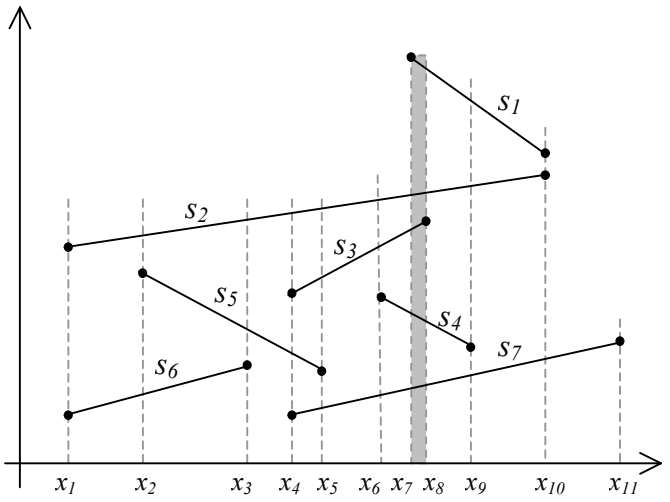
More precisely, if we project the 2-d segments into the x-dimension, they become one-dimensional intervals. The end points of these intervals divide the 1-d space into consecutive segments such that an interval either does not fall in a such segment

(ignoring the end points) or falls in it in full. The 2-d space can be segmented into slabs the same way. The segments inside each slab can be ordered by their *y*-values because these segments don't intersect one another. A binary tree structure is the easiest to index the canonical subset of segments falling in each slab. Figure 3-5(a) shows how the space is segmented into vertical slabs, and Figure 3-5(b) is an example binary tree built to index the canonical subset inside a slab.

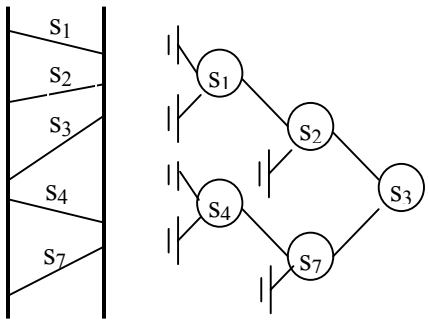
The consecutive intervals in *x*-dimension that defines the slabs can be indexed with a binary tree too, since they don't intersect each other from the indexing point of view. The leaf nodes of this binary tree each corresponds to an *x*-interval and a subset of segments whose *x*-projection falls in this interval. A pointer links the binary tree on the canonical subset of each *x*-interval to the leaf node of the binary tree on the *x*-intervals.

Such a tree structure as shown in Figure 3-5(c) is named a segment tree. The subset of segments associated to the leaf nodes of the segment tree is the canonical subset of the corresponding interval.

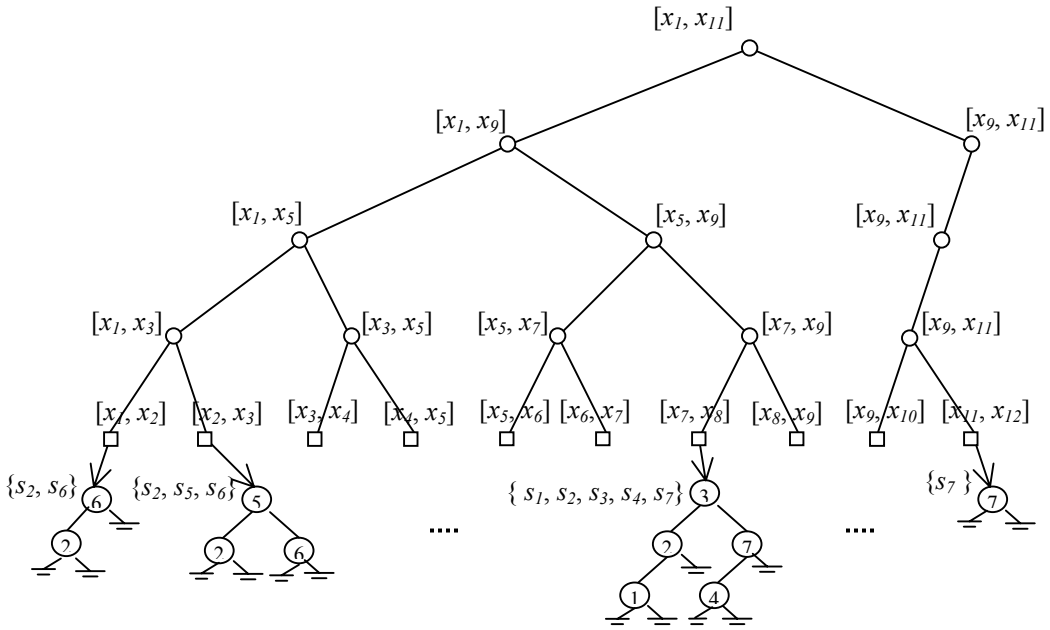
To answer the typical queries the segment tree is designed for, e.g. return the segments which intersect the segment $[(x, y_1), (x, y_2)]$, search begins from the root down to the leaf node where the given x is inside its x-interval. And a range query on $[y_1, y_2]$ is performed over its canonical subset then. The I/O cost for a query is $O(\log n')+O(\log n+k)$, where k is the total number of segments being reported, n is the total number of segments, and n' is the total number of x-intervals which could be up to n^2. As a result, the I/O complexity is $O(\log n^2)+O(\log n+k)$ which evaluates to $O(\log n+k)$.



(a) Divide the Space into Vertical Slabs



(b) Indexing the Canonical Subset in the Grey Slab in (a)

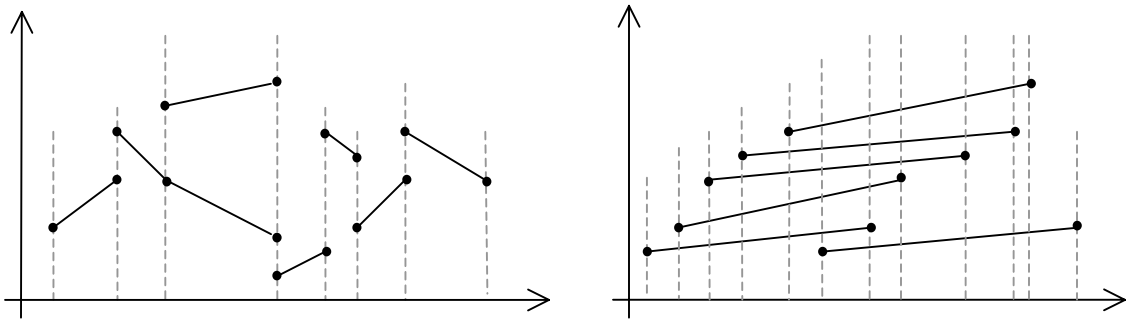


(c) Segment Tree for the Set of Segments Defined in (a)

Figure 3-5 Building a Segment Tree

Notice that the searching could involve two paths if x falls on the boundary of two consecutive intervals. The extra path could be avoided if we define the x-intervals as $(-\infty, x_1)$, $[x_1, x_1]$, (x_1, x_2),$[x_2, x_2]$, …, $[x_{n'}, x_{n'}]$, $(x_{n'}, \infty)$. However this way it would require double space for the tree structure as it is shown in Figure3-5(c), where intervals are defined as closed ranges $[x_1, x_2]$, $[x_2, x_3]$, …, $[x_{n'-1}, x_{n'}]$.

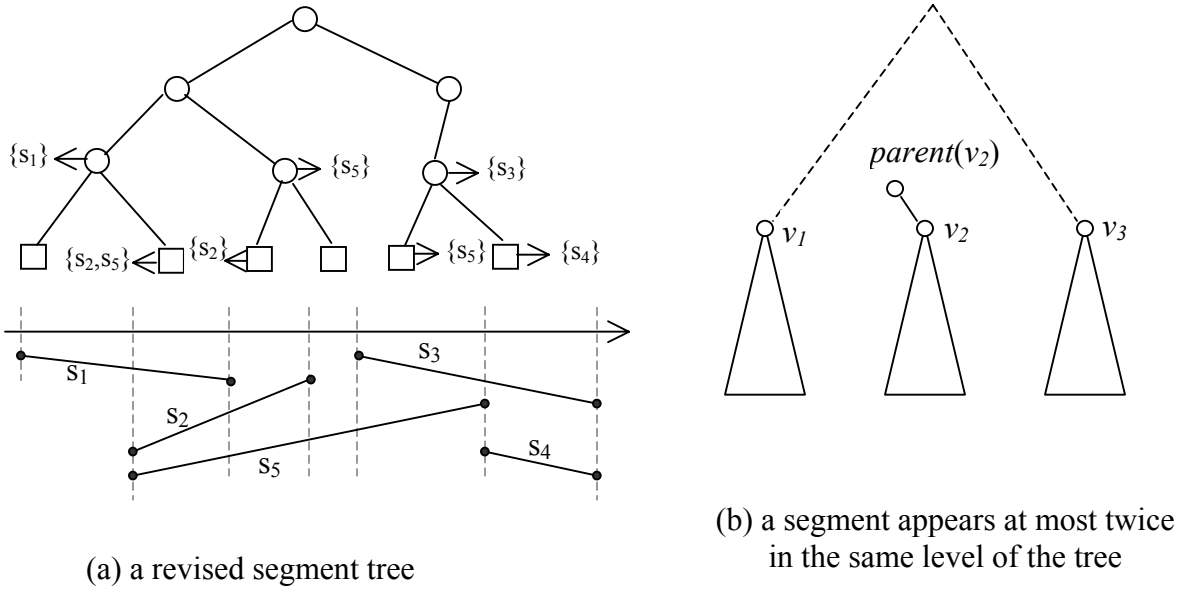
The space complexity of a segment tree is composed of the space used for the main tree and that for the canonical subsets. When the x-values of the given set of segments cover the complete range $[x_{11}, x_{n2}]$ and they do not partially overlap one another, as shown in Figure 3-6(a), the least of total space of $O(n)$ is required. The other extreme is when every segment overlaps all the other segments but none of them overlap completely when projected to x-axis, as shown in Figure 3-6(b). In such case the space complexity goes straight up to $O(n^2)$, $O(n)$ for the main tree and $O(n)$ for each of the n leaf nodes.



(a) No Partial Overlap Among Segments (b) Each Segment Overlaps all the Others

Figure 3-6 Two Extreme Layouts of the Segments

The space cost in the second case is obviously undesirable. Fortunately, there is a way to prevent it from happening by modifying the tree structure a little differently from its original version. This is done based on the observation that the canonical subset, instead of being associated to only the leaf nodes, can be associated to the non-leaf nodes. A subset of segments can be associated to any node whose covered range is inside the *x*-ranges of these segments. We put a segment to the canonical subset associated to the highest possible level of the tree when it is applicable.



(a) a revised segment tree

(b) a segment appears at most twice in the same level of the tree

Figure 3-7 Revised Segment Tree

The example of a revised segment tree is shown in Figure 3-7(a). If a segment appears at the same level of the tree in *m* nodes where *m* is larger than 2, its existence in the middle *m-2* nodes can be lifted to a higher level. This is shown in Figure 3-7(b). Since each segment is associated to at most two nodes in the same level in a revised segment

tree, a segment can at most be associated to 2(log$n$ – 1) nodes. The total number of basic intervals for n segments is at most 2n –1. So the total number of nodes in the main tree is at most 2(2n –1). The total size of all the canonical subsets is at most n•2(log$n$ – 1). The total tree size is hence at most 2(2n –1)+2n (log$n$ – 1) = 2n(log$n$+1)-2, which is O($n$log$n$).

Theorem 3.1 *A segment tree for a set S of n non-intersecting segments in the plane uses O(nlogn) storage and can be built in O(nlog^{2}n) time*[1]*. It takes O(logn+k) time to report all the segments that intersect a given vertical segment [(x, y_1), (x, y_2)], where k is the number of reported segments.*

Proof The proof of Theorem 3.1 includes three parts:

(1) The proof of the space complexity lies in the earlier analysis.

(2) The time cost in building a segment tree is composed of the time spent to build the main tree structure and that to build the binary trees for each canonical subset. To build the main tree, the order and hence the intervals of the segments in x-dimension need to be calculated. This takes O($n$log$n$) time. Building the tree then takes time O(n). Sorting each canonical subset of size v takes $v$log$v$ time. The total size of all the canonical subsets, i.e. Σv, is at most 2$n$log$n$. So the total time cost for building the binary trees for all the canonical subsets is Σ$v$log$v$ < (Σv)•log$n$ < (2$n$log$n$)•log$n$ = O($n$log^2n). The sum of the above steps yields O($n$log^2n).

[1] In [BKOS00] it is claimed that the preprocessing time can be improved to O($n$log$n$) by maintaining a partial order of the segments while building the segment tree.

(3) Time complexity for queries of the given type is the time taken to go through a path from the root to a leaf node, picking all the segments that fall in the *y*-range of the query from each associated canonical subset. The height of the main tree is O(log*n*). The height of the binary trees on the canonical subsets is also O(log*n*). Plus *k* answers are picked. The sum of the above three parts yields $O(\log^2 n+k)$.

Below are the algorithms for building and searching a Segment Tree.

Algorithm BuildSegmentTree(S)

Input. A set of disjoint line segments in the plane.

Output. A segment tree.

1. Sort the *x*-values of the end points of all the input segments into a sorted list. A set including all the segments which have an end point of the same *x*-value is associated to each of the corresponding item in the list. Then find the order of the left and right *x*-value of each segment in the sorted list.
2. Build a balanced binary tree, namely the main tree, out of the *x*-intervals formed by the sorted list obtained in step 1.
3. Since it is now known the order of each segment's left and right end points' x-value, it can be easily calculated to which nodes in the main tree a segment should be included in their canonical subsets. Set all the canonical subsets properly.
4. Build a binary tree for each canonical subset according to their vertical (partial) order.
5. Return the root of the tree built in step 2.

Algorithm SearchSegmentTree(Γ, l)

Input. A segment tree Γ and a vertical line segment l.

Output. All the segments in the segment tree that intersect the vertical line segment.

1. $S \leftarrow \Phi$.
2. Search the main tree of Γ to locate the path that the x-value of l is included.
3. Starting from the root of Γ down to the leaf node in the path, search each associated binary tree to locate the y-range of l. Add these segments whose y-value at l_x is inside l to S.
4. Return S.

A segment tree structure is suitable for indexing static input data set. Insertion and deletion in segment trees are expensive and cannot be well supported if they occur frequently. The revised version of the segment tree trades searching time complexity for space, comparing to the original version. With the original segment tree, it costs only O(log$n$+k) time to report all the segments that intersect a given vertical line segment.

3.3.2 Indexing Structure

The indexing structure in a location information tracking system for vehicles moving on the road networks mainly include two parts: indexing the road networks and vehicle's moving trajectory. The distance and relative position between different vehicles and that between a vehicle and a static location are determined by vehicles' geolocation as well as the layout of the road segments. Some forms of queries, such as a range query involving a given area in the map, may need to find the road segments inside the given area before

the query can be answered. On vehicle's motion, since each single vehicle's moving trajectory is already well organized for range queries on time span, it makes sense to build a main indexing structure over all of the vehicles' trajectories for queries on space/location range. To do so, the time span of each vehicle running on a road segment is designed as the basic unit of data, and a corresponding indexing structure, namely a time interval tree, is built and associated to each road segment. Queries on given space range and time span are answered by first look into the road network and get all the road segments inside the given space range, and then go to the time interval trees associated these included road segments to get the vehicles running on these road segments at the given time span.

Coincidently, the segment tree structure is adopted for both indexing structures. Details on building and maintaining these indexing structures and how the search operations are carried out are discussed in subsections 3.3.2.1 and 3.3.2.2. The corresponding algorithms are given, and the space cost and I/O complexity for each operation are analyzed as well.

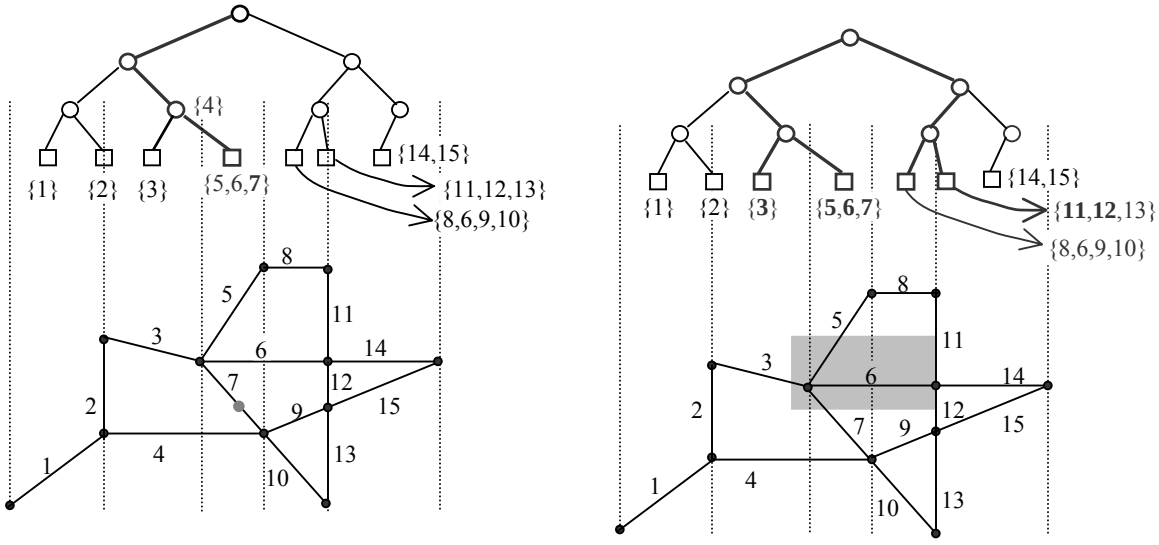
3.3.2.1 Indexing the Road Segments

The term "road segment" in our system was defined in section 3.1. For the sake of simplicity, whenever there is an intersection between two physical road segments, both of them are divided into two segments to store in the database system. The dividing point is the point of intersection. This applies to the case when there is no intersection viewing from three-dimensional space but there is one when projecting the roads to 2-d plane.

Example of such case is when a highway "intersecting" a local road, where in 3-d space there is no intersecting point at all.

Now the road segments in our system almost meet all the requirements for building a segment tree over them. The only two things that need special care are the coincidence of the end points of the segments, and the segments that are vertical themselves.

For the coincidence of two or more end points at the same coordinate, there is no need to re-modify the indexing structure as described in subsection 3.3.1. The specialty is with the search operation for query answering. When answering a point query, all the segments intersecting at the point should be returned if the query point is right on the intersection. However, when answering a rectangular range query, the segments that have only one point inside the range are excluded.



(a) Locate a Point (b) Locate a Rectangular Area

Figure 3-8 Searching a Segment Tree

As for the segments whose layout is strictly vertical in the plane, a new x-interval is added at the phase of processing the x-intervals while building the segment tree. This new x-interval is a closed range starting at the x-coordinate of a vertical segment and ending at the same value. So the x-intervals will appear as $[x_1, x_2]$, $[x_2, x_3]$, …, $[x_{i-1}, x]$, $[x, x]$, $[x, x_i]$, …, when there is a vertical segment $[(x, y_1), (x, y_2)]$. And a vertical road segment is included in the canonical subset of only the corresponding leaf node.

The preprocessing of building tree is basically the same as in the case of the standard segment tree. The time and space complexity for building tree is analyzed earlier, and they remain the same.

Figure 3-8 shows how to locate a point and a rectangular area and return the intersected road segments. The search algorithms and the I/O cost are explored as follow.

