University of New Orleans Theses and Dissertations   Dissertations and Theses

8-7-2008

Spatiotemporal Indexing With the M-Tree

John Finigan
University of New Orleans

Spatiotemporal Indexing
With the M-Tree

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements of the degree of

Master of Science
in
Computer Science

by

John Finigan

B.A. Fordham University, 2004

August 2008

Acknowledgement

Table of Contents

List of Figures

Abstract

Modern GIS applications for transportation and defense often require the ability to store the evolving positions of a large number of objects as they are observed in motion, and to support queries on this spatiotemporal data in real time. Because the M-Tree has been proven as an index for spatial network databases, we have selected it to be enhanced as a spatiotemporal index. We present modifications to the tree which allow trajectory reconstruction with fast insert performance and modifications which allow the tree to be built with awareness of the spatial locality of reference in spatiotemporal data.

Keywords:

spatiotemporal, database, index, trajectory, M-Tree, network

Chapter 1 – Background

1.1 - Introduction

Spatial databases are a specialization of the traditional database in which the database is optimized for the storage and querying of geometric data types. A spatial database is designed with awareness of distance between objects and relationships between objects, such as overlap. Spatiotemporal databases further specialize by including support for data representing the passage of time. A spatiotemporal database, therefore, can answer queries about both the location of objects and the change in objects' locations over time. Spatiotemporal databases are useful for tracking and analyzing the behavior of moving objects for such applications as defense and shipping.

The kinds of queries required of a moving object tracking database include historical queries, such as range queries (What objects were within 5 miles of point A in the last three days?), and trajectory queries (Where has object O been since we first observed it?). Such a system may be used for data mining of huge data sets, or for interactive querying. In either case, to be usable, the system must be able to answer queries quickly.

Due to the large volume of data which can be collected in a spatiotemporal database, efficient query execution depends in part on effective indexing. In this thesis we will explore the effectiveness of the M-tree [13] as a spatiotemporal index. The M-tree was created to index multimedia databases, but it is actually a general purpose index applicable to any data which can be classified by a metric function. Ioup et al. in [2] showed that the M-Tree could be an effective spatial index, using it to index spatial networks, in which the distances between objects are constrained by available routes, which may not be direct. Because spatial network databases are the natural representation of maps involving roads, they are an important class of spatial

databases. However, they are challenging to index, with Ioup et al. demonstrating one of the few successful methods.

Because the M-Tree has been shown to be useful for spatial networks, we propose enhancements to the M-tree which allow exploitation of spatial locality in spatiotemporal data and assist in reconstruction of object trajectories using the index. This thesis will describe the existing state of spatiotemporal and network indexing and detail these enhancements to the M-Tree.

1.2 – Spatial Database Indexes

Spatial database indexing has been a topic of Computer Science research since at least the 1980s, with the publication of Guttman's paper on the R-Tree [1]. The R-Tree is a balanced tree which is organized using the concept of *bounding boxes*, rectangular spatial boundaries which contain the spatial coordinates of the objects in the tree. A leaf node in the tree defines a bounding box which is large enough to contain all of the objects assigned to the leaf, and a non-leaf node's bounding box is defined as containing all of the bounding boxes of that node's children. Thus, the root node of the tree is associated with a bounding box that is large enough to contain every object in the tree.

Because the R-Tree is the foundational data structure for spatial indexing, we will examine it more closely. Guttman designed the R-Tree as a practical index for multidimensional data, citing the B-Tree as an analogous data structure which, however, assumes one dimensional ordering of the data in it. He cited existing data structures that could handle multidimensional data, such as methods based on grid partitioning and existing trees such as the Quad Tree, as being unsuitable, either due to static partitioning (which is decided before the data is known and may not lead to an even distribution of the data into the partitions) or due to a lack of

consideration of paged virtual memory. What Guttman required was a data structure which was built dynamically (in response to the inserted data), was balanced, was suitable for use in an environment where memory was operated on in pages, and could handle representations of arbitrary spatial shapes.   Interestingly, he was aware of GIS applications, but he actually had physical circuit layouts for microchips in mind; his test data in his original paper being the layouts of the Berkeley RISC II microprocessor.

The R-Tree is similar in organization to the B-Tree but is built for indexing n-dimensional numerical tuples, such as Cartesian coordinates. All data is stored in leaf nodes, and all leaves are on the same level. Each leaf node defines a *bounding box*, which is a rectangle (or rectangular prism, or hyperrectangle, depending on the dimensionality of the dataset) which is large enough to contain all of the points in the node. Each leaf contains between m and M entries, where $m \leq M/2$. In this paper, we use the term *fanout* synonymously with M.

Each non-leaf node has at least m children and defines a bounding box that covers its child subtree.   The root node, or course, may have less than m entries or children before the tree has two full levels. The root node has a bounding box that covers the whole tree.

The tree is searched recursively by testing whether the search points are within successive bounding boxes below the root, and in the best case, the algorithm must traverse only one route to find the desired leaf. However, the bounding boxes of separate nodes are allowed to overlap. Because of overlap, multiple paths sometimes need to be searched. Since the data inserted into any given node is not likely to naturally define a rectangle, the bounding box drawn around it is likely to contain some empty space. The combination of this empty space and overlap is called *dead space* and is detrimental to search performance.

Because dead space is undesirable, the default method of insertion into the tree is to insert the new entry into the subtree which will result in the smallest possible increase in bounding box sizes throughout the tree. This insertion criteria is common to most subsequent spatial indexes.

A special case of the spatial index is the spatial *network* index. Common spatial indexes such as the R-Tree are built around the implicit assumption that the straight-line (Euclidian) distance between objects is significant to the application. However, spatial networks are an important class of objects in GIS. An example of a spatial network is a network of roads. Road networks constrain the direction of travel of the objects on them; it is not possible to assume that the travel distance between two points on a road network is closely related to the Euclidian distance between them. Conventional spatial indexes are of little use in spatial network databases. One of the few indexes which has had success is the M-Tree, as implemented by Ioup et al. [2]. The work of Ioup et al will be extensively explored in Chapter 2.

1.3 – Spatiotemporal Database Indexes

The R-Tree and its derivatives, such as the R+ tree and the R* tree are generally successful as indexes for two dimensional spatial data, which has led researchers to consider them for indexing spatiotemporal data. The simplest method which has been suggested is to consider spatiotemporal data as generalized three dimensional data and to build a conventional R-Tree to index this data: this kind of data structure is a 3D R-Tree [4]. In this method, the bounding boxes of the classical R-Tree become bounding rectangular prisms. This method is workable, but sometimes inefficient, since excessive dead space becomes hard to avoid. The R-Tree may be used on data of even higher dimensionality, but with each added dimension, the aforementioned problem becomes worse. This is known in the literature as *The Curse of Dimensionality* [3].

Givaudan [5] developed a taxonomy of spatiotemporal trees, classifying them by representation of time, update semantics, insert semantics, supported query types, storage efficiency, and trajectory preservation. She described three variants of the 3D R-Tree:

1: Each leaf may be dedicated to the different versions (locations) of a particular object.

2: The trajectories of each object may be represented as polylines which are stored in the R-Tree.

3: Each object may be stored in a standard R-Tree with time as an additional dimension.

Other trees classified by Givaudan include those that store data in multiple trees, such as the 2+3 R-Tree [6], Givaudan's 2+3 TR-Tree [5], the HR-Tree [7], and the MV3R-Tree [8]. The 2+3 R-Tree uses an auxiliary tree of current positions to store temporary data about trajectories which are not known to be complete; once they are completed, they are inserted as polylines into the main three dimensional historical data store. The 2+3 TR-Tree does not index polylines of trajectories like the 2+3 R-Tree does. Instead, it separately indexes position points and lines, with lines representing the passage of time in the same position. The dead space associated with trajectory polylines is thus avoided. The MV3R-Tree and the HR-Tree are multiversioning data structures where multiple roots which are valid for different times share common data blocks to reduce wasted space.

Givaudan also categorizes some trees which are not strongly spatial, instead concentrating on efficient representations of trajectories without the spatial organization inherent in most R-Tree derivatives. The STR-Tree [9] and the TB-Tree [9] both insert new trajectory segments as close together as is possible, with the TB-Tree making the further guarantee that each leaf is devoted to only one trajectory. The STR-Tree attempts to compromise spatial locality and trajectory preservation, while the TB-Tree places little importance on spatial locality and will store trajectory segments which are spatially close in different nodes. The TB-Tree does

allow one trajectory to span multiple nodes, and so implements a doubly linked list between nodes to facilitate trajectory reconstruction.

Givaudan's categorization scheme for spatiotemporal trees will be applied to our work in Chapter 3.

1.4 – Trajectory Reconstruction

Trajectory reconstruction is defined in this paper as the ability to retrieve all historical locations of an object once the object itself has been located. In some data structures, trajectory reconstruction also includes interpolation, meaning that the position of an object is known at any particular timestamp using estimation based on known trajectories when actual data is not available.

More formally, given a moving object O, the position of the object at a given time t can be denoted as O_t. If the first known position of an object is at time t1, and the last known position is at tn, the trajectory of the object is the ordered set $\{ O_{t1}, O_{t2}, \dots , O_{tn-1}, O_{tn} \}$.

Trajectory reconstruction can be at odds with efficient spatial indexing. Some trees represent trajectories as actual spatial objects (polylines) and index these. This generally results in too much dead space, and thus poor performance. Other trees attempt to optimize the paging of trajectory segments by dedicating nodes to single objects' trajectories. This makes trajectory retrieval fast but compromises spatial query performance, since spatial locality of reference is compromised.