Algorithm SearchPoint(q, Γ)

Input. A given point q in the plane and a road segment tree Γ.

Output. All the road segments that intersect point q.

1. $S \leftarrow \Phi$.
2. Locate the path(s) in Γ of the nodes whose x-range include q_x.
3. Along the path(s) found in step 2, search the y-range against q_y in each of the canonical subset. If point (q_x, q_y) is on a segment, add the segment to S.
4. Return S.

The maximum number of possible paths located in step 2 is 3. It happens when q_x is the end point of multiple segments and there is a vertical segment whose x-value is q_x. Going through one path and the associated subsets takes $O(\log^2 n)$ I/O accesses. Plus reporting k results, it is $O(\log^2 n+k)$. The worst case I/O cost is $O(3\log n+k)= O(\log n+k)$, where k is the total number of segments reported.

Algorithm SearchRectangle(R, Γ)

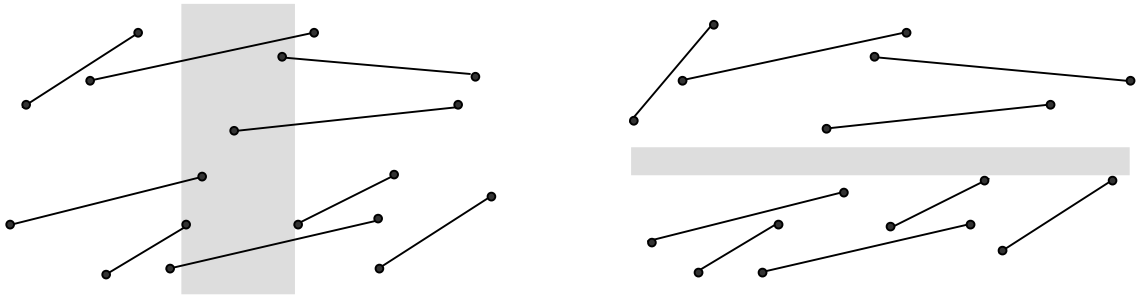
Input. A rectangle R in the plane defined by (x_1, x_2, y_1, y_2) and a road segment tree Γ.

Output. All the road segments that intersect the rectangular area defined by R.

1. $S \leftarrow \Phi$.
2. Locate the range in Γ of the nodes whose x-range covers or intersects $[x_1, x_2]$. This will result in a set of paths from root to leaf nodes in Γ.
3. Along each of the paths found in step 2, search the y-range against $[y_1, y_2]$ in their canonical subsets. If a segment intersects the area (x_1, x_2, y_1, y_2), add it to S.
4. Return S.

In Theorem 3.1 we know that it costs $O(\log^2 n+k)$ I/O operations to report all the segments intersecting a vertical query segment. Does the I/O cost remain the same order with algorithm SearchRectangle? It remains the same order when either the x-range of the query rectangle is very small comparing to the x-range of the segments in the set, or the majority of the segments whose x-range intersect the x-range of the query window actually intersect the query window in 2-d plane. Refer to Figure 3-9(a) for illustration.

However, cases exist where the I/O cost for SearchRectangle goes beyond $O(\log^2 n+k)$. Look at an extreme case as shown in Figure 3-9(b), where the query rectangle crosses the whole range of the *x*-values of the segments but its *y*-range is small and the rectangle doesn't intersect any of the segments. In this case, the search operation goes through all the nodes in the main tree and retrieves $O(\log n)$ depth in each binary tree on canonical subset. The number of reported segments is 0. Total complexity in terms of I/O accesses goes up to $O(n\log n)$.



(b) Desirable Case (b) Worst Case

Figure 3-9 Different Cases for Algorithm SearchRectangle

The worst case shown in Figure 3-9 won't happen in the road segments, since the road networks are connected. But we can't prevent similar situation from appearing. However, limiting the scope to road networks only, the size of the querying rectangle decides the number of the reported segments in the majority cases. Hence the average query cost is $O(\log^2 n+k)$, and in worst case it is $O(n\log n)$.

Putting all the above analysis together, we conclude the following Corollary:

Corollary 3.1 Using a segment tree to index the road segments in the plane, it costs $O(\log^2 n+k)$ I/Os to locate all the road segments containing a given point; the average cost for locating road segments intersecting an axis-parallel rectangle is $O(\log^2 n+k)$, with the worst case of $O(n\log n)$[1]. Such a segment tree can be built in $O(n\log^2 n)$ I/Os, using $O(n\log n)$ storage.

$O(n\log n)$ is the optimal space complexity a data structure can achieve for 2-dimensional range queries without joining operation involved. The preprocessing time on building tree is not a concern in this case since it is basically built once last forever, plus $O(n\log^2 n)$ isn't bad at all. The query performance is of our most concern. The I/O cost of $O(n\log n)$ in rectangular queries answering in the worst case is strongly undesirable, why is this approach still used to index the road segments? The reason lies in the probability of the occurrence of the worst cases. Such case appears only when the query area falls in a lake, a mountain, marshes or alike where there are no roads constructed, but not in urban area. It rarely occurs in city regions. If larger areas are covered, an auxiliary data structure can be added to handle these areas where none or rare roads exist.

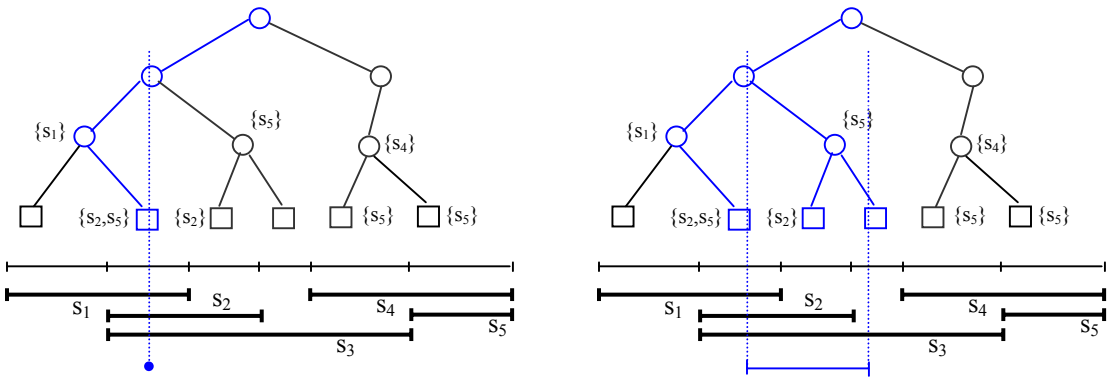
The segment tree structure has another advantage for road segments in keeping the history. Multi version tree can be easily adopted to reflect the discrete change of the road segments.

[1] In [BKOS00] it is claimed as $O(\log^2 n+k)$, which is inaccurate.

3.3.2.2 Indexing Vehicle's Motion

The basic unit of vehicles' movement in our indexing structure is measured by the duration while a vehicle running on the same directed road segment. The structure of a segment tree is used to index the time intervals during which a vehicle is on the same road segment. A one dimensional segment tree on time intervals is called a ***Time Interval Tree*** in the scope of this dissertation.

Since time intervals are one-dimensional, there is no need to sort or build an indexing structure for the canonical subsets associated to each node in the time interval tree. Point queries can be answered in $O(\log n+k)$ I/Os where k is the number of the reported time intervals. And range queries can be answered $O(\log n+k)$ I/Os too. The searching path(s) for point and range queries are shown in Figure 3-10. The search algorithms are similar to the vertical line segment and rectangle window search in the revised version of the 2-d segment tree.



(a) Locate a Point (b) Locate a Range

Figure 3-10 Answering Point and Range Queries in Time Interval Tree

The online update of the time interval tree in a vehicles location information tracking system include only one kind of operation – insertion[1]. The insertion of a time interval is done in two steps: (1) At the time of entering a road segment, the start of a time interval is sent to and recorded in the tree; (2) At time of leaving a road segment, a time interval is explicitly added to the tree structure. Notice that the time point or time interval to be inserted is always at the right most in the *x*-axis, or they are of the latest comparing to the existing data in the tree.

Let's first clarify the data structures in a (balanced) time interval tree. The data types involved in a time interval tree include node, tree, and an auxiliary data structure called a node locator for the purpose of indicating the most current (time) node in the tree. Details are as follows:

- A node, leaf or non-leaf node, in a time interval tree is defined by the time span from time point *a* to time point *b* and a set *S* of V_{id}'s of the vehicles that share the same time span [*a*, *b*]. It appears as ([a, b], {V_{id}, …, }).
- A time interval tree is a doubly linked balanced binary tree. Each node in the tree has three pointers to link their parent node, left node and right node.
- A node locator contains a pointer *r* to a node in the time interval tree, time point *t*, and the level *l* of the pointed node in the tree. The level of a node is ordered from the leaf nodes up, and the level of the leaf nodes is 0.

[1] Offline update could incur delete and/or modify operations at time of data recovery. The delete and modify operations are very rare even if they do appear, and the performance of these operations won't affect the overall performance of the data structure.

Algorithms for insertion in a time interval tree are shown as below.

Algorithm InsertTimePoint(t, Γ, P)

Input. A new time point, a time interval tree Γ and Γ's most recent record P.

Output. Updated time interval tree Γ and its most recent record P.

1. If ($P.t < 0$) $P.t \leftarrow t$;
2. Else if ($t > P.t$) AddNewLeaf (t, Γ, P) ;
3. Return.

Algorithm InsertTimeInterval([a, b], V_{id}, Γ, P)

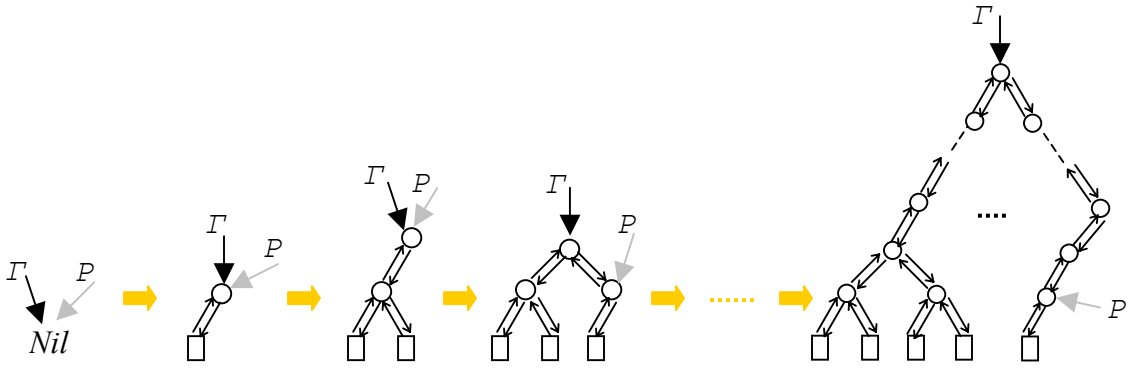
Input. A time interval [a,b], V_{id} of the vehicle that generated [a, b], a time interval tree Γ and its most recent record P.

Output. Updated time interval tree Γ and its most recent record P.

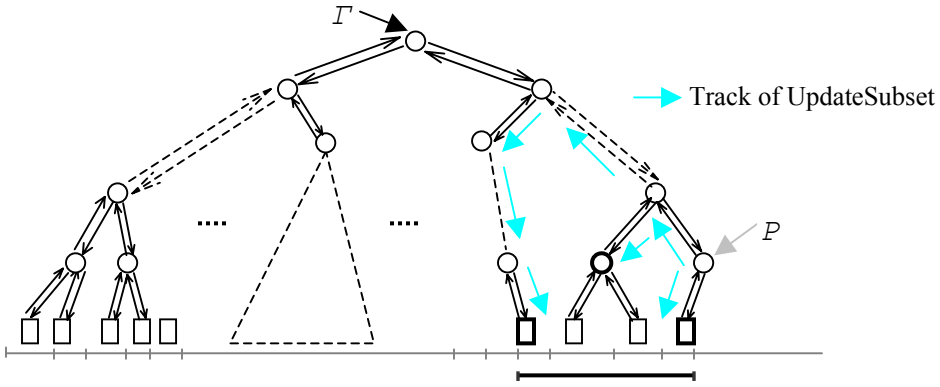
1. If ($b > P.t$) AddNewLeaf(b, Γ, P);
2. UpdateSubset([a, b], V_{id}, Γ, P);
3. Return.

In these two algorithms, procedures AddNewLeaf and UpdateSubset are the key. Their performance defines the update performance of the time interval tree. Variable P is introduced solely to reduce the time and I/O cost while locating nodes to perform data updates. It always points to the newest position in the interval tree in the order of time sequence.

Figure 3-11(a) shows the course of the tree's growth when leaf nodes are added one after another. Notice that a new leaf node is always added at the right most position, since a newly coming in time point is always later than those already existing in the time interval tree. Both Γ and P are dynamic as the time interval tree grows. Figure 3-11(b) shows the process of UpdateSubsets when a new interval is added. The V_{id} is added to the subsets of the bold nodes.



(a) Adding a New Leaf Node



(b) Updating the Subsets for a New Interval

Figure 3-11 Insertion of an Interval to the Time Interval Tree

Here are the algorithms AddNewLeaf and UpdateSubset:

Algorithm AddNewLeaf (t, Γ, P)

Input. A new time point, a time interval tree Γ and Γ's most recent record P.

Output. Updated time interval tree Γ and its most recent record P.

1. If (Γ = Nil) or ($P.r$ = Nil)
2. Add a new leaf node v_0=Node([$P.t$, t], {});
3. Add a new node v_1=Node([$P.t$, +∞], {});
4. Link v_0 to v_1 as the left child;
5. Point both Γ and P to v_1; $P.t \leftarrow t$; $P.l \leftarrow 1$.
6. Else If P is at any level higher than 1
7. Add a new node v_0=Node([$P.r.left.b$,+∞],{}) and link it as P's right child;
8. While the level of v_0 is larger than 1
9. Create left child node v_1 to v_0; $v_0 = v_1$;
10. $P.r \leftarrow v_0$; $P.t \leftarrow t$; $P.l \leftarrow 1$;
11. Add new leaf node v_1=Node([$P.t$, t],{}) and link it as P's left child.
12. Else
13. Add new node v_0=Node([$P.t$, t],{}) and link it as P's right child;
14. While v_0 is the right child of its parent node
15. $v_0 \leftarrow v_0.parent$; Replace v_0's ending time point +∞ with t;
16. If Γ is at v_0
17. Add a new node v_1 = Node([$v_0.a$, +∞],{}); $v_1.left = v_0$;

18. Point Γ to v_1;

19. Point P to $v_0.parent$.

20. Return.

Algorithm UpdateSubset([a, b], V_{id}, Γ, P)

Input. A time interval [a,b], V_{id} of the vehicle that generated [a, b], a time interval tree Γ and its most recent record P.

Output. Updated time interval tree Γ and its most recent record P.

1. $v = P.r.left$; //down track
2. While ([$v.a$, $v.b$] $\cap$ [a, b] $\neq \Phi$) and (**!**([$v.a$, $v.b$] $\subseteq$ [a, b]))
3. If ($v.left.b > a$)
4. $v.right.S \leftarrow v.right.S \cup \{V_{id}\}$;
5. $v \leftarrow v.left$;
6. Else
7. $v \leftarrow v.right$;
8. If ([$v.a$, $v.b$] $\subseteq$ [a, b])
9. $v.S \leftarrow v.S \cup \{V_{id}\}$;
10. If ($P.r \neq \Gamma$) //going up from P
11. $v \leftarrow P$;
12. *While* ($a<v.a$)
13. $v \leftarrow v.parent$;
14. $v \leftarrow v.left$;

15. While (**!**([$v.a$, $v.b$] $\subseteq$ [a, b])) //going down again at the left side

16. If ($v.left.b > a$)

17. $v.right.S \leftarrow v.right.S \cup \{V_{id}\}$;

18. $v \leftarrow v.left$;

19. Else

20. $v \leftarrow v.right$;

21. $v.S \leftarrow v.S \cup \{V_{id}\}$;

22. Return.

The total space usage for a time interval tree is the space each node takes plus the pointers linking the nodes. Inside each node, there is a time interval defined by two time points and a canonical subset. There are three pointers for each node. So excluding the canonical subsets, the space taken for a time interval tree of n time intervals is at most $2 \cdot 2n \cdot 5$ data units. We have analyzed earlier that the total size of the canonical subsets is not larger than $2 \cdot 2n \cdot \log n$. The total size of the tree is hence no larger than $(2 \cdot 2n \cdot 5 + 2 \cdot 2n \cdot \log n) = 20n + 4n \log n$, which yields $O(n \cdot \log n)$.

The worst case I/O cost for a single data update, as shown both in Figure3-11 and in the algorithms, is the standard answer for any binary trees which is $O(\log n)$. However, the average cost is better than in the general binary trees because of the small constant attached to $\log n$. In the procedure of AddLeafNode, each node is accessed at most two times: while its creation and the closing of its time span (end time point b changed from $+\infty$ to a meaningful value). So the total I/O spent on AddLeafNode for the whole tree is $2 \cdot 2n \cdot 2 = 6n$ and the average cost for inserting each time interval is constant. The I/O cost

for each single procedure UpdateSubset depends on the length of the involved interval. Generally speaking, the longer the interval spans, the more I/O accesses is needed for the insertion. Considering the real world situation in our objected application, the actual I/O complexity is far less than O(log$n$). This applies to both the average and the extreme situations:

(1) The time a vehicle running on a road segment is very short comparing to the time span of the tree (referring to the analysis in Chapter 5). For those non-major roads, most of the time there are only as few as no more than 10 vehicles running on the road segment at the same time. This implies that the update of the subsets for these corresponding intervals takes only $2 \bullet \log_2 10 < 8$ I/Os.

(2) The most time intervals appearing in unit time are the cases of rush hours on a super highway. On a highway of length 2 miles with 4 lanes and speed limit of 50 miles per hour, no more than 500 vehicles can be on the same segment at same time. And the I/O accesses are no more than $2 \bullet \log_2 500 < 18$.

So the actual worst case cost is about 18. It is conservative to say that the **average** I/O cost for UpdateSubset in our application model is O(1).

Theorem 3.2 A balanced Time Interval Tree for a set of n time intervals coming in order can be built in O($n$log$n$) I/O using O($n \cdot$log$n$) storage. It takes O(log$n$+k) I/Os to report all the k time intervals that contain a given time point x, and at the same order of I/O cost a range query of reporting all the k time intervals that intersect a given time span [a, b] can be answered. An insertion operation costs O(log$n$) I/O accesses.

Proof The proof lies in the earlier analysis in this subsection.

Corollary 3.2 In a balanced Time Interval Tree on time intervals generated by vehicles running on the same road segment, the insertion of a new time interval to the tree costs an average of O(1) I/O operations.

The way a time interval tree is built implies that a single data update from a vehicle won't immediately cause the update of the indexing structure. Instead, the indexing tree is updated upon a vehicle's entering and leaving of a road segment. Due to this fact, term "*Indexing Event Trigger*" (IET) is introduced for the simplicity for the system design and implementation of the applied applications, and it is defined as follow:

Definition 3.1 An ***Indexing Event Trigger*** is a data update which causes the need to update the corresponding indexing structure.