A further question in reconstruction is, if a trajectory spans multiple nodes, how are these nodes found? A slow and naïve method is to traverse the tree to find the next node in the trajectory. More sophisticated trees like the TB-Tree allow the trajectory to be followed through arbitrary nodes using a linked list.

1.5 – Spatiotemporal Queries

One spatiotemporal query type, trajectory reconstruction, has already been defined. Another type is a variant of the range query, in which the user specifies a location, a spatial range, and a range of times and asks what objects have been near to the specified location with respect to these parameters. This is called a spatiotemporal range query.

Various schemes for expressing spatial queries in SQL have been suggested, but commercially implemented systems such as PostGIS [10] have settled on a simple, function based model. In a PostGIS-like system, a spatial range query which selects all objects within 5000 units of a specified point can be expressed as:

SELECT object_id

FROM map_table

WHERE Distance(object_position, POINT(123, 456)) < 5000

Formally, this query returns:

$$R = \{\forall O \mid d(O, p) < 5000\}$$

Where:

- R is the set of results

- O is an object in the database

- p is the given point

- d is the distance function

From this query sample, and considering that standard SQL supports manipulating temporal data types, it is easy to see that a spatiotemporal range query which restricts the above results to only those objects seen since the first day of 2008 could be as simple as this:

SELECT object_id

FROM map_table

WHERE Distance(object_position, POINT(123, 456)) < 5000

AND object_timestamp > '2008-01-01 00:00:00'

Formally, this query returns:

$$R = \{\forall O \mid (d_s(O, p) < 5000) \wedge (d_t(O, t) > 0)\}$$

Where:

- R is the set of results

- O is an object in the database

- p is the given point

- t is the given time

- d_s is the spatial distance function

- d_t is the time distance function

Similarly, we can write a query in standard SQL to reconstruct the trajectory of a particular object since the beginning of 2008:

SELECT object_position, object_timestamp

FROM map_table

WHERE object_id = 6502

AND object_timestamp > '2008-01-01 00:00:00'

ORDER by object_timestamp

Formally, this query returns:

$$R = \{\forall (o_p, o_t) \mid d_t(o_t, t) > 0\}$$

Where:

- R is the set of results

- o is the specified object

- (o_p, o_t) is a tuple encapsulating an object's position and timestamp

- t is the specified time

- d_t is the time distance function

- R is returned in sorted order by ascending o_t

Implemented trajectory-capable spatiotemporal systems are less common than spatial systems like PostGIS, so it is reasonable to look at some theoretical suggestions of what spatiotemporal systems would look like. Some authors suggest new SQL keywords, such as *before* as a replacement for inequality operators (>, <, etc.) in temporal queries, which may make the user more comfortable but do not fundamentally increase the expressive power of SQL. Other systems actually introduce new operators. One such system was designed by Viquera and Lorentzos [11]. They implement two new operators, FOLD and UNFOLD, and define two categories of time, the *instant* and the *period*. An instant is a distinct timestamp, such as '2008-01-01 00:00:00'. A period is defined as an interval in time and is defined by a pair of timestamps, for example, ['2008-01-01 00:00:00' , '2008-01-01 12:00:00']. If time is viewed as a dimension, an instant is analogous to a point, and an interval is analogous to a line.

More interesting are their new operators. Given a set of instants which overdefine a line, FOLD outputs a period which contains these instants and is bounded by the least and most recent instants of the set. UNFOLD reverses FOLD, expanding a given period to its component instants. More simply put, FOLD removes redundant points from a dataset which defines a line or polyline, outputting the line or polyline as defined by the minimal necessary set of points. UNFOLD accepts a line or polyline and outputs the set of points needed to define it. Using these

operators, trajectories can be reduced to non-overdefined polylines which are easily interpreted for analysis or visualization.

1.6 – Design Goals

In designing a storage method for a database system, there are, in general, two goals to be pursued. The first is to enable certain types of queries to be answered, and the second is to increase the speed of query processing.

An obvious example of the first is a spatial index like the R-Tree. This tree allows natural representation of spatial relationships and allows these relationships to be queried, since the organization of the tree depends on the spatial relationships of the objects in it.

The second goal is to make these queries fast. This depends on two factors: the amount of calculation required (CPU cost), and the amount of data which must be sifted through (IO cost). Because processors can generally do arithmetic faster than they can fetch data from memory, and because disk is even slower than memory, IO cost generally dominates. Because memory is generally much smaller than disk and is volatile, it is usually used as a cache for data on disk. Query speed, then, depends in great part on algorithms which use memory as an efficient cache.

A theoretically optimal caching algorithm would always choose a page for eviction which is going to be needed farthest in the future compared to all of the other pages in the cache. This algorithm, known as *Belady's Algorithm* [12] or the *Clairvoyant Algorithm*, requires accurate prediction of the future and is thus impossible in practice. Practical algorithms evict pages which were accessed longer ago than or less frequently than the other pages in the cache. Thus, practical caching algorithms perform best on accesses that reference the same set of pages most of the time.

It thus makes sense to design an access method which references as few pages as is possible to answer a given query. A real-world cache design will be forced to evict fewer pages this way, because fewer new pages will be brought into the cache. It is also desired that a typical working set of the access method has good locality of reference, meaning (in this case) that the data objects on a particular page are somehow related and likely to be involved in the same query.   This is helpful since any given query should access fewer pages if each page contains more useful data. A worst case scenario would be if data objects were mapped to pages in a random fashion, since this would on average cause one page access per data object searched for.

 Our design goals are in line with the considerations just mentioned. We wish to design a tree which indexes spatiotemporal data with good spatial locality of reference to facilitate efficient cache usage. We also wish to design a lightweight trajectory reconstruction scheme which does not interfere with this locality of reference, so that trajectory queries may be answered efficiently.

Because the M-Tree has been shown to be useful for indexing spatial network databases, we will adapt the M-Tree for spatiotemporal indexing and trajectory reconstruction. The purpose of this thesis is thus to build an M-Tree based spatiotemporal index that will facilitate future integration of a system which can handle spatiotemporal data, trajectory reconstruction, and spatial network databases.

1.7 – Organization of this Thesis

In Chapter 2, we will introduce the M-Tree, metric spaces, and the spatial networks. We will then introduce the spatial network index work of Ioup et al.

In Chapter 3, we will discuss our enhancements to the M-Tree for spatiotemporal indexing and lightweight trajectory reconstruction. We will classify the resulting data structure

in Givaudan's taxonomy, and provide exposition and pseudocode on the actual construction of the tree.

In Chapter 4, we will discuss the testing environment, including the design of the M-Tree database index simulator. We will then present results for range queries and trajectory reconstruction against the simulator.

In Chapter 5, we will present conclusions and future work.

Chapter 2 - The M-Tree

2.1 – The M-Tree

The M-Tree is a dynamically updatable, balanced tree data structure which is organized according to a distance function applied to the entries in the nodes. The distance function is used to place each entry into a node containing other entries which are close to it. Thus, the new entry is located in the tree according to its relationships with all of the other nodes in the tree, and not according to its relationship with one or more fixed reference points. The M-Tree was proposed in a 1997 paper by Ciaccia, Patella, and Zezula [13].

All of the data in an M-Tree is stored in leaf nodes. All non-leaf nodes are known as routing nodes, and the entries in these nodes are routing entries. Each routing entry has an associated covering radius which is as large as the subtree that the routing entry is the parent of.

2.2 - Metrics

The M in M-Tree stands for *metric*, meaning that the distance function and data set exist in a metric space, in the topological sense. In general, this means that the distance function is defined such that distance between objects satisfies the conditions of symmetry, non-negativity, and the triangle inequality, as defined below [2], with p, q, and r being points in R^2, the two-dimensional real plane, and d() being a distance function:

1: Symmetry:
$\forall (p,q) \in R^2, d(p,q) = d(q,p)$
2: Non-negativity:
$\forall (p,q) \in R^2, p \neq q, d(p,q) \geq 0$
3: Triangle Inequality:
$\forall (p,q,r) \in R^2, d(p,q) \leq d(p,r) + d(q,r)$

However, for the purposes of spatial databases, it is not necessary that the above conditions strictly hold. In particular, the M-Tree has been successfully used to index data in which

distances were not strictly symmetrical [2]. Distance functions such as Euclidian distance on n-dimensional points:

- distance(p, q) = $\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$

are metric functions. However, functions which are, for instance, periodic do not allow for the triangle inequality and so are not useful for building the M-tree.

Euclidian distance is the most common member of an important metric family known as

L_p, defined as $\left(\sum_{i=1}^{n} |q_i - r_i|^p \right)^{\frac{1}{p}}$, where q and r are n-dimensional points. Other L_p metrics may be used with the M-Tree, as will be seen in Section 2.4.

If an M-Tree is built with Euclidian distance as the metric, the resulting tree is similar in concept an R-Tree of the same dimensionality with the exception that the nodes define bounding circles or spheres as opposed to the rectangles or rectangular solids of the R-Tree.

2.3 – Split Functions

As with any spatial tree, the function which redistributes entries when nodes split is an important factor in the overall shape of the tree. A good split function should minimize the overlap of the resulting nodes while minimizing the amount of dead space in the tree. Several split algorithms were suggested by Ciaccia et al, including random selection, direct calculation and selection of two entries providing minimal covering radius, and what Ciaccia et al. called the maximal lower bound method. Ciaccia further suggested the use of a *confirmed* split, where one of the entries chosen for promotion is the same as the node's parent entry. Maximal lower bound reduces to selection of the node's parent entry and the entry farthest from it when the *confirmed* variant is used.

Our M-tree tests employ the confirmed maximal lower bound split. It was decided empirically that random promotion too often resulted in poor performance, while the small improvement over maximal lower bound provided by minimal covering radius calculation did not justify its use, as a slow $O(n^2)$ algorithm.

2.4 – The M-Tree as an Index for Spatial Networks

The M-Tree was initially designed by Ciaccia et al. as an index for multimedia databases. The tree was designed to be applied to arbitrary data objects on which multidimensional feature vectors could be computed. These feature vectors would be the input to the tree's metric function, and the tree could thus accelerate similarity queries on the database. The M-Tree was applied to spatial databases by Ioup et al, who successfully used it to accelerate GIS queries on road networks.   Distance computation in road networks is more difficult than computation of Euclidian distance between points on a map because motion in a road network is constrained to pre-defined paths, which may define the possible paths between two points in a way that is significantly different than the Euclidian distance between them. As an example, consider Figure 1 (generated by MapQuest) below:

Figure 1: A real-world spatial network
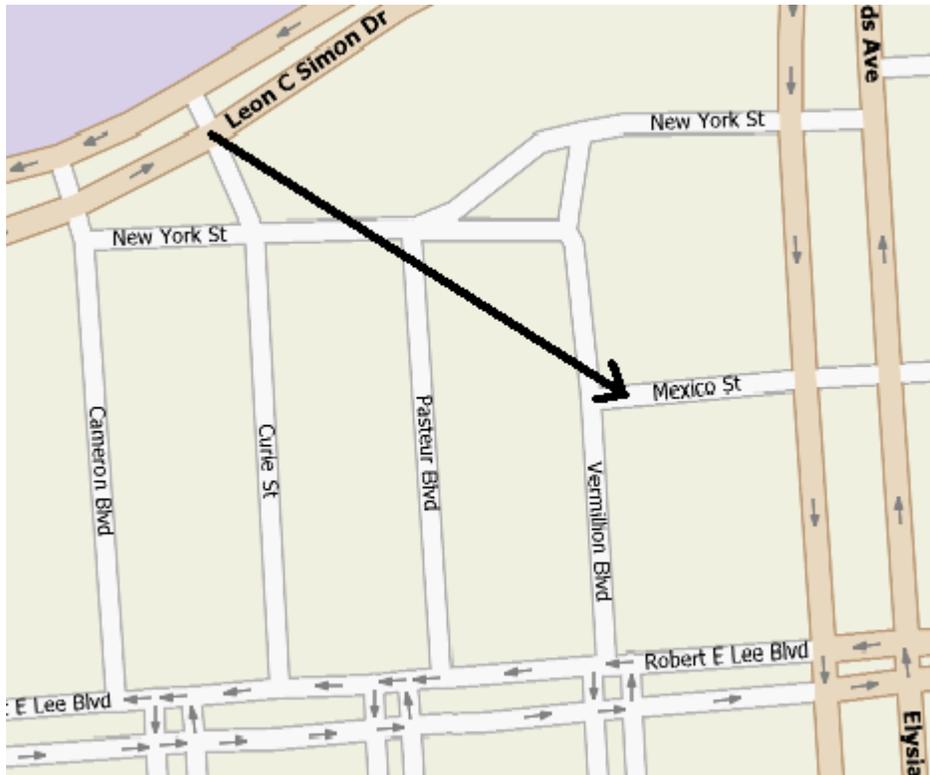
If someone driving in a car wished to reach Mexico St. from the corner of Curie St and Leon C

Simon Dr, it would be necessary to travel in a zig-zag path including New York St. and

Vermillion Blvd. to get there. The arrow in Figure 1 denotes a straight-line path from Leon C.

Simon Dr. to Mexico St, representing a Euclidian measure of the distance between the two points

of interest. It is easy to see that the network (road) distance between the two points is larger than

the Euclidian distance. In this case, the network distance between the points of interest is about

30% larger than the Euclidian distance between them. Thus, the Euclidian distance between the

points is a poor estimate of the network distance between them. The network in Figure 1 is

relatively well connected. In a rural area with few roads, Euclidian distance could provide significantly worse distance estimates.

Ioup et al. applied the M-Tree to the problem using an *embedding*, which maps the n-dimensional spatial points defining the road network into a k-dimensional metric space, k > n. The specific embedding used by Ioup et al is called Truncated Road Network Embedding, which was originally described by Shahabi et al. in [15], and which is defined by computing the metric distance between each given node and designated subsets of pre-selected nodes in the data set. The metric function used for distance between nodes is chessboard distance on k-dimensional points:

- distance(p,q) = max($|p_i - q_i|$), $\forall i \in k$

Chessboard distance is from the same family of metrics as conventional Euclidian distance, L_p, and was selected based on the work of Shahabi et al. Within this family it is known as L_∞ distance because it represents the limit of the L_p formula as p goes to infinity.

Without a method of estimation, network distance calculation done "on the fly" is limited in performance by the complexity of Dijkstra's algorithm, which according to Ioup et al, is O(n log n). Other algorithms, such as A*, may do better in run time, but are still O(n log n). Total precomputation of inter-node distances in a network can lead to lower search complexity, but with larger networks, precomputation becomes intractable in time and space.

Embedding with Truncated RNE provides a method of estimation which allows complexity performance better than O(n log n). Transforming the input data set into the embedded one which is actually inserted into the tree does require some precomputation, but it is limited in complexity to O(n $\log^2$ n) as opposed to total precomputation's complexity of O(n^3 log n), and is thus tractable for real world datasets such as the map of a US state.

Thus, the distances between the embedded points which are inserted into (and define) the M-Tree are estimates of the network distances between the original points processed using Truncated RNE. Distance queries on these embedded points reflect, with a high degree of accuracy, network distance queries on the unembedded data set.

Ioup et al showed that the M-Tree was useful not just for multimedia databases but also for spatial databases, and that it was particularly useful when used with a spatial embedding, since it was not designed specifically for use with Euclidian distance on low dimensionality points, as was the R-Tree. With a spatial embedding, the M-tree enables fast queries on network distance, which is an unusual capability. Thus, we decided to explore the M-Tree as a spatiotemporal database access method, with the value of an index which can handle network distance and spatiotemporal data in mind.

Chapter 3 – The Enhanced Spatiotemporal M-Tree

Our enhancements to the M-Tree were designed with two goals in mind. The first is to allow trajectory reconstruction queries, and the second is to take advantage of spatial locality of reference in spatiotemporal data. To these ends, two enhancements were made to the standard M-Tree design.

3.1 – Trajectory Awareness with Fast Insert

The first enhancement, for trajectory reconstruction, is the addition of a helper datastructure, which we call the *last list*. The last list is a key-value table, the keys of which are the object identifiers of each object in the tree. The values, which are updated as new data is inserted, are the entries representing the last known location of each object in the tree. When a new object is inserted into the tree, there are two possibilities: either the object has never been seen before, or it represents an new location for a known object. In the first case, the object is inserted, it is recorded in the last list, and no further action is taken. In the second case, the last list is first consulted to find the last location at which this object was recorded. This logical location becomes part of the metadata of the current entry to be inserted, and the last list is then updated with the location of the current entry.

Using the metadata recorded onto each entry using the last list, the various entries composing the location of a particular object form a linked list. At any particular entry in the tree, finding the previous locations of that object merely requires traversing the list. If there are known locations both before and after the time of the entry in question, finding the future locations means consulting the last list for the most recent location of the object, and then traversing backwards from this entry.

This scheme is inspired by the doubly linked list used in the TB-Tree [9], but it is more lightweight. The advantage of the last list over more complicated schemes such as a doubly-linked list is in insertion code simplicity and metadata sparseness. The last list allows trajectory reconstruction with minimal metadata overhead, and the amount of bookkeeping needed to do an insert is similarly minimal. A doubly-linked list would mean that an insert would require enhancement of the metadata associated with previously inserted objects, which would themselves have to be located before the insert was completed. Our singly linked list allows insertion without reading or writing to the object's previous entries.

Other indexes, such as the 2+3 TR-Tree, have stored trajectories that are not known to be complete (that is, where the position of the object is still evolving) in a temporary tree. Once the trajectory is known to be complete, it is moved into the main tree. The advantage of our method over this one is simplicity and the need to only search one tree instead of two when queries are processed.

The minor disadvantage of a singly linked list is that it may only be traversed in one direction, in this case, towards the past. We do not consider this disadvantage to be serious because the last list allows the most recent entry in the list to be found with ease, and once it is found, any part of the trajectory may be examined.

Also, since the linked list links entry to entry, we must have a translation table which maps entries to nodes, their logical locations in the tree. This represents some amount of overhead on inserts, in both space and time.

3.2 – Taking Advantage of Spatial Locality

The second enhancement has the goal of speeding up queries through better buffer utilization. Recall from Chapter 1 that practical caching algorithms perform best when the

algorithm accessing them has a small working set, small being defined here as smaller than the size of the cache. Also recall that the working set can be made smaller if the data on the pages has good locality of reference, since this causes fewer pages to be referenced for a given number of objects needing to be fetched. Since we represent a spatiotemporal location as a three dimensional tuple (x, y, t), it was realized that when the M-Tree is built using Euclidian distance, the natural spatial locality of close together (and likely to be related) objects is distorted by the inclusion of the third dimension, t. Of course, the same idea holds if temporal locality is considered; it is distorted by the x and y dimensions.

Thus, it was decided to try splitting (and thus paging) the tree according to a two dimensional spatial metric rather than a three dimensional metric. Recalling from the previous chapter that we are using a confirmed maximal lower bound split algorithm, we have enhanced that algorithm to take into account only the x and y dimensions of object locations when making split decisions.

Since the tree is split according to a purely spatial method, we can calculate both the normal M-tree covering radius of a routing entry, and also the two dimensional covering radius. Thus, when answering spatiotemporal queries, we should be able to prune whole subtrees that we would not otherwise be able to prune. This is because, in a normal three dimensional tree, the entanglement of the spatial and temporal dimensions makes it difficult to rule out whether a particular covering radius could contain a particular spatial location. With the secondary two dimensional covering radius, we can rule out certain subtrees as containers for our queried location, and prune them without further consideration.