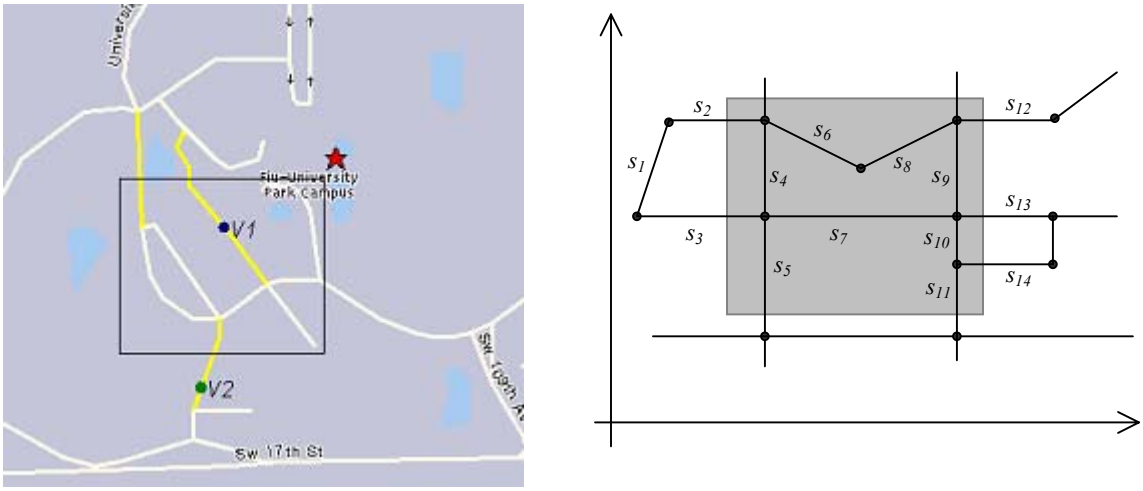
As we will discuss in chapter 6, an IET is not necessarily the leaving of a vehicle from a road segment. It can be of other types, such as when a vehicle enters (or leaves) a rectangular area or areas/locations of other forms. An IET can be associated to the system, a vehicle or a location in implementation.

3.4 Query Processing

In the cases of rectangular range queries, a vehicle included in an answer set returned from the time interval trees are not guaranteed to be one of the answers to the range query. As shown in Figure 3-12(a), vehicles v_1 and v_2 both are returned from the time interval trees when searching answers for range query defined by $[t_1, t_2]\times[x_1, x_2]\times[y_1, y_2]$.

However, v_1 belongs to the final answer set to the query but v_2 doesn't. During time [t_1, t_2], even though v2 appeared on a road segment that intersects the query space range, it was at the part of the road outside of the space range. The term "*Candidate Answer Set*" (CAS) is introduced for the handling of such situations.

Definition 3.2 A "***Candidate Answer Set***" is a set of data that contains all the answers to a given range query.



(a) A real world scenario (b) An abstracted picture

Figure 3-12 The Candidate Answer Set

The candidate answers in our system appear in the intermediate results of the rectangular range queries. Figure 3-12(b) gives more details about candidate answer set. On a range query on the vehicles appearing in space range R as shown in the gray area in the figure during time span [t_1, t_2], the road segments in set S = {s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9,

s_{10}, s_{11}, s_{12}, s_{13}, s_{14}} all intersect rectangular area R. However, segments in S_1={s_4, s_6, s_7, s_8, s_9, s_{10}} are fully inside R but segments in S_2={s_2, s_3, s_5, s_{11}, s_{12}, s_{13}, s_{14}} are partially overlapped by R. A vehicle appearing on the road segments in S_1 at any time during [t_1, t_2] should be included in the answer set. But those vehicles that are on the road segments in set S_2 may or may not be inside R during [t_1, t_2], since they could be on the part of these road segments which are outside R during the time range. When such a query is processed, S_1 is the final answer set and S_2 is a candidate answer set. To decide if the items in S_2 are included in the final answer, their moving trajectories during time [t_1, t_2] need to be verified against R. Those that do exist in R during [t_1, t_2] will be included in the final answer, while the rest excluded.

To answer queries involving future time, prediction on vehicles' possible moving trajectory may be necessary. An important technique required for such prediction is the computation of the near future paths of the vehicles. This section will talk about the Path Computation algorithm before showing and evaluating the query answering procedures.

3.4.1 Path Computation

The path computation on road maps is an active topic in several research communities. In the context of this dissertation, we need a path computation algorithm as a part for answering queries of form "which vehicles can reach a given point within time T". Among all the possible schemes discussed in literature such as [ARR00, SFL96, SMWH], the A* search mechanism, is recommended by Shekhar *et al* based on their analysis and experiments described in [SF96].

A* search is a heuristic search scheme taking the advantages of both greedy search and uniform-cost search. It evaluates a choice n at the current point by combining $g(n)$, the cost from the starting point up to the current point, and the estimated cost from the current point to the goal $h(n)$:

$$f(n) = g(n) + h(n)$$

Russell and Norvig showed in [RN95] that A* search with an h function that never overestimates the cost to reach the goal is complete and optimal. So we can define $g(n)$ and $h(n)$ as follow:

$$g(n) = \sum_i \frac{L_i}{S_i}$$

$$h(n) = \frac{Dist(p_n, p_e)}{V_{\max}}$$

where L_i is the length of the ith road segment in the chosen path, S_i is the speed limit of this segment. It is possible to take the advantage of the real time traffic data in the system and assign S_i the current actual driving speed on each road segment. If it is a road that will be reached later on, the history speed information on the same road segment at the same time of a day can be used. Hence $g(n)$ is the time taken to drive from the origin to the current point. The h function shown is the shortest possible time from current point to the end point, since $Dist(p_n, p_e)$ is the Euclidian distance from the current location p_n to the destination p_e and $V_{\max}$ the maximum speed limit within the range of the map.

Considering the huge main memory consumption in doing A* search, the search length is limited to M steps. M is a constant in an application. The value of M can be

decided by factors including the hardware and software configuration of the system where the application runs on.

Algorithm PathComputation(p, q)

Input. An origin point p and a destination point q each defined by a tuple (R, d), where R is a road segment and d the distance from the starting point of R to this point, and the road map (*global*) given in the format of a set of directed road segments as defined earlier.

Output. A route (from p to q) S, composed of a sequence of consecutive road segments.

1. Locate points p, q; $r \leftarrow p.R$;
2. $S \leftarrow \{r\}$; $p_n \leftarrow r.end$; $T \leftarrow \{\}$; $m \leftarrow 0$;
3. While (Goal NOT Reached)
4. For (all r's succeeding road segments r'_i)
5. If (r'_i in T or S) then $f(r'_i) \leftarrow \infty$;
6. Else $f(r'_i) \leftarrow r'_i.length / r'_i.SpeedLimit + dist(r'_i.end,q) / Vmax$;
7. $r \leftarrow$ the r'_i with the smallest $f(r'_i)$;
8. If (the smallest $f(r'_i)$ is ∞)
9. One step back and choose the next smallest answer;
10. Else
11. $T \leftarrow T \cup \{r\}$; m++;
12. If ($m \geq$M)
13. Move the first item of T to the new last item of S; m--;
14. Append T to S;

15. Remove loops, if there are any, from *S*;

16. Return *S*.

3.4.2 Answering the Queries

Range queries are answered by taking steps. Different types of queries may incur different kinds of operations. Figure 3-13 shows the data flow in the system while processing range queries. The possible query answering steps include:

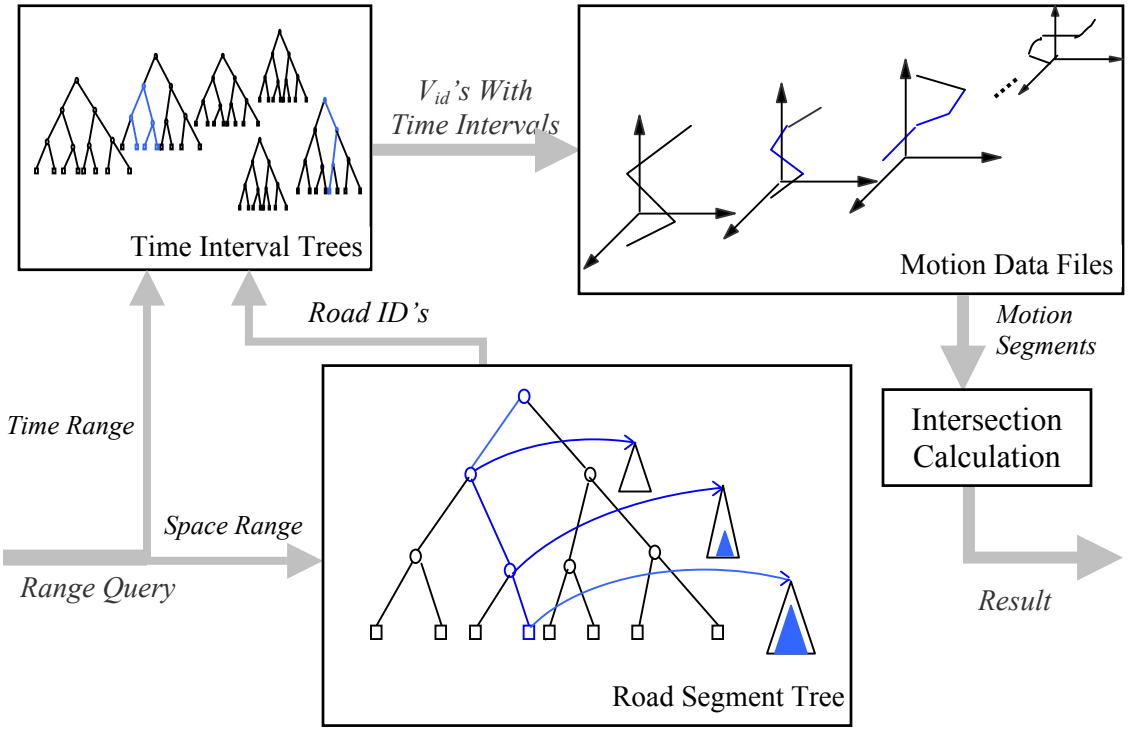


Figure 3-13 Data Flow in Query Processing

(1) Do path computation and predict vehicles future motion.

(2) Locate the road segments intersecting the given space range.

(3) Retrieve the vehicles that are on the road segments found in step 2 during the given time range.

(4) Finalize the candidate answers.

Depending on the query type, one or more of these steps will be involved. For each of the typical types of queries, their processing procedure and performance are analyzed as follow:

(1) For types of Q1 and Q_2 queries, searching for an answer is trivial.

Q_1. Display the motion of all or a given sub-set of vehicles in real-time.

Q_2. Replay the motion of a given set of vehicles from past time t_1 to time t_2, t_2 can be past, now or near future.

The current and most recent motion of all the vehicles will be kept and buffered in main memory. For the history data, each vehicle's motion is kept in continuous storage and well indexed, which makes the retrieval extremely simple. Locating the data mostly takes only 1 I/O, loading the meta data. Fetching each vehicles past motion in time frame $[t_1, t_2]$ takes *k/B I/O's*, where k is the size of the motion data and B is the block size. And the future motion prediction is done in main memory.

(2) For queries of type Q_3,

Q_3. Return the vehicles that have been on a given road segment during $[t_1, t_2]$.

The time interval tree associated to the given road segment is searched. After getting the V_{id}'s from the time interval tree, the motion data on each vehicle is retrieved. I/O complexity: $O(\log T+k) + O(k) = O(\log T+k)$, where T is the total number of time intervals

associated to the same given road segment, k is the number of vehicles appearing on this road during $[t_1, t_2]$.

(3) For queries of type Q_4, it takes three steps.

Q_4. Display the vehicles that have been in a given rectangular area R, during $[t_1, t_2]$.

First we search all the road segments $\{r_i\}$ intersecting R from the road segment tree. Then go to the interval trees associated to these road segments, searching for all the V_{id}'s of the vehicles which have appeared on any of these road segments during $[t_1, t_2]$. The vehicles appear on road segments inside R are certainly part of the final result. The trajectories of those vehicles appear on segments partially ovelapping R need to be examined and actual intersections need to be calculated.

The first step takes $O(\log N+m)$ I/Os where N is the total number of road segments and m is the number of segments reported, the second step takes $O(\log T +k_i)$ for each road segment where T is the total number of time intervals associated to the road segment and k is the total number of intervals reported. The third step is done in main memory and hence no I/O cost involved. The total *I/O* cost: $O(\log N + m) + O(m \bullet \log T + \sum_{i=1}^{m} k_i) =$ $O(\log N + m\log T + k)$, where k is the total number of vehicles that appear on the m segments during $[t_1, t_2]$.

Notice that when the query area is large, the number of road segments gets very large too, and the operation becomes costly. We know that any indexing structure has to trade off between space expense and I/O cost. In our targeted application model, the response time requirements for data updates and data queries are different. The balance is hence

among the costs on space, data update and data query. Since the data updates are the most critical type of operations, it can't be guaranteed that every type of queries be answered in optimal I/O operations. Also as we will discuss in chapter 6, if we know ahead of the frequently queried areas, Indexing Event Triggers and time interval trees can be set and built at data update stage. This way, the I/O complexity for these queries reduces to the same as Q_3 queries.

(4) Queries of type Q_5 and Q_6 require path computation.

Q_5. Display the vehicles that can reach a given point $p=(p_x, p_y)$, within time Δt.

Q_6. Predict the motion of the vehicles in the near future from t_1 to t_2.

For queries of type Q_5, we first do a range query of type Q_4 with range R defined as:

$$R = [\ (p_x - V_{max} \bullet \Delta t),\ (p_y - V_{max} \bullet \Delta t),\ (p_x + V_{max} \bullet \Delta t),\ (p_y + V_{max} \bullet \Delta t)\ ]$$

and time frame [*now*, *now*]. Then we do path computation, finding path and time cost for each vehicle whose V_{id} is included in the first query. The set of vehicles with time cost no larger than Δt is returned.

Queries of type Q_6 apply only to those vehicles with known destinations. We first find path for each of them and then calculate their motion in $[t_1, t_2]$ along their paths.

3.5 Conclusion

This chapter discussed the data model, data structure and indexing structure as well as the processing of queries for a real time tracking (and history retrieval) system for the location information of vehicles moving on road networks. The I/O complexities of the critical data access types are not only low in the order, but also with small constant

attached to the order. Updates on new coming data can be done in O(1) I/O operations. Based on the analysis in sections 3.3 and 3.4, we conclude that the data structure and indexing mechanism we adopt are efficient, and the real time requirements for data updates and retrieval can be well met.

4 SYSTEM ARCHITECTURE

A vehicles' location information tracking system mainly involves four kinds of participants: registered vehicles, data processing center(s), the end users and/or client services, and communication channels.

Figure 4-1 shows a complete picture of a tracking system for GPS enabled moving vehicles. Each vehicle included in the system is equipped with a GSP receiver to receive its up-to-date location information from the GPS satellites periodically. And there is a wireless channel, through ground communication towers or communication satellites, for the vehicles to update the data center on their current location and other time-variant information such as their destinations if applicable. The data center is the core of the system, which keeps the system running and functional. It is responsible for the organizing, storing, and accessing of the input and output data. It keeps the vehicles' current and history location information and provides retrieval services in a timely manner to different kinds of users. It not only needs to provide services with high performance, the data center is required to be highly reliable and persistently available. The "users" can include the drivers of the vehicles, the owners of a single vehicle or a group of vehicles and others who are authorized to access such data. A data item must be and is only visible and accessible to the group(s) of users who have been granted access permission. The communication channels include wired (for the local and wide area computer networks) and wireless (communication satellites and/or ground radio channels).

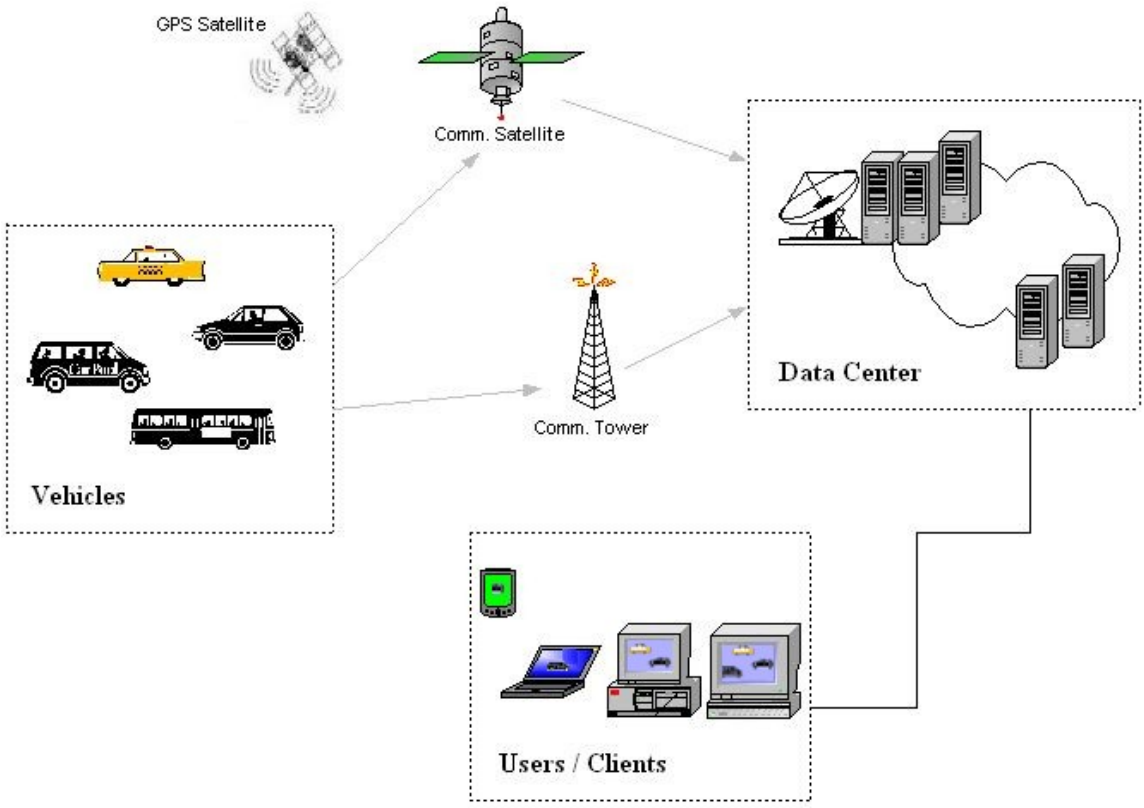


Figure 4-1 A Vehicles Location Information Tracking System

The system works as follows. The vehicles are real world vehicles that mainly move on road networks. A vehicle receives GPS signals from the GPS satellites to determine its geographic location, which can be converted uniformly to the Universal Transverse Mercator (UTM) coordinates. It then periodically updates the data center of the up-to-date location information via wireless communication channels. The data center stores vehicles' static attributes and the real-time location updates from the moving vehicles. It keeps the history data as well as the current information associated with each vehicle. The function of the data center is to provide the online client applications and/or end users access to these data typically in the form of query answering. The end users of

client applications can then make decisions or plans based on the query results, if they so desire.

The main concern of this thesis, especially in this chapter, is the structure and processes inside the data processing center. This chapter discusses the architecture of the data processing system, namely a real-time vehicle location information tracking system, inside the data center.

The rest of this chapter is organized as follows. Following a brief background introduction, Section 4.1 presents the overall architecture of the system. Sections 4.2 through 4.7 address how to handle the key issues in the system, including the system organization, the real-time requirements, storage management, backup and recovery, transaction and concurrency control, and security, respectively. Section 4.8 briefly mentions the User Interfaces. Section 4.7 concludes this chapter.

4.1 Overview

In the context of this dissertation, we don't intend to build a general database management system for Moving Objects. Even though our data structure and system architecture can be embedded into a DBMS as a data blade (this will be discussed in Chapter 6) and the system itself can be upgraded to a DBMS by adding other management components, at this stage it is solely a database application system independent of any existing DBMS's. The system will be built on top of the data structure introduced in Chapter 3. The system is designed to meet, but is not limited to, the following requirements:

- Keep track of the current location and motion of all the vehicles in real-time.
- Store and manipulate the history data efficiently.
- Retrieve both real-time and history data in a timely manner.
- Be constantly available and highly reliable.
- Disclose data items only to the authorized parties.

To maximize the high throughput, scalability and real-time properties that the application system desires and our data structure and indexing mechanism supports, a centralized distributed system architecture is adopted. On the one hand, parallel processing is desirable in such systems due to the massiveness of the data and the real-time requirements, and our data structure is designed to support highly parallel system architecture. On the other hand, the cost for communication, synchronization, and control in a pure distributed system is expensive, which would greatly compromise the performance brought by the parallelism in a distributed system. But a distributed system with centralized organization takes all the advantages a distributed architecture provides and avoids the unnecessary costs in communication and synchronization.

4.1.1 Distributed Architecture

In general, a distributed system provides higher performance and more reliable services by distributing the system workload to multiple nodes which run concurrently. In our case, several processes are basically independent of one another. So a distributed architecture is especially suitable.

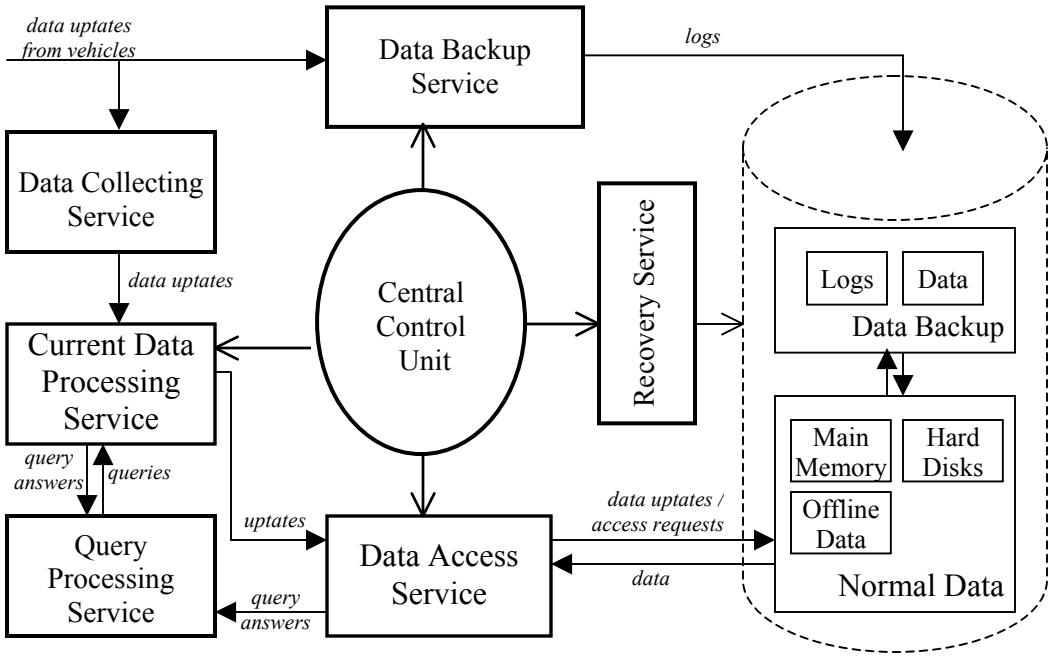


Figure 4-2 Distributed System With Centralized Organization

To meet the minimum functional requirements, modules to handle data collecting, query processing, and data accessing should be included in our system. Considering the system reliability requirement, a backup and recovery mechanism must be provided as well. To coordinate and synchronize these modules, a mechanism is needed to manage and support the communication among these modules. As shown in Figure 4-2, the main functional components in our system include:

- The Data Collecting Service is responsible for collecting the real-time raw data on vehicles. It receives real-time data from the vehicles through wireless channels. Upon receiving data, it first forwards them directly to the Data Backup Service so that the data update can be logged for recovery purpose. It then formats the data

and sends them to Data Preprocessing Service for database updates and retrieval purposes.

- The Data Processing Service plays the key role in enforcing our data model and data structure defined in last chapter. It locates the vehicles' current positions on the road map and hence associates each vehicle with the road segment it is running on. It checks data validity, and corrects errors if possible, dumps the wrong data which can't be corrected. Data on consecutive sampling points are compared here and the insignificant points are removed then. The vehicle's motion data on the most current road segment are always kept in the main memory of this service node. When it is time to update the persistent storage on any vehicle's current trajectory, it forwards the related data to Data Access Service. Since the most current data are stored in main memory on this node, queries on the up-to-date information are also processed here.
- The Query Processing Service is the bridge between the client applications or end users and our database system. It interprets queries coming in any formats and forwards them to Data Pre-processing Service or Data Access Service depending on the time range in these queries.
- The Data Access Service is responsible for updating the database and answering queries on recent or past data. When it receives an update from the Data Pre-processing Service, it puts the physical data to data file, and updates the indexing structure to allow the current update to be indexed.
- The Backup Service handles all the data that may be needed upon recovery. Keeping data logs is one and the most reliable way to be prepared for system

failure in our case. At least one copy of the recent data, along with the indexing structure, should be kept as well. The reason for this is that recovering from the data logs is much more costly than recovering from readily organized data. Even though logically the Backup Service "receives" raw data from the Data Collecting Service and "puts" them to data logs, one of the physical media where such logs reside in could very likely be the hard disks of the Data Collecting Service node. The advantage of such an arrangement is obvious.

- The Recovery Service restores the disrupted data and recovers the system in case of any forms of failures. Since the system is required to be functioning at any time, in case of a "core dump", no time can be given to the recovery process before bringing the main system functions back. So the recovery processes are always supposed to be going in parallel to the system's main services. The Recovery Service is also responsible for bringing offline data, as we will discuss in section 4.4, back into the system online upon requests.
- Storage in our system exists in several forms. Strictly speaking it doesn't provide any system functions independently. Details on storage management will be discussed in section 4.4.
- The Central Control Unit controls and coordinates the other service nodes to achieve the performance, reliability and security of our system. Next section, Section 4.1.2, will address the details.

The term "service" frequently appears in this section. It is conceptual, flexible and scalable in the context of our system architecture. There is not necessarily a one on one relationship between a service and a physical server machine which provides the service.

At implementation stage, one service can be carried out by several physical server machines and multi-services can be integrated on one machine, depending on the size of the system.

4.1.2 Centralized System Organization

By dividing the system functionalities into services and distributing these services to multiple server nodes which can be running in parallel, the overall system performance can be greatly improved. However, in traditional distributed systems, the communication, synchronization and coordination among these independent server nodes are expensive and sometimes difficult. Due to the real-time requirement of our system, such costs are strongly undesirable. To avoid these expenses and improve system response time, a centralized system organization would be a wise choice.

Data and messages may pass from one to another among the service nodes, but when synchronization or arbitration is needed, requests are sent to the Central Control Unit. The Central Control Unit, upon receiving a request, makes the decision and sends back control back to involved parties. In our system, not all the service nodes must be under the control of the Central Control server. Some services, such as the Data Collecting service and the Query Processing service, are relatively independent of the core of the system. They don't need to request any authorization or arbitration from the Central Control server, and hence are not directly under the control of the later. As shown in Figure 4-2, the nodes that provide data pre-processing services, data access services, data backup service, and data recovery services exchange data among each other, they are also under the direct control of the Central Control Unit whenever a control is needed.

The system works as follow. The only source of the data input to the system is the Data Collecting Server. It sends the data to Data Preprocessing server. Data are processed there, kept in the main memory of this node, and later on forwarded to Data Access server. The Data Access server records the data updates into the database in persistent storage upon getting permission to do so from the Central Control Unit. When a query comes from the user/client end, the Query Processing service first interprets the request. It then forwards the interpreted request to Data Pre-processing server or the Data Access server, depending on the time range defined in the query. Security checks are enforced at the time of processing the answers to each query.

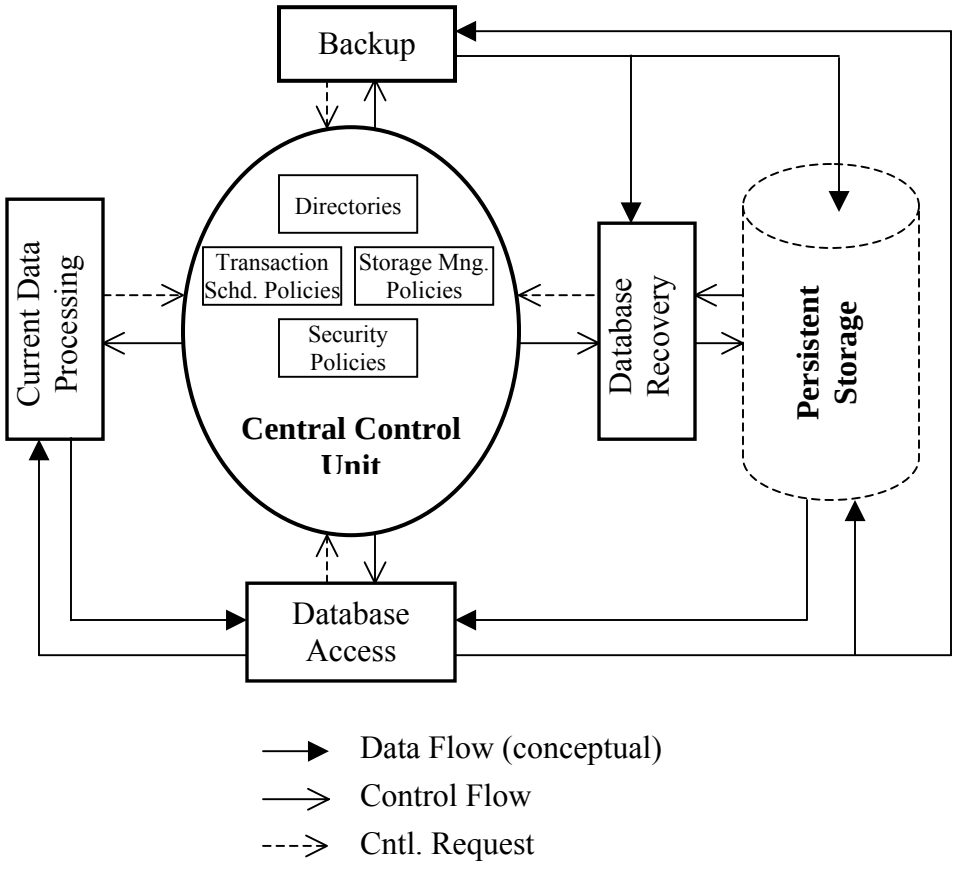


Figure 4-3 The Central Control Unit

In the background, the backup processes run with permission from the Central Control Unit, assuring that the foreground processes are not affected. In times of database recovery, potential conflicts with the real-time database updates occur frequently. Transactions are scheduled by the central control server in such cases.

As shown in Figure 4-3, the Central Control Unit mainly contains the following data in its main memory:

- A directory on the locations of the data and services.
- Transaction scheduling policies
- Storage management policies
- Security policies.

It takes charge of concurrency control and transaction scheduling, storage management, and security enforcement. Within storage management, both the data backup and recovery are included. The central control server always works closely with the data structure to ensure the functionality, availability and reliability of the system. Any operations involving data items in the database must be under the management and control of it.

A centralized organization eliminates the number of messages and amount of data passing through the functional nodes. It also makes it simpler, easier, and more reliable to enforce or carry out security and other policies.

4.2 Meeting the Real-Time Requirements

Real-time systems can be divided into three categories according to their requirements on meeting the deadlines:

In hard real-time systems, the missing of the deadlines reduces the value of the data to negative. Data updates are of value only if they are committed before or on the deadlines. Not only becoming meaningless, the missing of their deadlines cause negative effects to the system.

A soft real-time system prefers that all the data updates are committed meeting their deadlines. However, data are still of value when their deadlines are missed. The value of the data reduces eventually to 0, after the deadline is missed.

Firm real-time systems are those with a deadline requirement in between.

In a real-time tracking system, users have the needs to access the most current data. In cases such as bus / taxis scheduling, obsolete data may result in bad or even wrong schedules. Individuals who plan their trips based on information provided by the tracking system could miss their appointments or even opportunities if the system can't keep the data up-to-date. It is highly desirable for the data updates to meet their deadlines. However, the data updates won't become useless even if they are committed after their deadlines. Such updates are still of value when we do history retrieval and analysis. So our system falls in the firm real-time category. In case they are missed, the system should try to commit the transaction as early as possible.

According to their degree of urgency, transactions in our system can be classified to three types:

- Urgent – The most current data on vehicles' location. These group of data need to be put into database as soon as they come in.

- Medium Urgent – Transactions acquiring the current information on vehicles' movement. These transactions should be committed as early as possible if they don't delay the commits of the Urgent transactions.
- Neutral – Queries on the history data. There is not a clear deadline for such queries. Slowness in response to such queries is only evidence of bad system performance.

In our system, the real-time requirements are met by means of the following system architecture and management policies:

(1) The distributed architecture enables several processes to run at the same time. System throughput and performance are then highly ensured. The parallelism in the system also makes it more likely for the conflicts among transactions to be resolved without missing any of their deadlines.

(2) Any data update that can be approximated by its preceding or succeeding updates is not stored into the persistent storage. This reduces overall load of the system, hence makes it easier to meet the deadlines for urgent transactions.

(3) The most current data are kept in main memory due to the high access demands on them.

(4) The transactions are scheduled to commit according to their levels of urgency, when there are conflicts.

One of the most important factors that define the real-time performance of real-time database systems is their storage management mechanisms. Another important factor that

affects the real-time property of a database system is its concurrency control policies. We have taken into full consideration of meeting the real-time requirements when defining the storage management, transaction management and concurrency control policies of our system. The following two sections will discuss how these policies are defined to ensure the real-time property as well as meet the other requirements of the system.

4.3 Storage Management

Four levels of storage are used in our system. Obviously, the latest data on vehicles' motion are in main memory before newer update data are received. Some frequently accessed data structure may need to permanently reside in main memory as well. The recent data on vehicles' motion are accessed relatively frequently, they should be conveniently reachable. These data are stored in local hard disks. Older data can be stored in file systems in local network since they are accessed infrequently. These three levels of storage are all online. There is yet another kind of storage, which is offline though. The out-of-date data are not likely to be accessed in a daily basis, but there are needs in some cases to retrieve information from these data. Considering the huge volume of data there could be for each day, Compact Disks do not have enough capacity for this purpose. High capacity tapes are used to store the far past data.

Figure 4-3 shows the distribution of data of different "age" in different levels of storage and transfer between them.

The trajectory of each vehicle's movement on a road segment is not written to disk until it leaves the current road segment and enters the next segment. So the motion on the current road segment is always kept in main memory for all the vehicles. The data

describing the road segments are frequently accessed for purposes such as calculating vehicle's motion between sampling points, path finding, and vehicle's motion prediction. The current version of this set of data also needs to be in main memory.

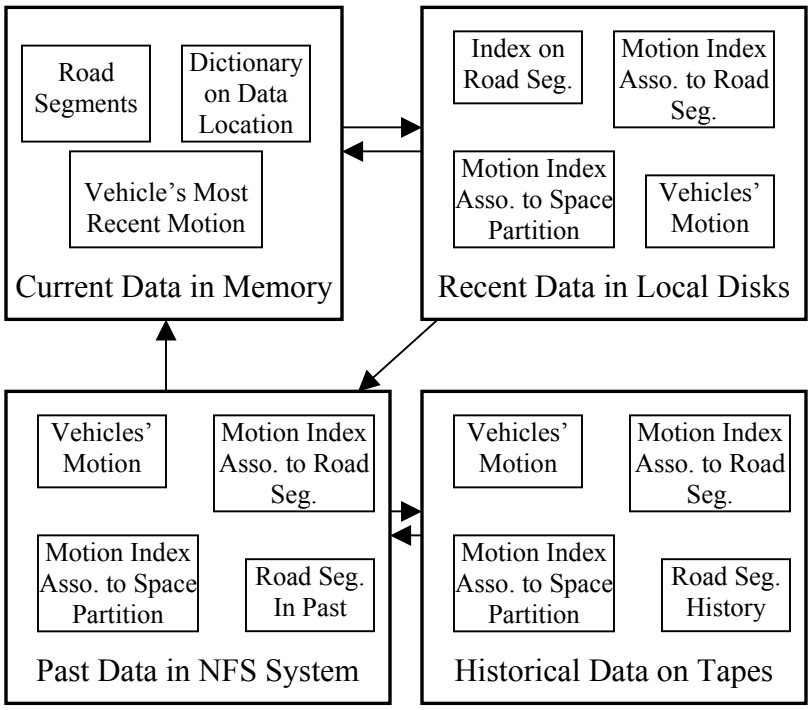


Figure 4-4 The Four Levels of Storage

Only next to the current data, data on vehicles' recent movements are the second most frequently accessed. This set of data, including the indexing trees and the data files on each vehicle, is stored in local disk. The indexing structure on road segments is used for some range queries. The frequency of accessing it is close to that of the retrieval on vehicle's recent motion. It is also kept in local disks.

The frequency of accessing the past data decreases as reversing time back. It is reasonable to put the past and history data in storage media that incurs longer access time.

The past data on both vehicles' motion and the road segments resides in Network File Systems. They are still highly available, only slower in accessing time. The historical data are not necessarily to be online. Putting them in tapes and loading to hard disks upon request is the economical way to make them reachable.

Our indexing mechanism enables and fully supports storing data separately on different media, according to the time the data is on. A dictionary (easily accessible to the control server) residing in the main memory is built up for locating storage location where a given set of data is stored.

In managing the main memory, the classical techniques, such as buffering and anticipatory prefetching, can also apply to our system. However, these techniques work closely with Operating Systems the application runs on. They can be selectively adopted in the implementation stage, taking consideration of our system performance requirements and the features the Operating System and hardware support.

The management of the secondary storage, namely the hard disks in our system, is supported by applying the following mechanism:

- The indexing scheme minimizes I/O costs in locating physical data.
- The organization of the motion data for each vehicle ensures that there is at most two I/O accesses overhead in loading consecutive data of any size on one vehicle.
- Data and access are evenly distributed to multiple servers and/or disks depending on the size of the application system.
- Queries processing is optimized.

General data buffering and disk scheduling techniques that the operating system and hardware support should be considered in the implementation phase.

4.4 Transaction and Concurrency Control

For database systems, correctness in the presence of concurrent access and/or failure is tied to the notion of a transaction. A transaction is a unit of work, possibly consisting of multiple data accesses and updates, that must commit or abort as a single unit. Transaction executions are supposed to respect the ACID properties:

- Atomicity – either all or no operations are completed.
- Consistency – all transactions must leave the database in consistent state.
- Isolation – transactions cannot interfere with each other.
- Durability – successful transactions must persist through crashes.

The goal of transaction management is to maximize the system throughput and minimize the amount of restart overhead. The mechanisms used to resolve conflicts and schedule the execution of the transactions is called Concurrency control.

The main issue in transaction management in a real-time system is how to meet the urgency of transaction executions and, at the same time, to maintain the database consistency.