Pruning unnecessary subtrees has the effect of making the query algorithm's working set smaller and reducing cache evictions, since fewer total nodes have to be examined to complete a

query. It should be most effective when there is significant overlap of covering radii, since this is when many different paths have to be traversed in the tree to find a particular entry.

3.3 – Classification under Givaudan's Taxonomy

As mentioned in Section 1.1, Givaudan [5] developed a taxonomy of spatiotemporal trees, classifying them on 8 criteria. These are as follows:

1 – Time Dimension: Valid time or transaction time. Valid time databases can record any timestamp for any object, and timestamps can be deleted or modified. Transaction time databases only record the time at which the object becomes part of the database, and this timestamp is immutable.

2 – Time Evolution: Discrete or continuous. Discrete time databases store each discrete timestamp at which an object is observed. Continuous time databases store trajectory segments (polylines), allowing the database to answer queries by interpolation where exact data is unavailable.

3 – Data Evolution: Moving, growing, or fully evolving. Moving means objects may change position, but the cardinality of the dataset cannot grow, i.e., objects may be not be added. Growing means that objects may be added to the database, but once added, they may not move. Fully evolving means both moving objects and growing the cardinality of the database are allowed.

4 – Data Acquisition: Static, chronological, or dynamic. Static databases may only be built from datasets which are known ahead of time. Chronological means that inserts are allowed at any time, but newly inserted objects must have higher timestamps than previously inserted objects. Dynamic means that objects may be inserted at any time with any timestamp.

5 – Query Type Support: Historical, trajectory, or combined. Historical queries find all objects meeting a spatial or temporal criteria. Trajectory queries retrieve all of the historical positions of one object. Combined databases support both types of queries.

6 – Discrimination: Spatial, temporal, or spatiotemporal. Spatial databases index objects by spatial position, temporal by timestamp. Spatiotemporal databases index and retrieve objects based both on position and on timestamp.

7 – Insertion / Split Strategy: Least enlargement, minimum overlap, linear ordering, or trajectory preservation. With least enlargement, a new entry is inserted into the node which will result in the least enlargement of the bounding boxes or circles in the tree. Minimal overlap inserts entries such that overlap between bounding shapes of nodes is minimized. Linear ordering places entries according to a deterministic ordering such as a Hilbert curve or Z-order. Trajectory preservation groups entries which belong to the same trajectory, regardless of spatial locality.

8 – Version Duplication: Duplication or no duplication. Databases which force version duplication will store multiple entries for the same object if it is observed at the same place but at different times. Databases with no duplication store one entry per object position, regardless of timestamp.

The spatiotemporal M-Tree is classified in Givaudan's taxonomy as follows:

	Spatiotemporal M-Tree
Time Dimension	Valid Time
Time Evolution	Discrete
Data Evolution	Fully Evolving
Data Acquisition	Dynamic
Query Type	Combined
Discrimination	Spatiotemporal
Insert / Split Strategy	Least Enlargement
Version Duplication	Duplication

The rationale for these classifications is as follows: Time and space in the spatiotemporal are

both handled in the same way in the spatiotemporal M-Tree, although time is not taken to

account in all distance calculations. Time dimension, data evolution, and data acquisition all

reflect the fact that the M-Tree is a general-purpose database access method: with the correct

algorithms, updates, deletion, and insertion are as straightforward as in a B-Tree. Time evolution

is discrete since no facility for interpolation has been added, although there is no reason to

believe this is not possible. Query type is combined since the tree supports both historical

queries and trajectory reconstruction. Discrimination is spatiotemporal, since the tree is

organized on both space and time dimensions. A least enlargement algorithm is used for

insertion, and version duplication does happen.

3.4 – Implementation Pseudocode

The implementation of the enhanced tree is presented below in Python-like pseudocode, which is

a natural choice for an implementation written in Python. The reader should note that control

flow in Python is determined by indentation level; that is, a "block" of code is one which is on

the same indentation level, and which would be enclosed in braces in a C-family language.

Comments in the pseudocode are preceded by a #.

First we will examine the recursive insert code, which is based on the abstract algorithm

presented in [13].

```
def insert(newEntry, node):
  if node is a routingNode:
    for each entry in node:
      dist = distanceFunction(entry, newEntry)
      if dist <= entry.radius:
        # inRadius and outRadius are associative arrays
        inRadius[entry] = dist
      else:
        outRadius[entry] = dist - current covering radius of entry

    if inRadius contains entries:
      # we can insert the entry without enlarging the node's covering radius
      minDist = smallest dist in inRadius
      minEntry = entry associated with minDist
    else:
      # newEntry is outside the current covering radius of its new node;
      # we enlarge the radius as little as possible
      minDist = smallest dist in outRadius
      minEntry = entry associated with minDist
      3d covering radius of minEntry = 3d_euclidian_distance(minEntry, newEntry)
      2d covering radius of minEntry = 2d_ euclidian_distance(minEntry, newEntry)

    insert (newEntry, child node of minEntry)

  else: #node is a leaf
    if node is not full:
      newEntry.distance = distanceFunction(parent entry of node, newEntry)
      add newEntry to node
    else:
      split(node, newEntry)
```

This code may be understood as follows: When the function is initially called, the argument *node* is the root. The overall goal is to traverse the tree to find a leaf to insert the new entry into. We try to choose a routing entry in the node which will not have its covering radius enlarged by the new entry. If this is not possible, we choose the routing node whose covering radius will be enlarged the least. If we did have to enlarge the covering radius of the routing entry chosen, we calculate the new radius, and then recursively call insert with *node* set to the child of our chosen routing entry. Once we reach a leaf, we must decide if the entry can be inserted into it, or if it is full. If it is full, we create a new leaf by calling split, defined below.

```
def split(node, newEntry):
    newEntry.distance = distanceFunction(parent entry of node, newEntry)
    add newEntry to node

    create a new empty node: newNode

    promotedEntries = node.promote()
    #promoteEntries yields two entries in the form of an array
    promotedEntries[0].childId = node.nodeId
    promotedEntries[1].childId = newNode.nodeId

    oldNodeParentEntryId = node.parentEntryId

    node.parentEntryId = promotedEntries[0].entryId
    newNode.parentEntryId = promotedEntries[1].entryId

    # partition
    # arbitrarily, promotedEntry[0] belongs to the original node, and [1] belongs new node
    for each entry in node:
        dist0 = 2d_ euclidian _distance(promotedEntries[0], entry)
        dist1 = 2d_ euclidian _distance(promotedEntries[1], entry)
        if (dist0 >= dist1):
            entry.distance = dist1
            if entry has child:
                child.parentNodeId = newNode
            move entry to newNode
        else:
            entry.distance = dist0
```

```
#compute covering radii of new parent entries
promotedEntries[0].3dradius = compute3dRadius(promotedEntries[0], node)
promotedEntries[1].3dradius = compute3dRadius(promotedEntries[1], newNode)
promotedEntries[0].2dradius = compute2dRadius(promotedEntries[0], node)
promotedEntries[1].2dradius = compute2dRadius(promotedEntries[1], newNode)

# put promoted entries in new parent node
if node is Root:
    create new empty root node: newRoot
    add promotedEntries[0] to newRoot
    add promotedEntries[1] to newRoot
    node.parentNodeId = newRoot
    newNode.parentNodeId = newRoot

else: # new parent node is not root and so has a parent
    delete entry oldNodeParentEntryId
    promotedEntries[0].distance = distanceFunction(parent of new parent node,
promotedEntries[0])
    add promotedEntries[0] to new parent node
    set new parent node as parent of newNode

    if parent of node is full: #line 12
        split(parent of node, promotedEntries[1])
    else:
        promotedEntries[1].distance = MTree.distanceFunction(parent of node,
promotedEntries[1])
        add promotedEntries[1] to parent of node
```

Its length notwithstanding, the split function is simple. We select two entries from the splitting

node's parent node to be the parents of the product of the split. Using our confirmed maximal

lower bound promotion algorithm, one of these parent entries is the existing parent entry. We

then simply divide the splitting node's entries between that node and the new node, based on

which parent entry they are closest to, using Euclidian distance in the x and y dimensions. Then

we insert the new parent entry into the parent node. If this requires a split because the parent is

full, we recursively split all the way up to the root if needed.

The tree shown in Figure 2 below was built with the insertion and split code so far detailed:

Figure 2: Example M-Tree
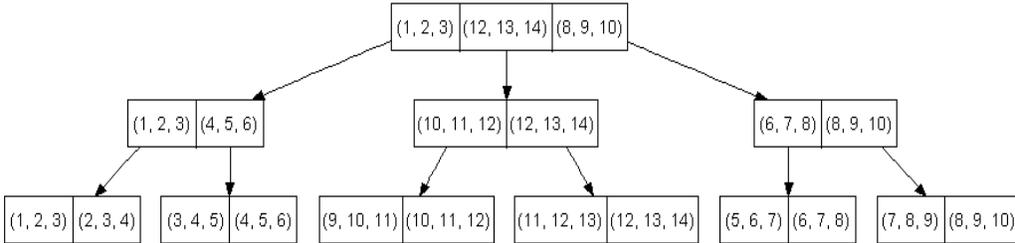
Three things can be noticed from the sample tree: Firstly, the split is indeed the confirmed split mentioned in Section 2.4, that is, when two entries are chosen to be promoted for a split, one of the chosen entries will be the same as its parent. Secondly, the organization of the tree is based, in this case, on the Euclidian distance between the x and y components of the points comprising the tree. The leaf containing (4,5,6), for example, is a child of the subtree belonging to (1,2,3) in the root. The distance between the x and y components of these two points is $\sqrt{18}$, while the distance between it and another root entry, (8, 9, 10) is $\sqrt{32}$. Thus, when a partitioning choice was made during a split, (4, 5, 6) and (1, 2, 3) were placed together. Thirdly, each non-leaf node defines a radius which contains all of its leaf children. Taking for example the leftmost node on the middle level, it is clear that a circle containing (1, 2, 3) and (4, 5, 6) also contains (2, 3, 4) and (3, 4, 5), and it is similarly clear that (1, 2, 3) and (12, 13, 14) in the root define a circle that covers all points in the tree.