The same as in general database cases, real-time concurrency control algorithms can be generally classified as either conservative or optimistic. Conservative algorithms prevent any violation of database consistency from happening. Some variations of conservative algorithms adopt multi-data-version strategies [LK01] to increase the concurrency level of the system. Typical conservative mechanism is lock-oriented real-

time concurrency control. Optimistic concurrency control algorithms often delay the resolution of data conflicts until a transaction has finished all of its operations. A validation test is then performed to ensure that the resulting schedule is serializable. In case of data conflicts, they usually use transaction aborts to maintain database consistency. The validation schemes, such as wait-50 and sacrifice, make the conflict resolution being priority-cognitive. Most of the existing optimistic concurrency control algorithms are for soft and firm real-time systems, because the abortion cost of a real-time transaction is often hard to quantify. For most of the real-time systems, it is very difficult to ensure the schedulability of the transactions under the optimistic approach.

A delayed write procedure is often adopted in many concurrency control algorithms: the update for each data object by a transaction is done in the local area of the transaction, and the actual write of the data object is delayed until the commit time of the transaction.

When a lock-oriented concurrency control algorithm is adopted, each object in the database is associated with a lock. When a transaction locks a data object, it must lock it in a proper mode, such as share or exclusive.

One of the trends in real-time database concurrency control is semantics-based concurrency control. It ensures correctness by enforcing final-state serializability, view serializability and conflict serializability. Data similarity in terms of their semantics is introduced and used to manage the transactions. The integrity management of real-time database systems involves two issues: 1) External consistency and temporal consistency constraints must be specified and justified to be adequate for the similarity relations specific to the application. 2) Real-time transaction scheduling algorithms are needed to enforce these constraints. Kuo and Mok, in [KM00], proposed the idea of physical

schedules in which a real-time database scheduler may skip unimportant computation or updates to meet time constraints and/or satisfy some safety requirements on the system. The correctness of the physical schedules is justified by the notion of similarity.

Even though transaction scheduling and concurrency control techniques have been extensively studied, there doesn't exist a total solution for the specific needs of our system. In our system, the data access operations include only Inserts and Searches. Operations of type Modify and Delete are not supposed to occur. In addition, the Inserts and Searches never appear in the same transaction because these two types of operations are always requested by different processes. Those read/write violations, which is one of the main problems in general database systems, are not likely to happen in our case. And serializability is always satisfiable. Our goal for transaction management is to maintain the real-time property, ensure data correctness, and maximize system throughput.

First, the similarity based concurrency control policy is adopted and "over" enforced in our system. As described earlier, the "similar" data updates are omitted, ignoring if there are conflicts and if it is missing the deadline.

Second, data are buffered in memory until a vehicle leaves a road segment and the set of data are written to hard disk in one transaction. This way it eliminates the number of transactions while maintaining the size of a transaction not to be too big, so to reduce the number of possible conflicts.

Third, in the cases that conflicts do occur, the update transactions have the highest priority. The transactions containing Read (search/retrieval) operations are scheduled in the order of the time of the data they request. E.g. a transaction that acquires current data

has higher priority than one which requests recent data, and a transaction that requests recent data has higher priority than one that queries past data.

The system ensures the correctness and performance by enforcing these rules in the transaction management along with other mechanisms discussed in other sections.

4.5 Backup and Recovery

In database systems, unpredictability exists in many aspects due to the dependence of the transaction's execution sequence on data values, the data and resource conflicts, dynamic paging and I/O, and rollbacks and restarts resulting from transaction aborts. Communication delays and site failures happen in distributed systems. These factors all could result in system failures.

The way to handle failures is by means of backup and recovery. When a failure happens, the system should be able to recover. To enable recovery, backup is necessary. Backup and recovery are always risky; in general, the following schemes can be used to minimize the chances of data lost:

- Store everything on a fault-tolerant disk array.
- Use battery backup.
- Use reliable hardware.

Research and industrial experences also show that the following mechanisms improve the chances of successful recovery, in case it does happen:

- Store backups on a third disk on another controller
- Store backups on a different computer on a different network in a different physical location.

- Plan and configure for recovery from the beginning.
- Test the recovery strategy from time to time.
- Make the recovery strategy easy to maintain and test.

Down time in a real-time system, no matter how short it lasts, is fatal and strongly unacceptable. This fact makes it critical and more difficult to recover a real-time database system when it is needed, than in the normal database systems cases. It is even more complicated in a real-time current status tracking and history retrieval system, since in such systems the common event log scheme used in general database systems would not work. General database systems recover themselves upon failures or errors by means of keeping one or more copies of an event log, which records the history of all the transactions that have committed. In our case the database itself is the data defining the most up-to-date status and records of the transaction history. So the traditional way of keeping event logs and restoring the database based on these logs may not be simply applied to our system.

The process of recovery is another challenge for real-time systems. To recover systems without real-time requirements, short system down time may be permissible. But in real-time systems like in our case, any recovery must be done in “hot” mode. That is to say, the recovery process should not interrupt the running of the system.

For Real-Time Database systems, the following strategies and techniques can help in keeping the real-time properties while performing backup and recovery: reduce log traffic; speedy logging and recovery; use transaction priority oriented logging and recovery, data class oriented logging and recovery; partitioned and parallel logging and

recovery; ephemeral logging; use Non-Volatile High Speed Store as a fast persistent backup store.

Considering the characteristics and performance requirements of our system, we need to secure our data in all the possible aspects, by means of multiple mechanisms. As a result, the following policies are used in our system:

- Redundancy. The Central Control server, the Data Pre-Processing server and the Data Access server are the most critical nodes in our system. A hot backup server is allocated for each of them.
- Event logs are done at the data collecting server. Data is logged to disk as soon as they are received by the Data Collecting server. In case of data lost caused by any sorts of failure, we can recover the database by redoing the calculation based on these data logs.
- Fast I/O disks to keep the most recent data.
- A server solely for the backup and recovery purposes respectively.

These policies altogether work as follow.

- The process of backups: Raw data coming in from the vehicles are written to event logs upon their arrival. This happens at the Data Collecting server. After preprocessing, data are written to persistent storage with two copies, one of which is kept as backup. Older and history data are backed up to tape.
- The process of restoring.
    - Restore from event logs. The event logs are in fact the raw data. To restore, the recovery server needs to re-calculate as the pre-processing

server does to the incoming data. And the rest processes are the same as to the data after processed by the pre-processing server.

- Restore from backup. The backups are ready to use as soon as they are put back online, since the physical data files and the indexing structures are readily well organized. The directory in Central Control unit needs to be updated to reflect the new status of the data locations in this case.

4.6 Other Issues

To make our system sound and complete, we need to address a couple of other issues in addition to the topics talked about in last few sections. Security, Query Processing, and User Interface in our system will be discussed in this section.

4.6.1 System Security

A simple but widely-applicable security model is the CIA triad. Standing for Confidentiality, Integrity and Availability, they are three key principles that should be guaranteed in any kind of secure system. Confidentiality is the ability to hide information from those who are not authorized to view it. It is perhaps the most obvious aspect of the CIA triad and the one that is attacked most often. Integrity is the ability to ensure that data is an accurate and unchanged representation of the original secure information. Availability is defined as the assurance that the systems are accessible when needed, by those who are authorized to. It is important to ensure that the data items are readily accessible to the authorized users at all times. If any one of the three aspects can be breached serious consequences may be caused.

A software application system's confidentiality, as well as other security aspects in some extent, is basically and greatly impacted by the security mechanisms the Operating System on which the software is running adopts. It is important to choose a secure operating system and build the application security policies on top of it. First we choose a secure O.S. Two most possible options are OpenBSD and Sun's Trusted Solaris. The open source OpenBSD was built from the ground up to be secure, by constantly auditing the operating system's code for potential security problems. OpenBSD also incorporates encryption in the operating system. Sun Microsystems claims that its Trusted Solaris is one of the most widely deployed trusted operating systems. However neither of these two Operating Systems are suitable for desktops. In such case, Linux would be an appropriate choice.

Second, we adopt role based access control (RBAC) in our system for authentication and authorization. RBAC is chosen because of the following facts:

- The users of our system are grouped in nature, because the vehicles may be used for different purposes and managed by different owners. This matches the way RBAC works.
- It is not complicated to implement. The cost for implementing a role based system is inexpensive.
- It has reduced complexity in administration of such systems.
- It is suitable for network and distributed systems.
- It is now an American National Standard.

The integrity and availability of our system are mainly provided and maintained by the mechanisms we stated in the last few sections, together with a good protection of the system confidentiality. System integrity and availability are ensured by the following mechanisms which have been discussed in detail earlier:

- Good system organization.
- Reliable and high performance hardware.
- Redundancy.
- Our data structure and indexing scheme.
- Backup and "hot" recovery.

Potential attacks and problems exist at other points of the system, such as the data communication between the vehicles and the data center, the communication channels between users and the data center, etc. Cryptography and Encryption may be used; Data Compression may be necessary since the answers to some queries could be huge in size.

4.6.2 Query Processing

At the beginning of this writing the types of queries of our concern are listed. An out-of-fashion yet simple and straightforward way to interpret these queries is to require them to come in the form of a function call. The query criteria are passed to the system as parameters. Even though it is simple and efficient, it is obviously not flexible and not complete.

In the literature of Moving Object Databases and that of Spatio-Temporal Databases, query languages for sptio-temperal objects have been broadly explored. Among those

proposals, the FTL, which standards for Future Temporal Language, proposed by Wolfson [WSXZ99] is the most closely applicable to our system since the data model of our system is similar to which was used in [WSXZ99]. However FTL is not designed for the same case as ours, some work may be needed to extend it to fit our case.

4.6.3 User Interface

The data consumers of our system include the end users and applications that are built on these data. The end users include the commuters who check their bus status, drivers who monitor their current locations or retrieve the past route, etc. Applications can be built for vehicles scheduling, traffic and other statistics, using the data retrieved from our system. The interface for both cases is closely connected to the format of the queries the system supports, but should not be the same considering their different characteristics.

Generally speaking, the end users request smaller amount of data each time and the data they request can be represented in simpler form, while the applications may request large amount of data in complicated form in terms of being represented by standard queries. So the formalized query language support is about enough for the end users. And Application Programming Interface, due to the reduced processing procedures, must be provided for the applications.

4.7 Conclusion

A successful software system is the integration of a well designed system architecture and an efficient data structure. As we have discussed in the last sections of

this chapter, the system architecture we adopt not only well fits the data structure and indexing scheme we proposed, but also provides support to the real-time, flexible and reliable properties that a moving object database system desires.

The distributed property enables highly parallel processing. System performance is hence tremendously improved. The distribution of system functions to multiple server nodes also makes the system scalable and flexible. This type of architecture is especially suitable for our indexing scheme since our indexing structure is a set of indexing trees among which there is little interference between one another.

The advantage that a centralized system organization brings to a distributed system is the reduced message passing or communication expenses, the simplified scheduling and control procedures, and improved system performance.

To prevent severe damage to the system in case of single point failure, redundancy and backup mechanisms are provided to the key server nodes and multi-levels of storage. The "hot" backup and recovery policies reduce the time for data recovery to the least possible.

System performance, especially the real-time property, is guaranteed by the parallelism and the concurrency control polices. The reliability and confidentiality is ensured by the security mechanisms along with other aspects of the system architecture.

As a result, the system architecture and the mechanisms used for the handling of various issues inside the database system altogether provide the system the following properties:

- Performance. The system performance can be broken down to the real-time property, data update and retrieval performance it can provide. Our system

overall performance is achieved by means of the distribution of system workload to server nodes working in parallel, the reduced communication cost, and simplified control mechanism. The real-time property is assured by the transaction management and scheduling policies, and the "hot" recovery scheme.

- Reliability. The system is reliable because of the redundancy, backup and recovery mechanisms, and the security issues, which we have taken into full consideration during the stage of system design.
- Confidentiality. The combination of secure Operating Systems and the security policies which is applicable and appropriate to both the database application system and the operating systems makes the software system the most possibly secure.
- Scalability. From the data structure on to the system architecture, the system is highly scalable at every level. A personal computer running Linux Operating System can carry out the tasks we designed for all the server nodes, if we apply our design to a small system with limited number of vehicles running on limited area. Several mainframe servers together with huge numbers of high end server machines, and wide area networks can be involved if we intend to build a tracking system for large number of objects running nation or even worldwide.
- Flexibility. Even though the system is designed to have the above important properties, it is yet flexible. As technologies develop, or in case any problems are found in implementation of application stages, the change of any functional module and/or policies will basically need to affect just the involved modules or servers and it wouldn't hurt the overall functionalities of the system.

5 IMPLEMENTATION

In any fields of the real world, the implementation based on a design cannot cover the flaws inside the design, if there are any. However, a perfect system design based on perfect ideas, on the other hand, could be greatly ruined by a bad implementation. It is especially true in the development of software systems. To maximize the advantages of the data structure and system architecture presented in this dissertation, we need to carefully and correctly arrange the various details involved in the system implementation. This chapter discusses the issues closely connected to the implementation of a database system for tracking and retrieval of vehicles location information using the data structure and system architecture talked about in chapter 3 and chapter 4 of this writing.

We choose the Miami area and the vehicles running in this area as the objective of our system implementation. The designed system capacity is to cover the greater Miami area and all the vehicles that mainly run in this area. This area is chosen because of the size of it and its population, which implies the possible number of vehicles possessed. They are large enough to challenge the data structure and system architecture the application system is adopting, but not too large in terms of cost in hardware and workforce for the implementation. Yet such an implementation, if successful, could be extended without much technical difficulty.

In the rest of this chapter, we will start with the system requirements analysis, followed by the options and decisions on hardware, network and operating system platforms. The parameters configuration and implementation of the main functional modules are discussed thereafter. The performance of the finished system is also

evaluated. In the last section of this chapter, we will explore how to extend it to cover a nation or even the entire world and to include all the vehicles anywhere inside the covered areas.

5.1 System Requirements Analysis

We start with a series of simple calculations to get the order of the sizes of the data that will be stored in our database system. Since it is impossible to get the exact numbers, and what we need is only the order of these numbers, all the calculations are approximate. And the data are calculated based on the time range of 1 day, for those with time involved. We predefine the data update frequency from the vehicles to the data center to be once per second.

According to the US Census Bureau in "American Community Survey Profile 2002" [USCen], in year 2002 the population in the Miami and Fort Lauderdale area was 3,975,250. Assume every two people possess a car, and these cars are all registered to our system, then we will have a total of about 2,000,000 vehicles in our database system. Referring to appendix B, which is a map of greater Miami area, the size of this area is about 95x132km. Appendix C shows that the number of directed road segments in each square kilometer is around 200. The total number of road segments will be approximately 2,500,000.

Now we can approximate the average total number of cars that run through each road segment. The average length of each road segment is approximately 0.1 kilometers, since there are 200 directed road segments in each square kilometer. The standard annual mileage for a car is 12,000 miles. The daily average is 33 miles or 53 kilometers. The

total number of road segments a car runs through in a day, is 53/0.1=530. And the average number of vehicles that run through each directed road segment in each day, can be calculated as: $2{,}000{,}000 \times 530 \div 2{,}508{,}000 = 423$.

However, for those road segments on highway or crowded local roads, the daily number of vehicles passing through is much bigger. In the rush hours (6-9am, 4-7pm), there could be 1/2-1 cars entering a road segment in each second. The total number of cars entered in the rush hours is up to $6 \times 60 \times 60 \times 1$=21,600. The total number of cars entering such a road segment at other times of the day can be around this number. So the final number is around 43,000.

The possible size of the interval trees associated with each road segment is analyzed as follows. The data size for each single interval tree is at most $20n+4n\log n$ data units where n is the total number of time intervals in the tree. Suppose 4-byte integer and real data types are used, in the average cases, the size of an interval tree is around $(20n+4n\log_2 n) \times 4 = 16n(5+\log_2 n) = 16 \times 423 \times (5+\log_2 423) = 94{,}752$ bytes. While for the highway roads, the number goes sharply up to: $16n(5+\log_2 n) = 16 \times 43000 \times (5+\log_2 43000) \approx 14{,}500{,}000$ bytes. The maximum possible total data size for the indexing structure in a day is approximated to: $2{,}500{,}000 \times 94{,}752 \approx 237{,}600{,}000{,}000$ bytes ≈ 237GB. Even though the percentage of highway roads over all the roads is small, in practice the actual final number could be slightly higher considering the outstanding data size generated by this type of road segments.

Since each car runs 53km, we can approximate the total size of primary data stored in our database as follows. The time spent on road to cover the distance of 53km (33miles) should be within 1 hour in most of cases. The data size for each update is 24

bytes: 4 bytes each for Road ID, Time of Update, Easting, Northing, Speed and Distance from the beginning point of road segment. So $24 \times 60 \times 60 = 86{,}400$ bytes. And the total size of the primary data in a day is $2{,}000{,}000 \times 86{,}400 = 172{,}800{,}000{,}000 \approx 172$GB.

About the road segments, we have the following original and/or calculated numbers. The size of the data defining each undirected road segment is 32 bytes: 1 integer for Road ID, 2 real numbers for start point geolocation, 2 real numbers for end point geolocation, 1 integer for UTM zone, 1 real number for total length and 1 integer for speed limit. For the total of 2,500,000 directed segments, which is approximated to 1,254,000 undirected roads, the total data size is $32 \times 1{,}254{,}000 = 40{,}128{,}000$ bytes $\approx$ 40MB. Referring to section 3.3.3.2, when there are n total road segments the size of the segment tree is no larger than $32n+24n\log_2 n$. So the indexing structure for the road segments could take a total size of $32 \times 1{,}254{,}000 + 24 \times 1{,}254{,}000 \times \log_2 1{,}254{,}000 = 672{,}144{,}000 \approx 672$MB.

On the topology of the road map, as we have discussed in chapter 3, it is represented in the format of an array. The last cell for each directed road segment is a pointer pointing to a linked list. This linked list keeps the ID's of the vehicles that are currently running on this road segment. The average number of vehicles that run on a road segment at any time is at most $2{,}000{,}000 \div 2{,}500{,}000 \approx 1$, since there are a total of 2 million cars and about 2.5 million road segments. However in the extreme case, e.g. when there is traffic jam, the total number of vehicles can be up to the product of the total number of lanes and the length of the road segment divided by the length of a car. Inside a 1 mile highway with 4 lanes, there could be as many as $4 \times 1609.3 \div 6 = 1073$. (The average length of a car is about 5 meters; 1 meter is added as the distance between two vehicles.)

A vehicle doesn't send any updates to the data center when it is not moving at all. In the rush hours, most vehicles are moving and hence the total data being sent to data collecting server is at the highest volume at these times. The extreme case is when all the vehicles are running. In this case, the data are rushing from vehicles to the data collecting server at the speed of: $24 \times 2{,}000{,}000 = 48$M Bytes/second. This only considers the data meaningful to our system. To receive this amount of useful data, more data are actually transmitted through the wireless channels. However, since the average time a vehicle spends on road is within 1 hour, the average data flow is 1/24 of the above extreme case, which is 2M bytes/second.

The data flow from servers to servers inside the data center can be estimated as follows. Vehicles' speed changes mostly happen at stops and re-starts. Normally the total time a vehicle spends on stops and re-starts is less that when it is running at even speeds. After preprocessing at the data collecting server, the data updates from vehicles should be reduced to half or less of the original size. So the data updates flow from server to server is within $48\text{M} \div 2 = 24$M bytes/second. And the average is 1M bytes/second. The amount of data transferred for query answering depends on the data range defined by the queries.

Based on the above analysis, the system requirements can be abbreviated as:

- 172GB raw data comes in each day.
- Indexing structure can take up to 237GB per day.
- Maximum of 43,000 vehicles run on a directed road segment in a day, and the average number is 423.
- Maximum of 1073 vehicles can be in the same directed road segment at same time, and the average number is 1.

- Data transmit speed from vehicles to data collecting server can be up to 48M bytes/second, the average is within 2M bytes/second.
- Transfer rate for data update between servers inside data center ranges from 1M bytes/second or less to 24M bytes/second.

Besides the above analysis that measured in numbers, there are other system requirements which mainly include:

- Incoming data need to be properly processed so that all meaningful data are kept in database.
- Data update need to be processed in a timely manner such that instead of obsolete data, the "current data" in our system reflect the true situation in the real world.
- Queries on both current and past data should be answered promptly, while priority given to current queries at times of conflicts.

5.2 Setup the Parameters

To meet the system requirements and maximize the performance of the application system, we need to properly set up the following parameters: the time granularity *chron*, the change of speed δ_1 and the change of acceleration δ_2 as discussed in section 3.1, and the time range for the indexing trees. Based on the system requirements analysis results provided in last section, these parameters are set as follows.

- *Chron*. It is decided by the data update frequency. We set it to 1 second. Considering the overhead over the communication channels, 1 second per update

means 24M bytes data could be sending to the data center every second. This is even a great challenge for local networks. On the other hand, it is ideal to set it to ¼ to 1/3 second so that the distance between two consecutive updates is around twice the GPS data error range. As for the data processing, it basically is possible to handle data coming in 24M bytes per second. So 1 second is the number after balancing all the affected parties.

- δ_1 and δ_2. The math is based on the fact that the omit of speed change which results in the distance difference within the GPS error is acceptable. The GPS error is about 3 meters, and the data update interval is one second. 3 meters per second is about 6 MPH in speed. The highest known speed limit is 75 MPH. $\delta_1 = 6 \div 75 = 0.08$. Such a number for δ_1 is large if a highly accurate GPS system is used in the future, so we don't set up a value for δ_2 and the related calculation will be measured by the production of δ_1 and the Speed Limit of the roads only.
- Time range for each indexing tree. The time interval trees associated to each directed road segment have a space complexity of $n\log n$. We can't do anything about the n inside this formula, but the $\log n$ part is what we want to eliminate. Considering the nature of the road traffic distribution over time, such as there are rush hours during the day time, and at late night there are very little traffic on any road, we can break it down into days and build separate trees for different time ranges. The basic unit is day, and the separator can be set at 2 O'clock in the morning since there is the lowest traffic at this time. This way $\log n$ can never grow to too large. Since the total amount of data for one day, as analyzed in last section, is not small in our case, one day is good for our specific case. Such a

setting will result in the average indexing tree size of about 30K and the largest up to 5M bytes.

5.3 Hardware Platform and Operating System

The system architecture described in chapter 4 can be implemented, due to its scalability, with a few high end servers such as IBM mainframe, some powerful workstations, or a relatively big number of PC servers. To show the system organization and load balance mechanisms in the distributed system design, PC servers are chosen as the hardware platforms for servers at all functional nodes in our system. The current high end PC servers are powerful enough, in terms of computing capability, to do the data processing, since the amount of data calculation needed in our system is not huge. However, due to the requirement of the main memory residing road networks and the most current motion of the vehicles, sized at the order of 10^2 MB each, high volume of main memory is required for the servers. A minimum of 1GB is necessary for most of the servers, and 2GB is suggested for the critical ones, such as the servers that handling the current data.

For the system security consideration, OpenBSD is the ideal choice as Operating System. The fact that OpenBSD is open source, which makes it available to us at no cost, is another reason for us to favor OpenBSD.

Now the compatibility of OpenBSD and Personal Computers becomes the key factor to decide the feasibility of implementing our system on PC servers running OpenBSD.

OpenBSD/I386 is the PC version of OpenBSD. It runs on the standard PC's and clones [OBSD], with a wide variety of processors, I/O bus architectures and peripherals

supported. Our concerns of the Operating System supported hardware mainly include the processors, I/O buses, hard disks, RAID storage; tape drives, and network connection which include wired and wireless LAN/WAN support. The currently OpenBSD supported PC hardware of our concern can be summarized as below:

- The supported processors include Intel, AMD, Cyrix and other series;
- Supported I/O bus architectures include All standard ISA/EISA/VLB/PCI buses, 16-Bit PCMCIA PC Cards, 32-Bit CardBus PC Cards and Universal Serial Bus.
- Hard disk controllers: ISA MFM, ESDI, IDE, and RLL.
- Various RAID and Cache Controllers.
- Tape Drives include: most SCSI tape drives and tape changers, QIC-02 and QIC-36 format tape drives.

Network connection: 10/100Mbps and Gigabit Ethernet, wireless, ATM and FDDI interfaces.

OpenBSD supports server and workstation product lines from vendors such as Hewlett-Packard, Motorola, Sun Microsystems, and the Alpha-based and VAX-based systems from Digital. So it is possible to put some workstations besides the PC's in our system if the budget permits to do so. Otherwise, the migration of partial or all of the hardware system to any higher products in the future wouldn't cause any problem in system architecture, and there is basically no need to re-write the source code after hardware upgrade.

The current 1G bps local network fits all other I/O devices inside a PC well in terms of bandwidth. Vehicles can be grouped if network bandwidth becomes a bottleneck, and

more than one machine are used so that each machine accepts and processes only data from vehicles of certain group(s).

The bottleneck for query answering is with the hard disk access. Our indexing mechanism and distributed system architecture, along with fast access hard disks, altogether ensure that queries be processed promptly. When distributing the data load, both the primary data and indexing structure are distributed evenly to the different access servers. Primary data are distributed according to the vehicles, and indexing structures are distributed according to geolocation of the road segments.

5.4 Software Development

C/C++ is widely adopted for system and application software development due to its portability and compatibility. Even though it is frequently criticized for its lack of features such as ease of learning, variable default initiation, and garbage collection, these are not a problem for those true computer professionals. The reason that C/C++ is chosen as the programming language for our software system implementation is, instead of its popularity, its efficiency in implementing functional modules that access and/or manage the I/O devices. To implement software that frequently accesses I/O devices, e.g. hard disks and network ports, C/C++ is the choice for all reasons.

The data flow in our database system is shown in Figure 5-1. Besides the data flow shown in the figure, the control flow is maintained complying with the scheduling policies by means of communication among different servers.

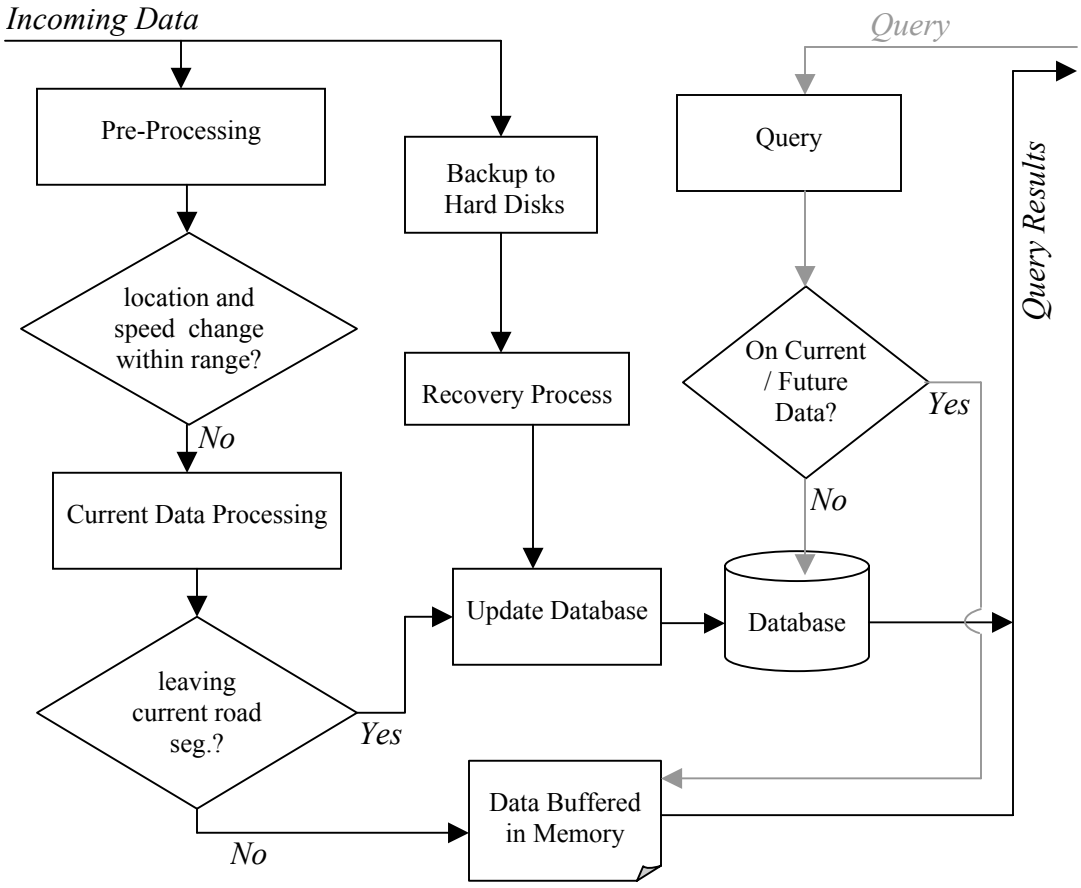


Figure 5-1 Data Flow in Vehicles Tracking System

The software implementation mainly includes the development of the following modules: incoming data processing, query processing, backup and recovery handling, data update/query algorithms, storage management, network and communication. The rest of this section will briefly address the key functional modules of our vehicle location information tracking system running on OpenBSD using C/C++ programming language.

- Algorithms. The algorithms for insert/search/pathfinder were given in chapter 3. The implementation of this part is basically to convert this abstracted code into

C/C++ format. Special care needs to be taken to the parts involving I/O access to storage.

- Communication and Central Control Services. Communication among servers is implemented by socket programming. To reduce network traffic as well as data transfer delay, queries are numbered before they are forwarded to one of the servers for answers and the results are sent back directly from the server which carries out the query process.

  Inside the central control services, there is a directory on data and service locations along with various system policies. The Central Control Server provides services such as transaction scheduling and load balancing, and ensures some of the security policies.

- Incoming Data Processing. The main goal of this module is to reduce as much as possible for the next phase of processing, while maintain the property that the vehicle's motion trajectory can be simulated within the required accuracy.
- Query Processing. It parses and interprets the queries from clients, forwards it to the corresponding server for answer, finalizes the result and sends it back to clients.
- Storage Management. Storage in our system exists in the forms of main memory, local hard disks, network file systems, and tapes. The most current data is kept in main memory. Recent data are in local and network hard disks. And past data are stored offline in tapes. Each vehicle's motion can be managed as separate files. But it could be very inefficient if the indexing trees are stored on hard disks in

the format of files. To update and access the indexing structure, direct hard disk access is necessary.

- Backup and Recovery. The data logs are the records of the incoming data in the time order of their arriving. These logs are written to hard disks as soon as data comes in. The backup from existing database is a copy of the database ranging certain time span stored in hard disks or tapes.

  Restore from backups is relatively easy, since it only needs to put the data online and update the data directory. Restore from data logs needs to go through the same process as the incoming data from vehicles. Since the data under restoring are old comparing to the newly collected data, the restore processes have lower priorities in case of conflict.

5.5 Conclusion

This chapter discussed the implementation of a vehicles location information tracking system based on the real world example of the greater Miami area. Starting with the system requirements analysis, issues from the system parameters setting, the choosing of hardware and operating system, to the software development are all briefly addressed. Research and analysis shows that such a tracking system covering the greater Miami area with all the vehicles mainly running in this area included can be implemented with about 15-20 high end PC servers and necessary persistent storage. OpenBSD is recommended for the Operating System based on which the software will be developed. C/C++ is chosen as the programming language.

6 OTHER CONSIDERATIONS

The system architecture described in chapter 4 is for an independent database application system. The data and indexing structures described in chapter 3 were originally targeted at and designed for the real world model of moving objects on constrained road networks in a 2-d plane. Is it possible to expand or embed the system design and the indexing mechanisms to a database management system, so that they can be more powerful and more widely applicable? Can we extend the data and indexing structures to manage objects moving freely in higher dimensional space without constraints on their moving trajectory? With a huge amount of vehicles' movement data stored in our databases, could we make more use out of it, to benefit other research and application fields such as traffic management and control?

This chapter explores these possibilities and ways to extend the application model and the data structures we proposed earlier. Section 6.1 discusses how to embed our data structure into Sem-ODB. Section 6.2 shows methods to extend the indexing mechanism. Section 6.3 explores ways to data warehouse the databases manipulated by our approach.

6.1 Embedding the Data Structure into Sem-ODB

Sem-ODB is a semantic object oriented database model originally proposed by Rishe [Rish92]. A database management system has been developed by the High Performance Database Research Center at Florida International University. Various GIS and other applications have been built and still are successfully running on top of the Sem-ODB DBMS.

The accessibility and the familiarity of Sem-ODB make it a natural choice for the expansion of our data structure and application model. But there are other reasons as well. The introduction of semantics into a DBMS improves the efficiency of data management and reduces storage cost. The indexing mechanism (for traditional static data) in Sem-ODB is efficient and uncomplicated. High flexibility is provided to the application system and the developers by not requiring a key for data of any category, which is equivalent to a table in the relational model. These characteristics of Sem-ODB make it suitable for the management of static and some discrete data types for a vehicles location information tracking system.

Embedding our data structure and system architecture into Sem-ODB would provide the following benefits for the system designers and developers:

- Efficient management of static data,
- No need to develop a new query interpreting service,
- Easy incorporation of our transaction scheduling mechanism,
- Easy implementation of Role based security policies,
- Rich Application Programming Interface (API)
- Many existing tools.

To do this embedding, care needs to be taken to key issues including dealing with different indexing structures, transaction scheduling and concurrency control policies, and processing of spatio-temporal queries. The API may need to be expanded as well to meet the program needs of accessing spatio-temporal data. These issues can be handled as follows:

- Embedding the Indexing Structure. Only the data on continuously moving objects motion is indexed using the STAPS approach. Other data, including static and discretely changed data, are under the control of Sem-ODB's traditional management.
- Transaction and Concurrency Control. The updates of static data always yield the real-time updates, except for those involving security role changes.
- Expanding the API. The API for data updates on sptio-temporal data are allowed for system programs; users never are given permission to call these functions/interfaces.
- Adding Spatio-temporal Query Types. The parser and interpreter need to be updated, and SQL expressions need to be enriched.

6.2 Extend The Indexing Structure

Even though they were designed for the application model of vehicles moving on road networks, the data and indexing structures are applicable to objects moving in higher dimensions with or without constraints. This section briefly discusses how the indexing schemes work in an expanded space.

6.2.1 Indexing Objects Moving in Free 2-d Space

An IET is defined in chapter 3 as ***any*** event that triggers the update of the indexing structure. Corresponding to the definition of IET, the partition of the space is not necessarily restricted to road segments or any of the constraints on the objects' motion. Even for objects moving on road networks, the IET(s) can be defined as other events than

the entering and leaving of a road segment. The simplest way to partition the 2-d space so that the STAPS approach can be applied is to evenly grid the plane into equal rectangular areas.

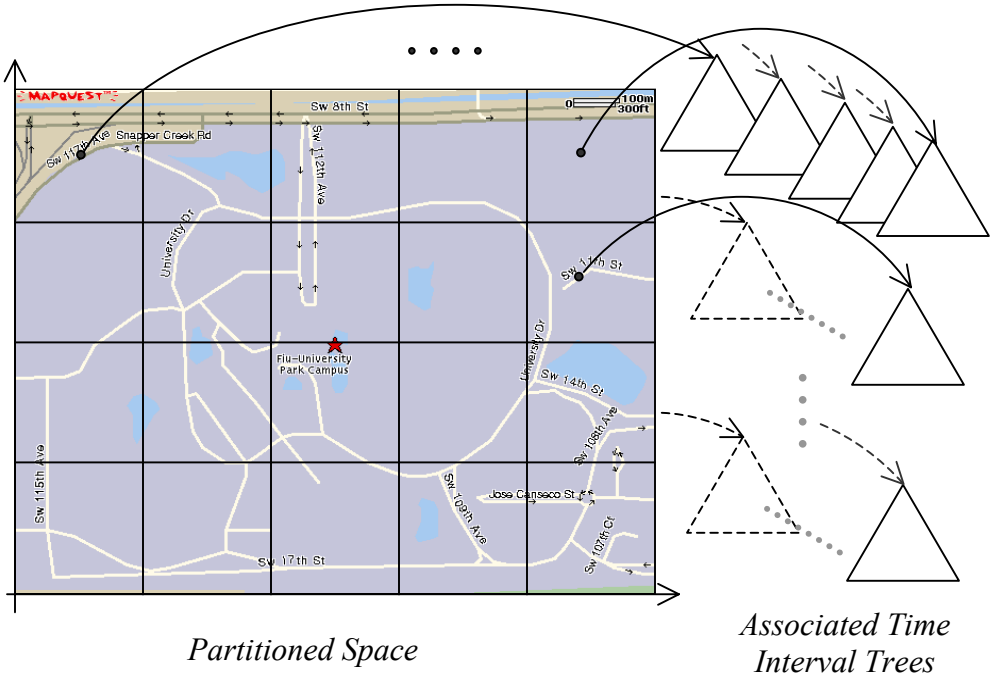


Figure 6-1 Building Indexing Structure for Objects Moving in Free 2-d Space

Figure 6-1 shows how such a space partition policy works. The IETs in this case are defined as the objects' entering and leaving of the rectangular area. When there is no restriction on objects movement, there is no need to index anything like the road segments as in the case we discussed in earlier chapters. The locating of a given query point or area into the grids in the plane can be easily calculated by their coordinates and the boundaries of the grids.

Queries of all types are processed similarly as in the moving vehicles on road networks case that we discussed in chapter 3. For range queries of type Q_4, the I/O

complexity is $O(m\log T + k)$, where m is the number of grids (partitions) that intersect the given query space range, T is the total number of time intervals associated to this space grid, and k is the total number of objects that appear in the space range during time $[t_1, t_2]$. This fact shows that the STAPS approach works for objects moving in free 2-d space at least as well as when the objects' movement is restricted to any road networks in the plane.

Notice that the IETs are not required to be uniformly defined even in the same application system. The involved space range doesn't have to be rectangular or any regular geometric shapes either.

6.2.2 Indexing Objects Moving in Higher Dimensional Space

The work on extending the STAPS approach to manage continuously moving objects in higher dimensional space becomes trivial after it is successfully extended to free 2-d space. For objects moving in 3-dimensional space or higher dimensional hyper plane, an efficient indexing structure can be built by defining an appropriate space partitioning rule, choosing an applicable static access method for the manipulation of the space partitions, and building an interval tree to handle the accesses to the associated time intervals. The easiest space partitioning method is to evenly grid it into hyper rectangles. And just as in the 2-d space cases, there are no restrictions to the space partitioning methods as long as any point in the space is assigned to, and mostly only to, one partition.

Observe that in both the 2-d and higher dimensional cases, one of the critical steps in applying the STAPS approach to build and manipulate the indexing structures is to

choose an appropriate space partitioning method. In the processing of a range query, the I/O complexity of $O(m\log T+k)$ greatly depends on how the space is partitioned. The larger each partition is, the smaller m, but the larger T and k will be. The system designers have to balance them considering the specific situation of the application.

The indexing structures for both the space partitions and the segmented time intervals can be any kinds which are appropriate to the application model. The system performance depends on how well the chosen indexing structures fit the criticality of different kinds of operations to the system. In the real-time vehicles location information tracking system, the most critical tasks are the data updates, that is why in our example application case a structure is chosen to provide the best update performance $O(1)$ I/Os, on the average.