Next we will look at the range query code:

```
def rangeQuery(node, queryPoint, searchRadius, results):
    parentNode = parent node of node
    parentEntry = parent entry of node

    if 2d_l2_distance(queryPoint, 2d_node_center) > (node.nodeRadius + searchRadius):
        return nothing #cull this branch

    if node is routing node:
        if node is root:
            d_Op_Q = 0.0
        else:
            d_Op_Q = distanceFunction(parentEntry, queryEntry)

        for entry in node:
            if node is root:
                d_Or_Op = 0.0
            else:
                d_Or_Op = entry.distance
            rQ_rOr = searchRadius + entry.radius


            if abs(d_Op_Q - d_Or_Op) <= rQ_rOr:
                d_Or_Q = distanceFunction(entry, queryEntry)
                if d_Or_Q <= rQ_rOr:
                    rangeQuery(child of entry, queryEntry, searchRadius, results)

    else: # nodeIndex points to a leaf node
        d_Op_Q = distanceFunction(treeNodes[parentEntry, queryEntry)
        for entry in node:
            d_Oj_Op = entry.distance
            if abs(d_Op_Q - d_Oj_Op) <= searchRadius:
                d_Oj_Q = distanceFunction(entry, queryEntry)

                if d_Oj_Q <= searchRadius:
                    add entry to results

    return results
```

To understand this query code, we should first define some variables:

● d_Op_Q is the distance between the parent entry of the node given as an argument to the

function and Q, the point at the center of the search radius.

- d_Or_Op is the distance between the given node's parent and the selected routing entry in the node.

- d_Or_Q is the distance between the selected routing entry and the query point, Q.

- d_Oj_Op is the distance between a selected leaf entry and the containing leaf node's parent.

- d_Oj_Q is the distance between a selected leaf entry and the query point, Q.

- the operator r( ) denotes the covering or search radius of an entry

With these variables defined, the above range query code is a straightforward implementation of Ciaccia et al's range query algorithm in [13], with the exception of the bolded statement, which will be explained last. Ciaccia et al's algorithm is a standard recursive traversal of a tree, informed by two inequalities, described in Ciaccia et al:

- Ciaccia Lemma 1: If d_Or_Q > r(Q) + r(Or), then for each leaf entry Oj in the subtree of Or, d_Oj_Q > r(Q), that is, the leaf entries are outside of the query search radius.

- Ciaccia Lemma 2: If | d_Op_Q – d_Or_Op | > r(Q) + r(Or), then d_Or_Q > r(Q) + r(Or). This is due to the triangle inequality and allows the subtree of Or to be pruned.

The code may now be understood as follows: The tree is searched recursively, recursing on each successive routing node in the path until the leaves are reached. The node with which the function is initially called is always the root of the tree, and for each entry in this node, we use Ciaccia Lemma 2 decide whether it is worth the CPU cost of calculating the distance between the query point and the entry. If the lemma indicates that the entry and query radii potentially intersect, we calculate the distance between them to test whether it is possible that the entry subtree could contain a point within the query radius. For each entry where the lemma indicates that this is possible, we call the range query function with the argument node being the root of the entry's subtree. This continues all the way to the leaves. Once we reach a leaf node,

we use Ciaccia Lemma 1 to decide whether each leaf entry could potentially be in the query radius. If this is possible, we use a distance calculation to see if the entry is within the query radius. If it is, we add it to the list of query results.

As noted above, there is one modification to the Ciaccia et al algorithm which is peculiar to this study. The bolded section of the pseudocode relies on the fact that during insertion, we have calculated not only the three-dimensional spatiotemporal radius of each node, but also the covering radius of the spatial components in the node's entries. We have also calculated the spatial center of each node. With this information, the bolded code decides whether it is possible that the node being investigated could contain the spatial component of the query radius. If it cannot, there is no point in investigating the node's subtree, so it is eliminated from further consideration.

Chapter 4 – Testing Methods and Results

4.1 – The M-Tree Simulator

The enhanced M-Tree is implemented in a simulator written in Python and executing

entirely in RAM. We include a buffer simulator in the M-Tree simulator, because the

performance dynamics of the simulator are not known to be similar to a database engine written

in a non-interpreted language and operating over both RAM and disk. Thus, we can benchmark

on simulated cache behavior, as well as on observed time. Figure 3 below shows a high level

view of the main parts of the simulator:

Figure 3: Structure of the M-Tree simulator
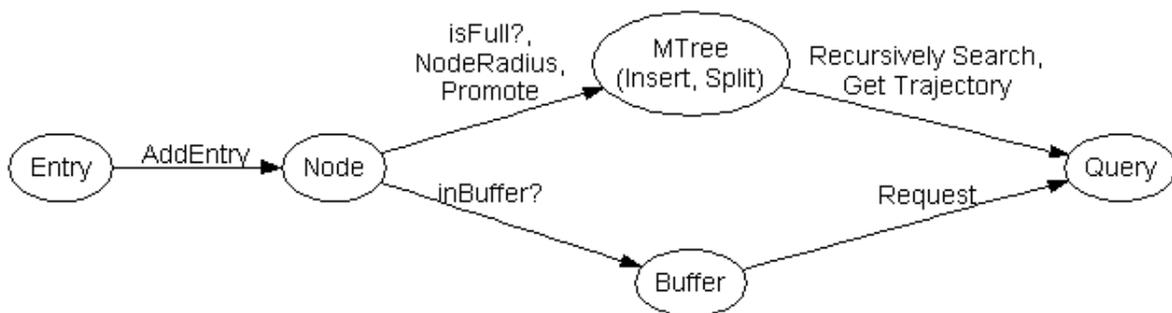
The fundamental data unit in the simulator is the Entry, which encapsulates the state of

one spatial object at one point in time, that is, its identity, spatial coordinates and timestamp. For

routing purposes, the Entry contains its distance from its parent, and routing Entries contain their

spatial and spatiotemporal covering radii, and the identity of their child nodes. Leaf Entries also

contain the identity of the Entry that preceded them in their trajectories.

The Node is composed of Entries and also holds bookkeeping information on its parent

and its node covering radii. It also encapsulates functionality to update covering radii when a

new entry is inserted and to determine when it is full, which means that a split is needed. For

splits, the Node encapsulates the promotion functionality, which decides which entries in the node are to be promoted to be parents of the nodes newly created by the split.

The MTree itself is a collection of Nodes with parent-child relationships. When a split is necessary, the Node being split picks the new parents, but the MTree is responsible for creating the new Nodes and correctly distributing entries between them. The MTree also accepts new Entries for insertion; these come from a binary load file.

The Query code is responsible for recursively searching the MTree in the manner described in Section 3.4. When a node is requested for examination, the request goes through the Buffer simulator, which determines if the Node is in the buffer, and updates statistics as needed.

The simulator equates a node in the tree with an on-disk page. This simplifying assumption is justified by considering that at a fanout of 63, a node will contain about 1.5 KB of raw spatiotemporal data (63 3-tuples of 64 bit floats) and will be larger than 1.5 KB due to metadata overhead. This value is close enough to normal on-disk and in-memory page sizes in commodity operating systems (4 KB is typical in Linux and Windows) that we do not believe it will influence the results unrealistically. Guttman's original R-Tree also mapped a single node to a single page [1].

After presenting simulated buffer performance, we will present results to show that the penalty of the increased computations incurred on insertions by our enhancements to the tree is constant and small. We feel that it is reasonable to benchmark insertion by observed time, since insertion is dominated by metric computation and requires only occasional access to Python-specific data structures.

Comparison of our results to those of other data structures is limited to the insert benchmarks because, to our knowledge, no other datastructure is adaptable to spatiotemporal data, trajectory reconstruction, and spatial network databases.

4.2 – The Dataset Under Test

The data set under test is synthetic. It is a product of the Brinkhoff spatiotemporal data generator [16], and consists of approximately 1.7 million entries. This number was chosen to result a tree with a height of four at a fanout of 63 entries per node. The objects in the dataset are constrained to motion on a spatial network. In this case, the network is a map of Oldenburg, Germany which is supplied by with Brinkhoff's generator software.

The Brinkhoff generator is designed to realistically model moving objects such as cars, and thus the motion of objects in the dataset is not random. Brinkhoff describes eight considerations which inform the patterns of motion of the objects in the dataset:

1: The objects almost always follow the network.

2: Most objects chose a fast path to their destination; e.g. a highway over a residential street.

3: Most networks have different classifications of roads which allow different traffic volumes and speeds.

4: On each type of road, there is a threshold of traffic volume where slowing begins.

5: If traffic slows, objects may plot a new course.

6: Traffic volumes depend on time of day.

7: Outside influences such as weather may affect traffic flow.

8: Different types of objects (cars, trucks, etc.) have different maximum speeds.

Points of origin in the dataset are selected with a uniform distribution over the nodes in the network. Since the nodes in a city map are not distributed uniformly, this means that denser

sections of the map will originate more vehicles. Destinations are picked based on the expectation that short trips predominate city driving.

With these considerations in mind, the Brinkhoff generator produces a moving object dataset over a network with characteristics much different from those of a randomly selected dataset. Within the constraints detailed above, there is enough statistical variety in the dataset to thoroughly exercise the simulator. For example, trajectories in this dataset have an average of 36 segments, but outliers have anywhere from 1 to 931 segments. The map is roughly square, approximately 25,000 units long on a side.

The Brinkhoff generator outputs a comma separated file which we transformed into a binary load file for the simulator. The same 1.7 million entry load file was used for all tests. The map is roughly square, approximately 25,000 units long on a side.

4.3 – Query Performance

First we will present the buffer performance of the enhanced M-Tree on a set of 3000 randomly generated spatiotemporal range queries, with an average range of 250. Recall from Chapter 1 that these are queries in which we specify a point, a time, and a range from both the point and the time, and we retrieve all points that are within both the spatial and the temporal range.

The dataset in question is the above-mentioned 1.7 million entry Brinkhoff dataset. The tree in this test had fanout held constant at a maximum of 63 entries per node. Figure 4 (below) shows that the performance gains due to increasing the buffer size decrease approximately logarithmically. This is a desirable trait, because it means that at a buffer size of less than one percent of the dataset size, performance is essentially stabilized, meaning that additional memory is not needed to improve performance.

35

Figure 4: Buffer performance on 3000 spatiotemporal range queries vs. buffer size
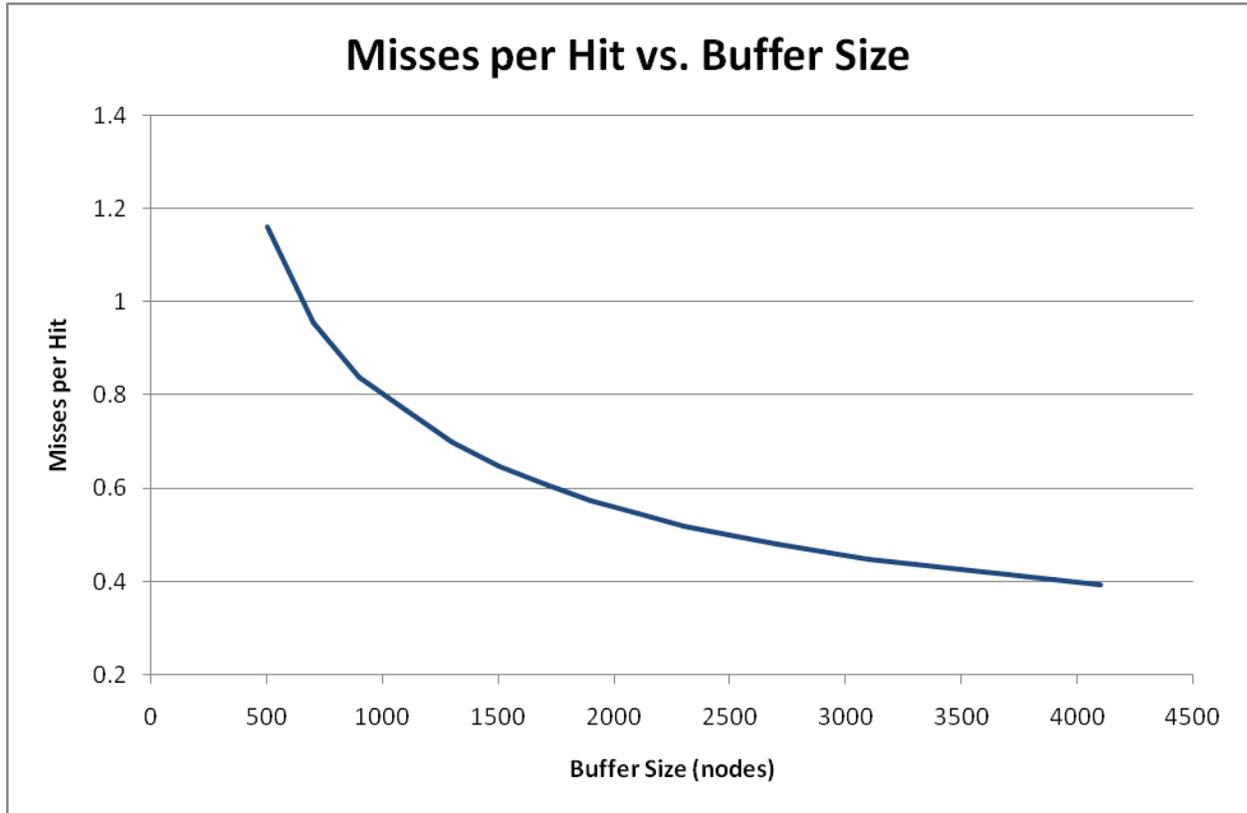
Next we will look at the effect of changing node fanout at a constant buffer size of 1000 entries. This buffer size is well under 1% of the dataset size, and thus is a good representation of a real-world situation where RAM is much more costly than disk space.

Figure 5: Buffer performance on 3000 spatiotemporal range queries vs. fanout
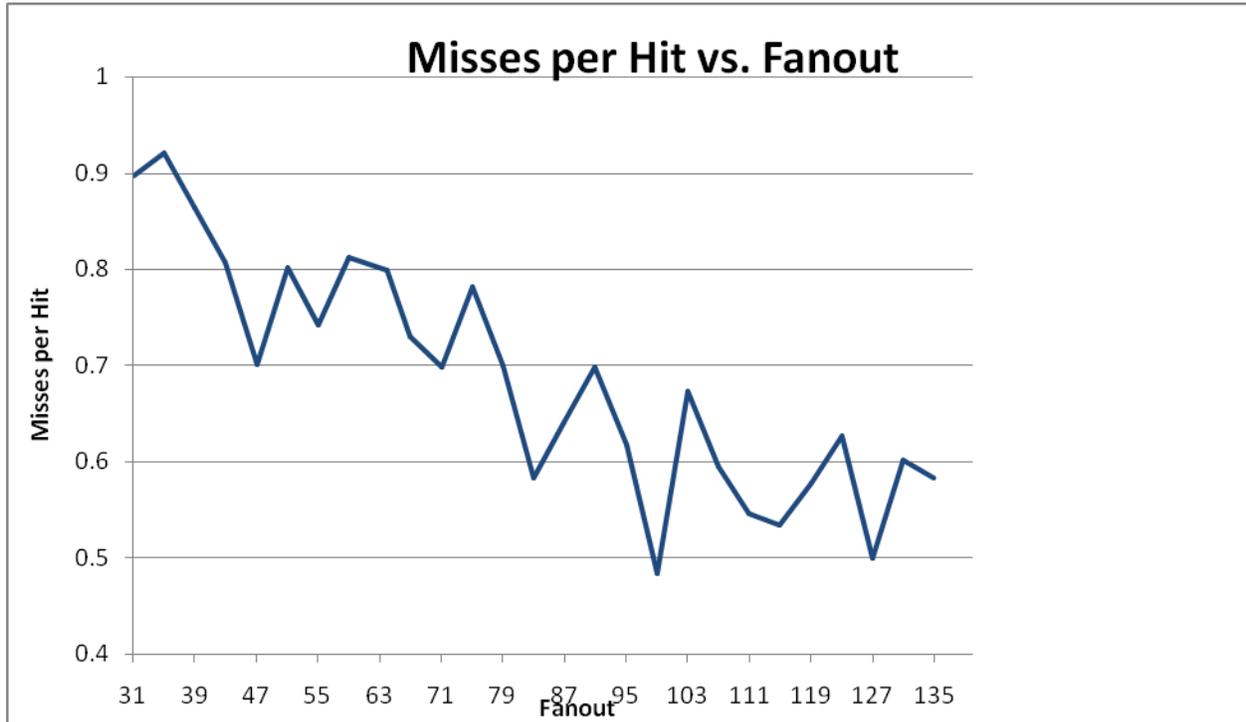
On the surface, Figure 5 shows the obvious: since we are measuring our buffer size in nodes, and since increasing the fanout decreases the number of nodes in the tree, increasing fanout should correspondingly decrease the number of misses per hit. However, it should be noticed that quadrupling the node capacity decreases the number of misses per hit by less than half. Again, we see that there is a "break-even point" in performance where increasing the availability of a resource—in this case, buffer capacity—no longer provides a corresponding increase in performance. In general, a node fanout of between 60 and 120 entries per node provides reasonable performance.

Figure 6 (below) sheds some light on why this is so. As can be seen, the total number of buffer accesses needed to answer the set of 3000 test queries begins to flatten out in a logarithmic fashion as fanout is increased. One possible explanation for why this happens is that as fanout increases, the dead space (empty space and overlap) in the tree increases, offsetting the smaller working set provided by a reduced total number of nodes.

At a fanout of 63, 1.7 million entries (the size of the dataset under test) is just larger than the minimum needed to make the tree four levels deep. Varying the fanout varies slightly the threshhold point where this happens, which may account for the somewhat jagged appearance of Figures 5 and 6: the four-level tree at fanout 63 is a three-level tree at fanout 67, since 63^4 is approximately 1.6 million, but 67^4 is approximately 2.0 million.