6.3 Data Warehousing

In chapter 5, we have analyzed the data size a region wide moving vehicles location information tracking system would generate in each day. As time passes, the data stored in the system would grow quickly, and soon it would become more like a data warehouse than a simple location information database. These data could provide valuable support to research and applications in the area of traffic control, such as traffic simulation [Trim, Cors], intelligent transportation systems [ITS02, DOT03], and real-time traffic information systems [Geor, Wash].

The mechanisms inside STAPS could be adopted for further analysis of the data stored in such a data warehouse to be used by such research and application projects. The data analysis could be done by putting IETs to the data types of concern and applying the

indexing structures and/or the data analysis algorithms whenever it is triggered. The data to which the IETs are associated could be any types, not just space. The data analysis could be done either in real-time, if it doesn't compromise the application's time-critical processes, or in background and/or offline by reviewing stored data.

7 CONCLUSION

In this dissertation we have explored data structure, indexing mechanisms, system architecture, and implementation issues involved in a real-time database system of continuously moving objects based on the real-world application model of a location information tracking and retrieval system for vehicles moving on road networks. We have proposed the Segmented Time Associated to Partitioned Space (STAPS) approach for the processing of continuously moving objects databases. A data and indexing structure has been designed for the real-time location information tracking and history retrieval system for vehicles moving on road networks using the STAPS method. Based on this data and indexing structure, a distributed system architecture has been presented. Targeting at a possible real-time database application system that manipulates all the vehicles moving in the greater Miami area, the main implementation issues have been addressed based on the system requirements analysis. The possibilities and means of extending the indexing structure have been briefly discussed as well.

The STAPS approach well represents the continuity of the objects movement by partitioning the space where the objects move in and segmenting the time into intervals during which an object is inside a space partition. There is no need for any assumptions on objects' moving trajectories and speed. The indexing structures for space partitions and time intervals are not predefined. Instead system designers can choose or design the structures that best fit their specific system requirements. STAPS is a general approach for manipulating databases on objects of any kinds that continuously move in any sort of free or constrained space.

Designed for the database systems of vehicles moving on road networks, the road networks are represented by a set of directed road segments indexed by a segment tree, and a balanced interval tree structure is used to index time segments associated to the road segments. These two structures working together can achieve ideal data update complexity of average $O(1)$ *I/O*s to meet the application system's real-time requirements, and *I/O* cost on most of the search operations is within the order of logarithm.

A distributed system architecture with centralized organization is presented for a real-time database system on moving vehicles on regional road networks. The key problems for a real-time database system, such as the real-time property, transaction and concurrency control, storage management, backup and recovery, and the system security, are discussed and solutions are given. The detailed implementation related issues involved in the development of such application systems are itemized based on the system requirement analysis on the real world scenario of moving vehicles in greater Miami area, and a suggested total solution is provided.

7.1 Dissertation Contributions

The main contributions of this dissertation research are concluded as follows:

1. STAPS is the first known systematic approach for building indexing structures for Continuously Moving Objects databases. Instead of being taken as another spatial dimension, time is treated separately from the spatial dimensions. The time dimension and the space dimensions interfere with each other at the points of IETs. This way the continuity of the objects movement is well represented.

Hence data and indexing structures developed applying the STAPS approach are potentially more efficient than those developed in the traditional ways.

2. The Segment Tree structure [Bent77] is adopted and carefully re-modified to fit the application model of real-time moving vehicles on road networks. The indexing structures with proven *I/O* complexity meet the system's real-time requirements with acceptable search performance. They can be extended to higher dimensional and/or free moving space without compromising the efficiency provided to the cases of 2-dimensional road networks constrained space.
3. The proposed indexing structure supports data insertions at worst case $O(c\log n)$ *I/O* cost with very small constant c, and the average *I/O* cost for data updates in the vehicles moving on road networks case is $O(1)$. The real-time data updates property is ensured.
4. An architecture and detailed system design is provided for the implementation of a real world application model – real-time integration of a location information tracking and retrieval system for vehicles moving on (region wide) road networks.

7.2 Future Work

This dissertation research has addressed most of the core issues involved in a moving objects database. Solutions are provided to the specific case of moving vehicles on road networks. However, there are yet important issues, which are either not discussed or not fully developed. Our next step will be to work on the implementation of the database

system on vehicles moving on road networks, using the data structure, system architecture and design presented in this dissertation. Then we will focus on the tuning of the searching performance on some types of queries, and the development of algorithms to answer other important types of queries, such as nearest neighbor. To itemize, the future work includes:

- System Implementation. To implement the proposed application system, and then embed the data and indexing structure to Sem-ODB is part of the main concerns in the near future plan.
- Reducing the *I/O* costs on range queries. Based on the experimental results from the running application, parameters can be adjusted to achieve better I/O performance for queries, especially the range queries.
- Nearest Neighbor Search. In practice, this family of queries are useful for companies or government agencies to do their scheduling and/or planning work. In research, it is a topic of interest to people in several research areas.
- Explore the possibility of a systematic method to partition the space and define the IETs. After or at the same time that the above tasks are carried out, we can work on the space partition methods to improve the overall system performance and speed the data structure and algorithms design for different kinds of MODBs and related applications.

BIBLIOGRAPHY

[AEG98] P. K. Agarwal, J. Erickson, L. Guibas. Kinetic Binary Space Partitions for Intersecting Segments and Disjoint Triangles. In *Proc. 9th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 107-116, 1998.

[AAE00] P. Agarwal, L. Arge, J. Erickson. Indexing Moving Points. *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000.

[AAV01] P. Agarwal, L. Arge, J. Vahrenhold. Time Responsive Indexing Schemes for Moving Points. *Workshop on Algorithms and Data Structures*, 2001.

[AG04] V. T. Almeida, R. H. Güting. Indexing the Trajectories of Moving Objects in Networks. *Technicla Report 309, Fernuniversität Hagen, Fachbereich Informatik*, 2004.

[ARR00] F. Araújo, B. Ribeiro, L. Rodrigues. A Neural Network for Shortest Path Computation. DI–FCUL TR–00–2, Departamento de Informática Faculdade de Ciências da Universidade de Lisboa Campo Grande, April 6, 2000.

[ASV99] L. Arge, V. Samoladas, J. S. Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing. In *Proc. 18th Annu. ACM Sympos.Principles Database Syst.*, 346-357, 1999.

[Arg01] L. Arge. Some Algorithmic Research Challenges and Opportunities in Geospatial Applications. 2001.

[Arm01] M. Armstrong. The Four Way Intersection of Geospatial Information and Information Technology. A White Paper Prepared for the *Workshop on the Intersection of Geospatial Information and Information Technology*, 2001.

[BGZ97] J. Basch, L. Guibas, L. Zhang. Proximity Problems on Moving Points. *Proceedings of the 13th Annual Symposium on Computational Geometry*, 1997.

[BGH99] J. Basch, L. Guibas, J. Hershberger, Data Structures for Mobile Data, J. Algoritms, 31(1):1-28, 1999.

[Bas99] J. Basch. Kinetic Data Structures. *PhD Thesis*, Computer Science Department, Stanford University, 1999.

[Bent77] J. L. Bently. Solutions to Klee's Rectangle Problems. *Technical Report*, Carnegie-Mellon Univ. Pittsburgh, PA, 1977.

[BP00] K. Beard, H. M. Palancioglu. Estimating Positions and Paths of Moving Objects. *IEEE Seventh International Workshop on Temporal Representation and Reasoning*, 2000.

[BGOW93] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. On Optimal Multiversion Access Structures. In *Proc. of Symposium on Large Spatial Databases*, Vol. 692, 1993.

[BGOS96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, Vol.5, No.4, pp.264-275, 1996.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *In Proceedings of ACM SIGMOD Int'l. Conf. on Management of Data*, Vol.19 Issue.2, pp.322-331, 1990.

[BJKS02] R. Benetis, C. Jensen, G. Karciauskas, S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *IDEAS*, 2002.

[BKOS00] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, *Computational geometry: Algorithms and Applications* (*Second Edition*), Springer-Verlag, 2000. pp. 223-229.

[BM??] A. Boroujerdi, B. Moret. Persistence in Computational Geometry. ??

[Cha86] B. Chazelle. Filtering search: a New Approach to Query-Answering. *SIAM J. Comput.,* 15(3):703--724, 1986

[CC02] Y. Choi , C. Chung. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[CR99] J. Chomicki, P. Z. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. In *Proc. 6th Intl. Workshop on Time Representation and Reasoning*, 41-46, 1999.

[CAA01-1] H. Chon, D. Agrawal, A. Abbadi. Using Space-time Grid for Efficient Management of Moving Objects. *Second ACM Iinternational Workshop on Data Engineering for Wireless and Mobile Access*, May 2001.

[CAA01-2] H. Chon, D. Agrawal, A. Abbadi. Storage and Retrieval of Moving Objects. In *Proceedings of the International Conference on Mobile Data Management*, 2001.

[CAA02] H. Chon, D. Agrawal, A. Abbadi. Query Processing for Moving Objects with Space-Time Grid Storage Model. *Third International Conference on Mobile Data Management*, pp.121, 2002.

[CCT93] J. Clifford, A. Crocker, A. Tuzhilin, On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models. *Temporal Databases: Theory, Design and Implementation*, pp. 496-533, 1993.

[Cors] CORSIM Overview at *http://www.fhwa-tsis.com/corsim_page.htm*.

[CFGN01] J. Cotelo, L. Forlizzi, R. Güting, E. Nardelli, M. Schneider. Algorithms for Moving Objects Databases. FernUniversität Hagen, Informatik-Report 289, October 2001.

[DOD01] U. S. Department of Defense. *Global Positioning System Standard Positioning Service Performance Standard*. http://www.navcen.uscg.gov/gps/geninfo/2001SPSPerformanceStandardFINAL.pdf.

[DOT03] U.S. Department of Transportation, Federal Highway Administration Next Generation Simulation Program: *http://ops.fhwa.dot.gov/Travel/Traffic_Analysis_Tools/ngsim.htm*.

[EGSS01] J. Eisenstein , S. Ghandeharizadeh , C. Shahabi , G. Shanbhag , R. Zimmermann. Potpourri: Alternative representations and abstractions for moving sensors databases. *Proceedings of the 10th International Conference on Information and Knowledge Management*, 2001.

[EN00] R. Elmasri, S. Navathe. *Fundamentals of Database Systems, Third Edition*. Addison-Wesley, 2000.

[ES02] M. Erwig, M. Schneider. Spatio-Temporal Predicates. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 4, pp881, 2002.

[ESRI99] ESRI, *ArcView Tracking Analyst: Complete Tracking Solutions*, May 1999.

[FGNS00] L. Forlizzi , R. Güting, E. Nardelli, M. Schneider. A Data Model and Data Structures for Moving Objects Databases. *ACM SIGMOD Record , Proceedings of the 2000 ACM SIG MOD International Conference on Management of Data*, Vol. 29 Issue 2, May 2000.

[FGGJ99] A. Frank, S. Grumbach, R. Güting, C. Jensen, M. Koubarakis, N. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H. Schek, M. Scholl, T. Sellis, B. Theodoulidis, P. Widmayer. Chorochronos: a Research Network for Spatiotemporal Database Systems. *ACM SIGMOD Record*, Vol 28 Issue 3, 1999.

[Fren03] E. Frentzos. Indexing Objects Moving on Fixed Networks. *In Proc. of the 8th International Symp. on Spatial and Temporal Databases*, pages 289-305, 2003.

[GN93] S. Gadia,S. Nair. Temporal Databases: A Prelude to Parametric Data. *Temporal Databases: Theory, Design, and Implementation*, pp. 28-66, 1993.

[Garm] *http://www.garmin.com/*.

[Geod] *http://www.geodan.nl/uk/*.

[Geor] *www.georgianavigator.com*.

[GSS01] R. Greenwald, R. Stackowiak, J. Stern. *Oracle Essentials : Oracle9i, Oracle8i & Oracle8 (2nd Edition)*. O'reilly, 2001.

[GFPH01] T. Griffiths, A. Fernandes, N. Paton, B. Huang, M. Worboys, C. Johnson, K. Mason, J. Stell. Spatiotemporal Databases: Tripod: a Comprehensive System for the Management of Spatial and Aspatial Historical Objects. *Proceedings of the 9th ACM International Symposium on Advances in Geographic Information Systems*, 2001.

[GRS98] S. Grumbach, P. Rigaux, and L. Segoufin, Spatio-Temporal Data Handling with Constraints, In *Proc. Sixth ACM Int'l Symp. Advances in Geographic Information Systems*, pp. 106-111, 1998.

[Gut84] A Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the SIGMOD Conference*, 1984.

[GBEJ00] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems (TODS)*, Volume 25 Issue 1. pp1-42, March 2000.

[HKTG02] M. Hadjieleftheriou, G. Kollios, V. Tsotras, D. Gunopulos. Efficient Indexing of Spatio-temporal Objects. *EDBT*, 2002.

[HKT02] M. Hadjieleftheriou, G. Kollios, V. Tsotras. Efficient Indexing of Spatiotemporal Objects. *EDBT*, 2002.

[IKK02] Y. Ishikawa, H. Kitagawa, T. Kawashima. Continual Neighborhood Tracking for Moving Objects Using Adaptive Distances. *Proceedings of the International Database Engineering and Applications Symposium*, 2002.

[ITS02] The Intelligent Transportation Society of America. *National Intelligent Transportation Systems Plan: Ten-Year Vision*. Jan. 2002.

[Jen02] C. Jensen. Research Challenges in Location-Enabled M-Services. *Third International Conference on Mobile Data Management*, Singapore, pp.3, January 2002.

[JBBS01] M. Jones, C. Berkley, J. Bojilova, M. Schildhauer. Managing Scientific Metadata. *IEEE Internet Computing*, pp. 59-68, 2001.

[JP02] S. Jung, S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, pp.1029-1046, 2002.

[KL00] C. Kleiner, U. Lipeck. Efficient Index Structures for Spatio-Temporal Objects. *The 11th International Workshop on Database and Expert Systems Applications*, pp. 881, Sep 2000.

[KGT99-1] G. Kollios, D. Gunopulos, V. Tsotras. On indexing mobile objects. *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1999.

[KGT99-2] G. Kollios, D. Gunopulos, V. Tsotras. Nearest Neighbor Queries in a Mobile Environment. *Spatiotemporal Database Management*, 119-134, 1999.

[KM00] F. Korn, S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, Volume 29, Issue 2, 2000.

[KY00] J. Kuchar, L. Yang, A Review of Conflict Detection and Resolution Modeling Methods. *IEEE Transactions on Intelligent Transportation Systems*, Vol. 1, No. 4, 2000.

[KM00] T. Kuo, A. Mok. Real-Time Data Semantics and Similarity-Based Concurrency Control. *IEEE Trans. Computers 49(11)*: pp.1241-1254, 2000.

[KLL02] D. Kwon, S. Lee, S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. *Third International Conference on Mobile Data Management*, pp. 113, Jan. 2002.

[LULC01] K. Lam, Ö. Ulusoy, T. Lee, E. Chan, G. Li. An Efficient Method for Generating Location Updates for Processing of Location-Dependent Continuous Queries. *7th International Conference on Database Systems for Advanced Applications*, pp. 0218, 2001.

[LK01] K. Lam, T. Kuo. *Real-Time Database Systems Architecture and Techniques*. Kluwer Academic Publishers, 2001.

[Lee96] D.T. Lee. Computational Geometry. *ACM Computing Surveys*, 1996.

[MU99] Y. Masunaga, N. Ukai. Toward a 3D Moving Object Data Model -- A Preliminary Consideration. *International Symposium on Database Applications in Non-Traditional Environments* (*DANTE'99*), pp. 302, Nov. 1999.

[MSI02] H. Mokhtar, J. Su, O. Ibarra. OLAP and Constraints: On Moving Object Queries. *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002.

[MRA00] J. Moreira , C. Ribeiro , T. Abdessalem. Query Operations for Moving Objects Database Systems. In *Proceedings of the Eighth ACM International Symposium on Advances in Geographic Information Systems*, ACM Press New York, NY. pp.108-114 ISBN:1-58113-319-7, 2000.

[MRS99] J. Moreira, C. Ribeiro, J.-M. Saglio. Representation and Manipulation of Moving Points: an Extended Data Model for Location Estimation. *Cartography and Geographic Information Science*, 26 (1999) 109-123.

[Nav] NavTech. *NavStreets Product Guide.*2003.

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, Vol. 9, No. 1, pp. 38--71, 1984.

[Nova] *http://www.novatel.com/About_Us/racingfx.html*

[OBSD] http://www.openbsd.org/

[Ove87] M. Overmars, Computational Geometry on a Grid -- An Overview. 1987.

[PKGT02] D. Papadopoulos, G. Kollios, D. Gunopulos, V. Tsotras. Indexing Mobile Objects on the Plane. *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, 2002.

[PSZ99] C. Parent, S. Spaccapietra, E. Zimányi. Spatio-temporal Conceptual Models: Data Structures + Space + Time. *Proceedings of the seventh ACM International Symposium on Advances in Geographic Information Systems*, 1999.

[PT00] D. Pfoser, Y. Theodoridis. Generating Semantics-Based Trajectories of Moving Objects. *Proceedings Workshop on Emerging Technologies for Geo-Based Applications*, Ascona, Italy, 2000.

[PJ01] D. Pfoser, C. Jensen. Querying the Trajectories of On-line Mobile Objects. *Second ACM International Workshop on Data Engineering for Wireless and Mobile Access*, May 2001.

[PJ03] D. Pfoser, C. S. Jensen. Indexing of Network Constrained Moving Objects. *Proceedings of the eleventh ACM International Symposium on Advances in Geographic Information Systems*, November 2003.

[PS01] E. Pitoura, G. Samaras. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, pp. 571-592, July 2001.

[PSR99] R. Price, B. Srinivasan, K. Ramamohanarao. Spatiotemporal Extensions to Unified Modeling Language. *10th International Workshop on Database & Expert Systems Applications*, pp. 460, 1999.

[Rish92] N. Rishe. *Database Design: The Semantic Modeling Approach*. McGraw-Hill, 1992.

[RGMS] J. Roddick, F. Grandi, F. Mandreoli, M. Scalas. Towards a Model for Spatio-Temporal Schema Selection. *10th International Workshop on Database & Expert Systems Applications*, pp. 434, 1999.

[RN95] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[SM99] J. Saglio, J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *The 10th International Workshop on Database & Expert Systems Applications*, pp. 426, 1999.

[Sam84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR)*, Volume 16 Issue 2, 1984.

[SS93] A. Segev, A. Shoshani. A Temporal Data Model Based on Time Sequences. *Temporal Databases: Theory, Design and Implementation*, pp. 248-270, 1993.

[SG93] A. Segev, H. Gunadhi. Efficient Indexing Methods for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, pp. 496-509, 1993.

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos. The R+-tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, pp. 507, 1987.

[SDK01] A. Seydim, M. Dunham, V. Kumar. Location Dependent Query Processing. *Second ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2001.

[SKTA01] C. Shahabi, M. Kolahdouzan, S. Thakkar, J. Ambite, G. Knoblock. Distributed, Web-based GIS: Efficiently Querying Moving Objects with Pre-defined Paths in a Distributed Environment. *Proceedings of the 9th ACM International Symposium on Advances in Geographic Information Systems*, 2001.

[SF96] S. Shekhar, A. Fetterer. Path Computation in Advanced Traveler Information Systems. In *Proc. Intelligent Transportation Systems*, 1996.

[SFL96] S. Shekhar, A. Fetterer, D. Liu. Genesis: An Approach to Data Dissemination in Advanced Traveler Information Systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):40–47, 1996.