Figure 6: Total buffer accesses on 3000 spatiotemporal range queries vs. fanout
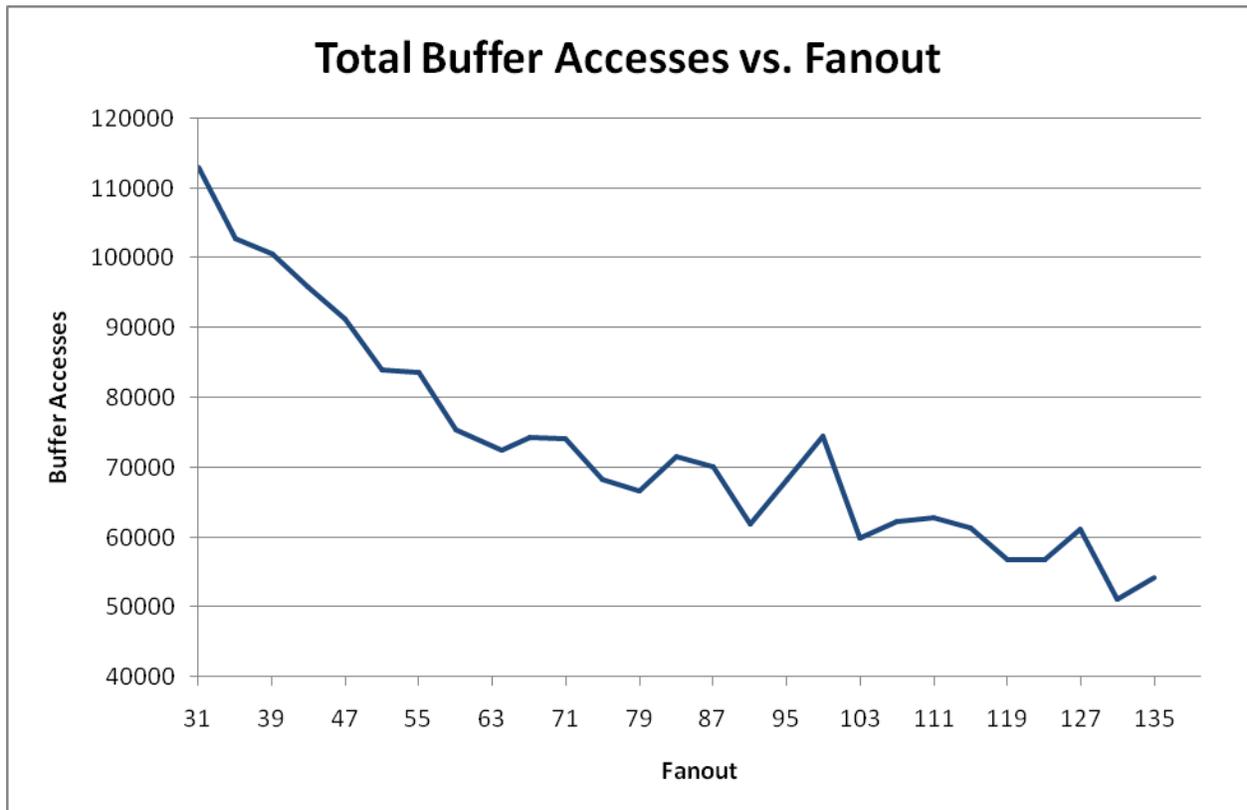
Next, we will look at the effect of increasing range on a range query, with the query point held constant. We increase the range to up to 10,000 units, much larger than in the previous test, in order to see what trends develop. Unlike the previous graph, we are showing the accesses required per query, not the cumulative accesses required to answer the whole set of queries.

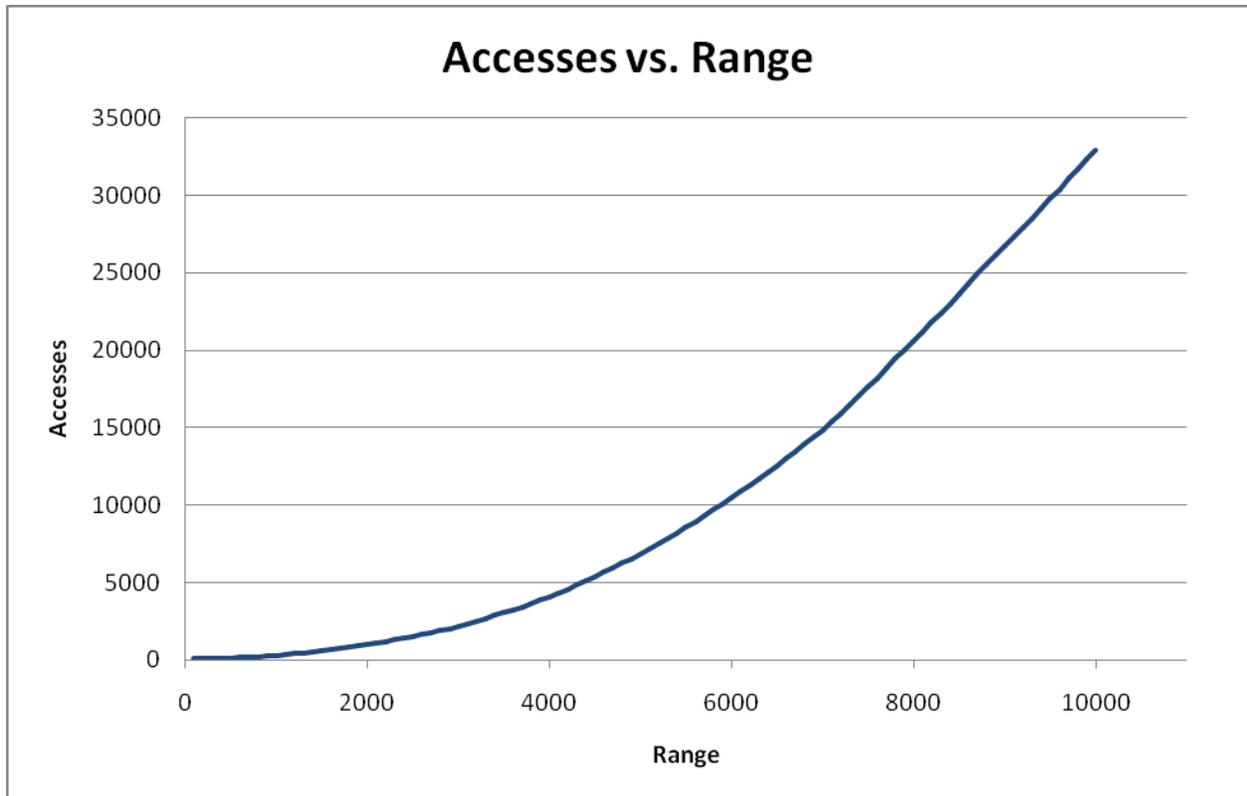Figure 7: Buffer accesses per range query, varying range

Figure 7 shows the expected result: that, as range increases, we must traverse more and more of the tree in order to answer the query. Since the tree indexes a three dimensional space, and thus the bounding volume of the range query increases in a cubic fashion with range, it is not surprising that the relationship in the graph appears to be polynomial.

Figure 8 (below) shows query time with and without trajectory reconstruction, again using the set of 3000 spatiotemporal range queries. When trajectory reconstruction is turned on, the simulator reconstructs the trajectory that each point found by each range query belongs to.

Figure 8: Query time with and without trajectory reconstruction
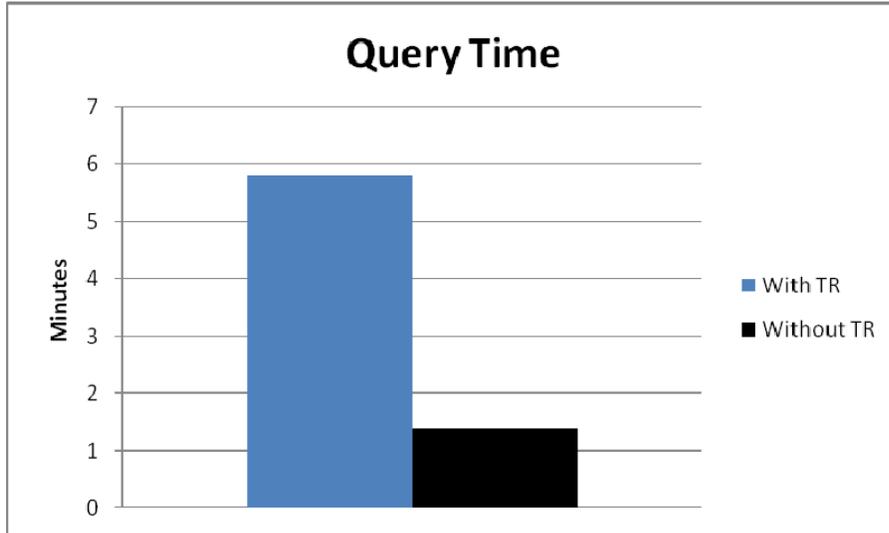
Turning on trajectory reconstruction increases the time it takes to run the set of 3000 range queries by over four times. However, recalling from Section 4.2 that the average trajectory in the data set has 36 segments, it turning on reconstruction requires the retrieval of 36 times more points, on average. It is clear that the calculations required to traverse the linked list embedded in the points which define a trajectory are not overly demanding.

4.4 – Insert Performance

Next we will look at the cost of computing the extra spatial covering radii which enable the spatial-only (two dimensional) split in the enhanced tree. Figure 9 shows that the insert time penalty for the enhanced M-Tree is constant at approximately 12%. This is to be expected since constructing the enhanced M-Tree is identical to constructing the M-Tree except for one additional node covering radius calculation per insert. Figure 9 shows that the cost of calculating the extra covering radii is relatively small and a constant factor as dataset size increases.

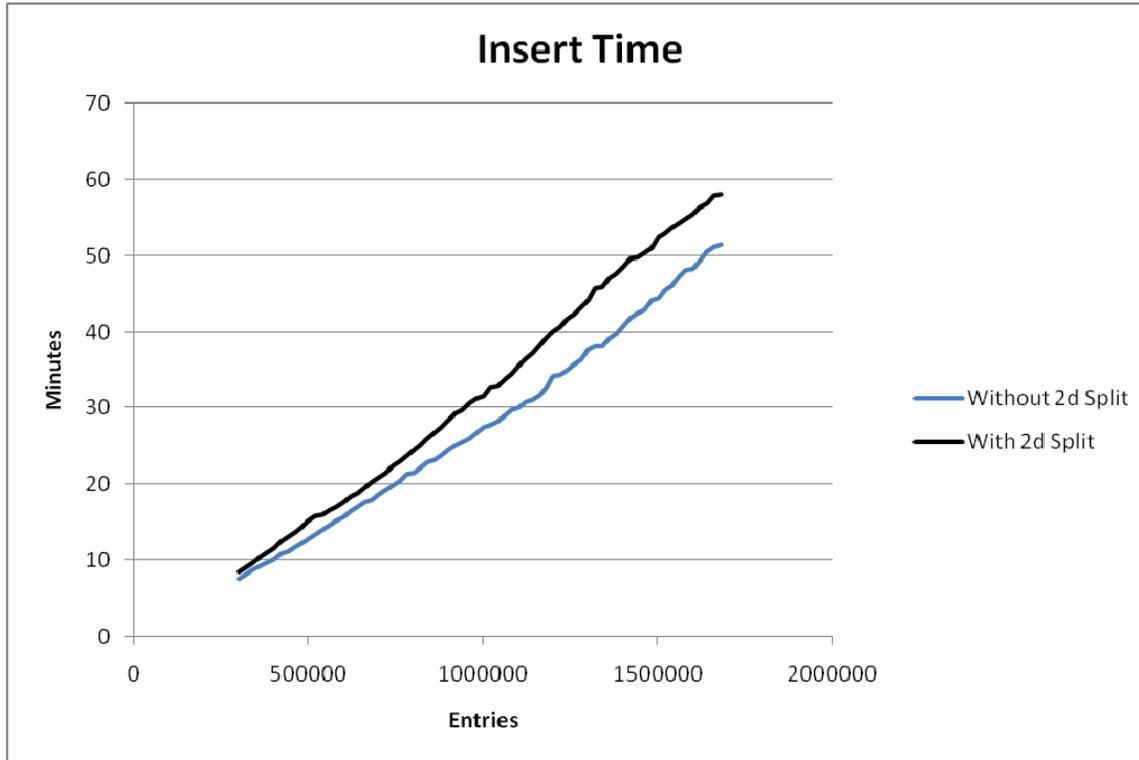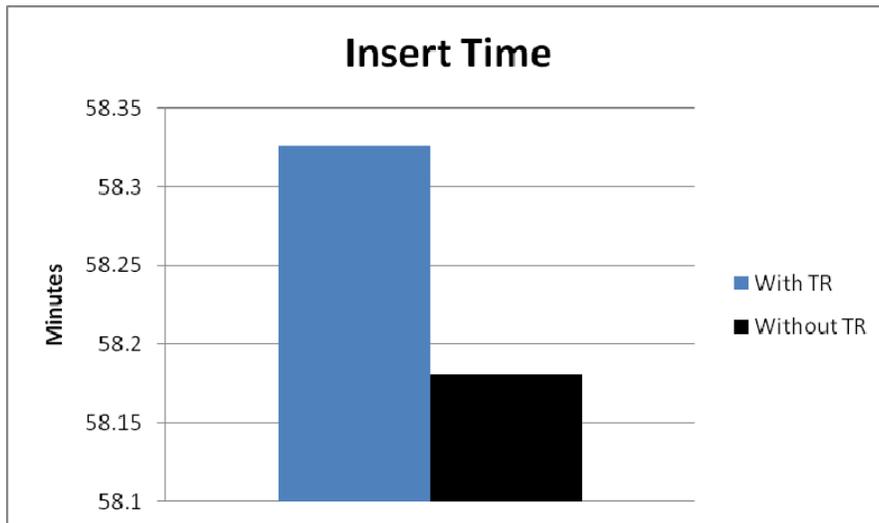Figure 9: Insert time with and without two dimensional split

Figure 10 shows the insert time penalty of turning on trajectory reconstruction. The figure shows that the penalty is insignificant; we tested only with the full 1.7 million entries to be able to show a discernable difference.

Figure 10: Insert time with and without trajectory reconstruction

Chapter 5 – Conclusion and Future Work

5.1 – Conclusions

We began by introducing the existing situation in spatial indexing. The R-Tree was one of the first spatial indexes proposed and it has served well as an index for two dimensional, non-moving spatial objects in situations where Euclidian distance is important. The R-Tree, however, is not ideal for moving objects, spatial networks, and data of high dimensionality.

We introduced the work of Givaudan, who categorized existing approaches to indexing spatiotemporal data, that is, maps of objects which can move. We introduced the concept of trajectory reconstruction and looked at the SQL constructs which are needed for querying a spatiotemporal database.

We then introduced the M-Tree, originally designed by Ciaccia et al, and introduced the work of Ioup et al, which adapted the M-Tree as an index of non-moving spatial network data. This use of the M-Tree provided motivation to adapt the M-Tree as a spatiotemporal index, since the M-Tree as implemented by Ioup et al is one of the few existing structures which can efficiently index spatial networks.

To this end, we introduced two enhancements to the M-Tree. The first allows us to preserve spatial locality in spatiotemporal data by making promotion decisions in splits based on the spatial components of the data. The second implements a linked list between the successive entries that represent the evolution of a particular object in time. This allows for basic trajectory reconstruction without the insertion overhead involved in doubly-linked lists or storing trajectories in a temporary data structure until they are known to be complete and inserting them en-bloc after completion.

We then presented test data which shows that buffer performance is reasonable with a buffer that is well under 1% of the size of the dataset, and data that shows that our overhead on insertion versus a M-Tree with none of the above described features is small.

We believe that we have shown that the M-Tree is a viable index for spatiotemporal data.

5.2 – Future Work

A clear direction for future work is the unification of this work and that of Ioup et al, to produce an access method which can handle spatiotemporal network data. Such a system should be ideal for applications that rely on real-time querying of vehicle movements, such as traffic monitoring, shipping logistics, and defense.

Two main strengths of the M-Tree are its dynamic nature (insertion, updates, and deletion are possible at any time) and its extensibility due to the ability to use domain specific metrics. We did not implement deletion and updates in the simulator, but there is no technical barrier to doing so. To use the tree in a general purpose database system, these functions would have to be implemented.

We focused on the use of Euclidian distance as the metric, but early testing showed that any metric in the L_p family worked similarly. A deeper investigation into metric functions may turn up a metric which is more effective than Euclidian distance, and a unification with the work of Ioup et al would probably require the use of Chessboard distance.

References

[1] Guttman, A. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. SIGMOD '84.

[2] Shaw, K.; Ioup, E.; Sample, J.; Abdelguerfi, M.; Tabone, O., Efficient Approximation of Spatial Network Queries using the M-Tree with Road Network Embedding. 19th International Conference on Scientific and Statistical Database Management, 2007. SSBDM '07.

[3] Korn, F.; Pagel, B.-U.; Faloutsos, C., On the "Dimensionality Curse" and the "Self-similarity Blessing". IEEE Transactions on Knowledge and Data Engineering, vol.13, no.1, 96-111, Jan/Feb 2001.

[4] Theoderidis, Y.; Vazirgiannis, M.; Sellis, T., Spatio-Temporal Indexing for Large Multimedia Applications, Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, 1996., pp.441-448.

[5] Givaudan, J. The 2-3 TR-Tree, A Trajectory-Oriented Index Structure For Fully Evolving Valid-Time Spatio-Temporal Datasets: A Thesis. University of New Orleans, 2002.

[6] Nascimento, M. A., Silva, J. R., and Theodoridis, Y. Evaluation of Access Structures for Discretely Moving Points. In Proceedings of the International Workshop on Spatio-Temporal Database Management, 1999. In M. H. Böhlen, C. S. Jensen, and M. Scholl, Eds. Lecture Notes In Computer Science, vol. 1678. Springer-Verlag, London, 171-188.

[7] Nascimento, M. A. and Silva, J. R. 1998. Towards historical R-trees. In Proceedings of the 1998 ACM Symposium on Applied Computing, 1998. SAC '98.

[8] Tao, Y. and Papadias, D. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In Proceedings of the 27th international Conference on Very Large Data Bases, 2001. P. M. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 431-440.

[9] Pfoser, D., Jensen, C. S., and Theodoridis, Y. Novel Approaches in Query Processing for Moving Object Trajectories. In Proceedings of the 26th international Conference on Very Large Data Bases , 2000. A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 395-406.

[10] PostGis. www. postgis. org.

[11] Viqueira, J. R. and Lorentzos, N. A. SQL extension for spatio-temporal data. The VLDB Journal. vol. 16, no. 2, 179-200. Apr. 2007.

[12] Aho, A. V., Denning, P. J., and Ullman, J. D. Principles of Optimal Page Replacement. Journal of the ACM vol. 18, no. 1, 80-93. Jan. 1971.

[13] Ciaccia, P., Patella, M., Rabitti, F., Zezula, P. Indexing Metric Spaces with M-tree. Atti del Quinto Convegno Nazionale su Sistemi Evoluti per Basi di Dati (SEBD'97). Verona, Italy. Pages 67-86. 1997.

[14] MapQuest. www. mapquest. com.

[15] Shahabi, C., Kolahdouzan, M. R., and Sharifzadeh, M. A road network embedding technique for k-nearest neighbor search in moving object databases. In Proceedings of the 10th ACM international Symposium on Advances in Geographic information Systems 2002. GIS '02.

[16] Brinkhoff, T., Generating network-based moving objects. In Proceedings of 12th International Conference on Scientific and Statistical Database Management, 2000. SSDBM 2000.

Vita

John Finigan was born in New Orleans in 1981. He obtained his Bachelor of Arts in Computer Science from Fordham University in New York in 2004.