[SMWH] R. Sherlock, P. Mooney, A.Winstanley, J. Husdal. Shortest Path Computation: A Comparative Analysis.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. *Proceedings of the 13-th International Conf. on Data Engineering*, Birmingham, UK, 1997.

[SWCD98] P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Querying the Uncertain Position of Moving Objects. *Springer Verlag Lecture Notes in Computer Science*, number 1399, pp. 310-337, 1998.

[SAA00] I. Stanoi, D. Agrawal, A. Abbadi. Reverse Nearest neighbor Queries for Dynamic databases. *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pp. 44–53, 2000.

[ŠJLL00] S. Šaltenis, C. Jensen, S. Leutenegger, M. Lopez. Indexing the positions of continuously moving objects. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Volume 29 Issue 2, 2000.

[ŠJ02] S. Šaltenis, C. Jensen. Indexing of Moving Objects for Location-Based Services, In *Proceedings of the 18th International Conference on Data Engineering*, pp.463-472, 2002.

[SS93] A. Segev, A. Shoshani. A Temporal Data Model based on Time Sequences. *Temporal Databases: Theory, Design and Implementation*, pp. 248-270, 1993.

[SMM99] M. Sède, A. Moine, P. Marceau. Systemic Approach for Spatio-Temporal Database Structuration. *10th International Workshop on Database & Expert Systems Applications*, pp. 462, 1999.

[TCGJ93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass. *Temporal Databases Theory, Design, and Implementation*. The Benjamin Cummings Publishing Company Inc. 1993.

[TP02] Y. Tao, D. Papadias. Time-parameterized queries in spatio-temporal databases. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[TPZ02] Y. Tao, D. Papadias, J. Zhang. Aggregate Processing of Planar Points. *EDBT*, 2002.

[TPS02] Y. Tao, D. Papadias, Q. Shen. Continuous Nearest Neighbor Search. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[TPS03] Y. Tao, D. Papadias, J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *VLDB*, 2003.

[TMP03] Y. Tao, N. Mamoulis, D. Papadias. Validity Information Retrieval for Spatio-Temporal Queries: Theoretical Performance Bounds. *SSTD*, 2003.

[TUW97] J. Tayeb, Ö. Ulusoy, O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. *The Computer Journal*, 1997.

[TSPM98] Y. Theodoridis, T. Sellis, A. Papadopoulos, Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. 10th International Conf. Scientific and Statistical Database Management*, pp. 123-132, 1998.

[TWZC02] G. Trajcevski, O. Wolfson, F. Zhang, S. Chamberlain. The Geometry of Uncertainty in Moving Objects Databases. *Proceedings of the International Conference on Extending Database Technology (EDBT02)*, Prague, Czech Republic, March 2002.

[TWXN02] G. Trajcevski, O. Wolfson, B.o Xu, P. Nelson. Real Time Traffic Updates in Moving Objects Databases. *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, 2002.

[Trim] TRIMARC *www.trimarc.org*, The GCM Travel Web site *www.gcmtravel.com*.

[USCen] http://www.census.gov/acs/www/Products/Profiles/Single/2002/ACS/Tabular/380/38000US49921.htm

[VV99] P. Varman, R. Verma. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering*, pp.391-409, 1999.

[VW01-1] M. Vazirgiannis, O. Wolfson. A Spatiotemporal Query Language for Moving Objects. *Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, Los Angeles, CA, July 2001.

[VW01-2] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. *In Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pp20--35, 2001.

[Wash] *www.wsdot.wa.gov/traffic*, The Washington State Department of Transportation.

[WCDJ98] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. *Proceedings of the 14th International Conf. on Data Engineering*, Orlando, FL, 1998.

[WSXZ99] O. Wolfson, P. Sistla, B. Xu, J. Zhou, S. Chamberlain, Y. Yesha, N. Rishe. Tracking Moving Objects Using Database Technology in DOMINO. *Proceedings of The 4th NGITS Workshop*, Zikhron-Yaakov, Israel, pp. 112-119, July 1999.

[WJSC99] O. Wolfson, L. Jiang, P. Sistla, S. Chamberlain, N. Rishe, M. Deng. Databases for Tracking Mobile Units in Real Time. *Proceedings of the Seventh International Conference on Database Theory (ICDT)*, Jerusalem, Israel, pp. 169-186, Jan 1999.

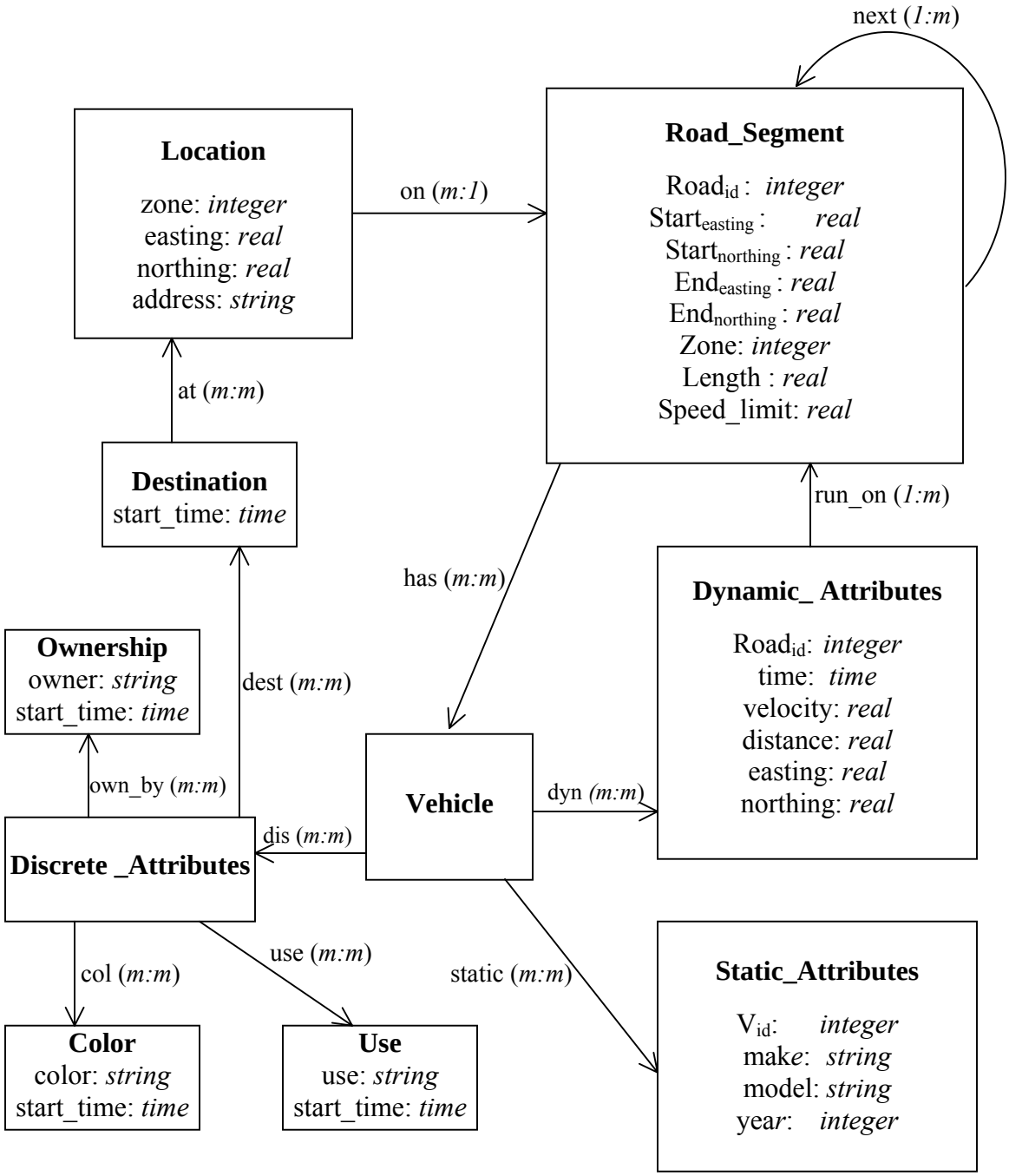
[WSCY99] O. Wolfson, P. Sistla, S. Chamberlain, Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Special Issue of the Distributed and Parallel Databases Journal on Mobile Data Management and Applications*, 7(3), 1999.

[WXC00] O. Wolfson, B. Xu, S. Chamberlain. Location Prediction and Queries for Tracking Moving Objects. *IEEE 16th International Conference on Data Engineering*, Sandiego CA, 2000.

[Wol02] O. Wolfson. Moving Objects Information Management: The Database Challenge. *The Proceedings of the 5th NGITS Workshop*, Caesarea, Israel, June 2002.

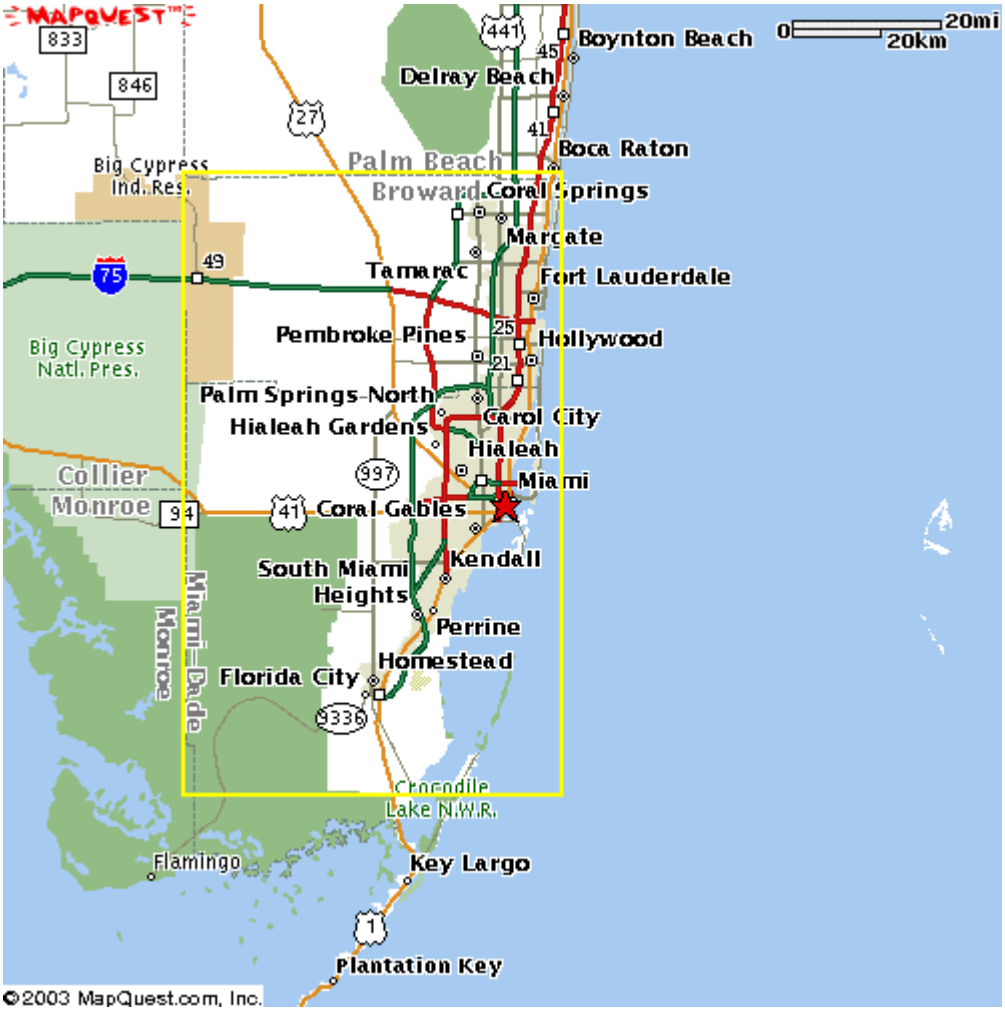
[Wor94] M. Worboys. A Unified Model for Spatial and Temporal Information. *The Computer Journal*, Vol 37, no. 1, pp25-34, 1994.

Appendix A Semantic Schema for Road Segments and Vehicles



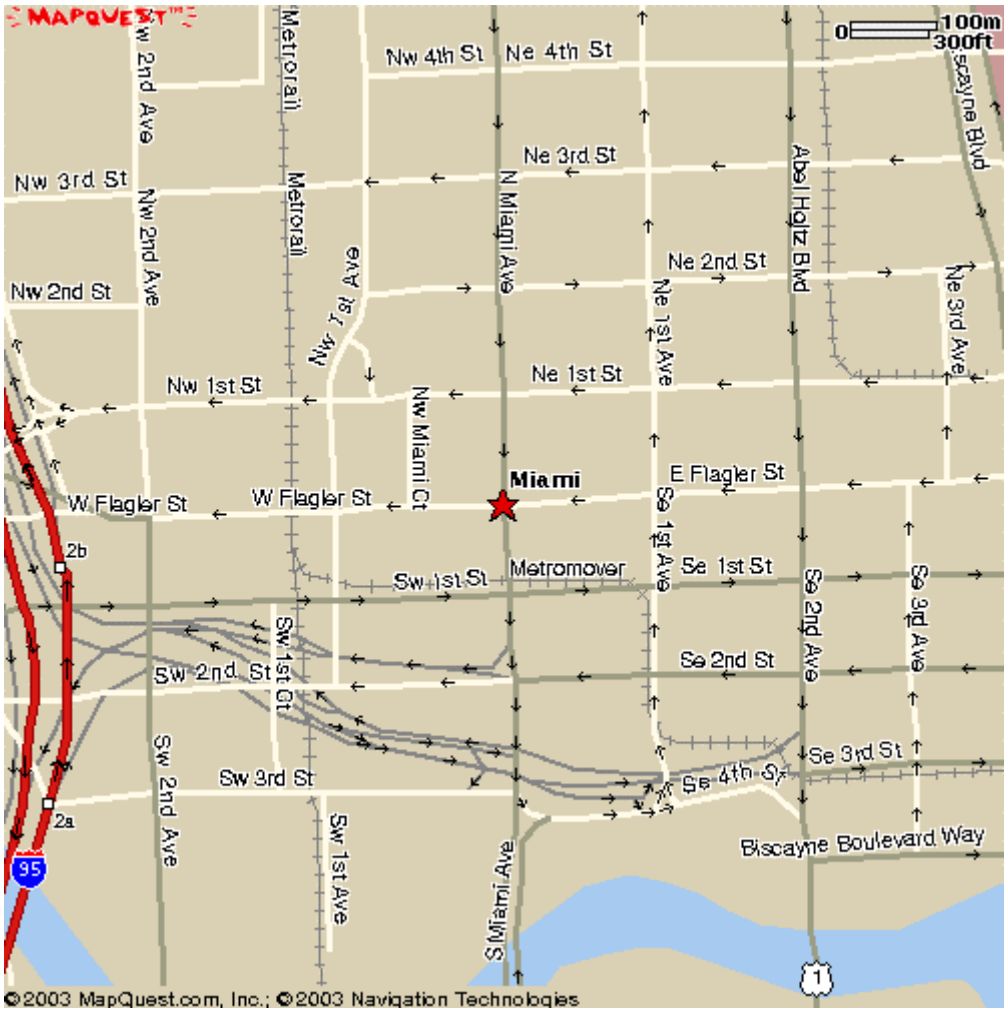
Appendix B Map of Greater Miami Area

Inside the Yellow Rectangle, Sized 95km×132km , is the Area Where Vehicles Would Most Likely Appear.

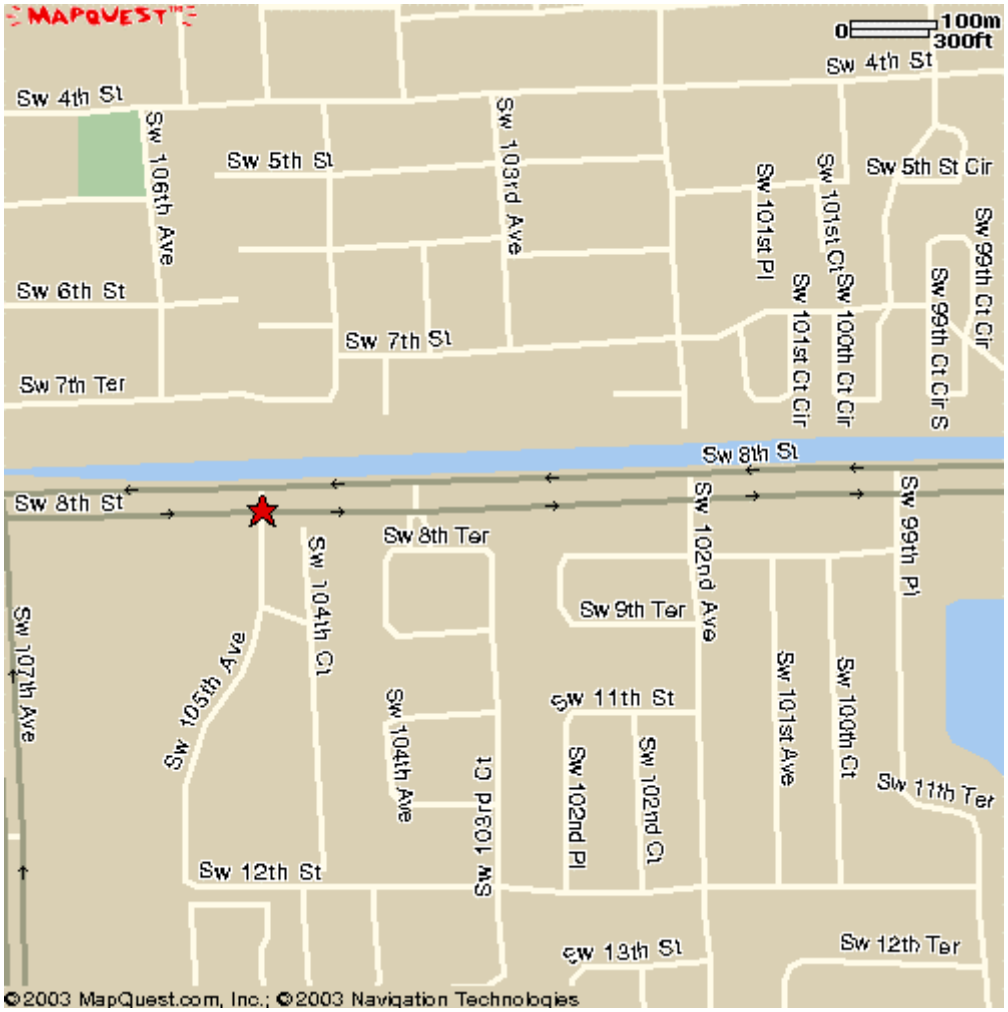


Appendix C Two Sample Road Segments Layout in Miami Florida

a. about 175 segments/km^2 -- Miami Down Town



b. about 210 segments/km^2 -- Miami SW 8th Street



VITA

XIANGYU YE

July, 1967	Born, Hubei, China
1988	B.E. in Computer Engineering Tsinghua University Beijing, China
1991	M.E. in Electronic Engineering China University of Mining and Technology Beijing, China
1991 – 1994	Software Engineer Software Engineering Research Lab, Chinese Academy of Sciences Beijing, China
1994 – 1997	System Engineer / Manager, System Integration Computer Network Information Center, Chinese Academy of Sciences Beijing, China
1997 – 2000	Research Associate High Performance Database Research Center, Florida International University Miami, Florida
2001	M.S. in Computer Science Florida International University Miami, Florida
2000 - 2003	Research/Teaching Assistant School of Computer Science, Florida International University Miami, Florida