Indexing Large Trajectory Data Sets With SETI*

V. Prasad Chakka Adam C. Everspaugh Jignesh M. Patel

University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
USA
{vchakkab, aeverspa, jignesh}@eecs.umich.edu

Abstract

With the rapid increase in the use of inexpensive, location-aware sensors in a variety of new applications, large amounts of time-sequenced location data will soon be accumulated. Efficient indexing techniques for managing these large volumes of *trajectory* data sets are urgently needed. The key requirements for a good trajectory indexing technique is that it must support both searches and inserts efficiently.

This paper proposes a new indexing mechanism called SETI, a *Scalable and Efficient Trajectory Index*, that meets these requirements. SETI uses a simple two-level index structure to decouple the indexing of the spatial and the temporal dimensions. This decoupling makes both searches and inserts very efficient. Based on an actual implementation, we demonstrate that SETI clearly outperforms two previously proposed trajectory indexing mechanisms, namely the 3D R-tree and the TB-tree.

Unlike previously proposed trajectory indexing structures, SETI is a logical indexing structure that uses existing spatial indexing structures, such as R-trees, without any modifications. Consequently, DBMSs that currently support R-trees can easily implement SETI, making it a both a practical and an efficient choice for indexing trajectory data sets.

*This work is supported in part by NSF under grant IIS-0093059, and by an IBM Faculty Award.

Proceedings of the 2003 CIDR Conference

1 Introduction

The need to accurately determine location has been a critical part of exploration endeavors since the early days of human history. The advent of satellite and atomic clock technologies in the second half of the last century lead to the development of satellite-based location determination techniques, such as the GPS [10]. For a long time, the GPS technology was only available for military applications, but since May of 2000 it has become possible for *civilian* GPS receivers to accurately identify the location of a GPS receiver within a few meters. GPS technology is quickly enabling a number of new applications, including tracking fleets of vehicles, navigating boats and ships, and tracking wildlife. Another popular application of this technology is in cellular phones with embedded GPS sensors. In the United States, cellular phones are now required to be E911-enabled. E911 is a federally mandated requirement [8] which states that cellular phone companies must be able to locate the geographical location of the cellular phone user with an accuracy of a few hundred meters in most cases. In case of an emergency, E911 will provide better location information to the emergency workers.

While GPS works well for computing the location in the outdoors, determining location when inside a building requires using other techniques. A popular technique for determining location indoors is using ultrasonic, or radio frequencies, or a combination of these two techniques [2, 13, 23, 31]. Location information produced by these techniques can be very accurate; for example, the BAT system [13, 31] can compute locations that is usually accurate within 9 cm of the actual location! The availability of indoor location information is also giving rise to a new class of applications, such as asset tracking and context-aware applications that adapt to the users needs as the user moves around in physical space. Rapid advances in semiconductor technologies have made it possible to build such location-computing devices for a small price, making mass deployments of such devices possible.

Using these location-computing techniques, as an ob-

ject moves around in space we can gather the successive location positions of the object. These successive locations can be viewed as a sequence of line segments that collectively form the *trajectory* for that object. Thus, a trajectory segment for an object moving in a k-dimensional space is essentially a line in a $k+1$-dimensional space, with time as the additional dimension. In the last few years, we have seen a rapid increase in the deployment of location-sensing devices and applications that make use of this information, and this trend is likely to accelerate in the near future. As a result, we will soon be faced with the task of managing large volumes of trajectory data. For example, if one were to continually collect GPS sensor readings from a fleet of one hundred thousand trucks, transmitting their location every minute, then the data set grows by 144M new segments every day. Such trajectory data sets can be used in a number of ways, including analyzing factors that contribute to accidents and examining factors that lead to missed delivery targets. If the trajectory data is combined with data from other instruments in the vehicle, such as the odometer, the braking system, and the fuel gauge, then this information could be used to identify factors that contribute to poor fuel consumption.

Indexing techniques have been used very effectively for managing large data sets in other application domains, and it is natural to expect that indexing techniques will play a crucial role in managing trajectory data sets too. In this paper we address the issue of efficiently indexing large trajectory data sets. We propose and present a new indexing structure called SETI, a Scalable and Efficient Trajectory Index, that decouples the indexing of the spatial dimensions from the time dimension. SETI partitions the spatial dimension into static, non-overlapping partitions, and for each partition, it builds a sparse index over the time dimension. Any spatial index can be used for this sparse index, including the R-tree [12], and its variants like the R*-tree [4].

The SETI indexing technique has the following key features and advantages over other trajectory indexing techniques:

- Since the objects that are actually indexed are one-dimensional time lines, the indexing structure does not suffer from the curse of dimensionality [5], which causes the performance of indexing structures to degrade rapidly as the number of dimensions increases.

  A straightforward way of indexing trajectories is to store each trajectory segment in a 3D R-tree [12], and then use standard R-tree search algorithms. Based on an implementation in SHORE [7], we show that SETI clearly outperforms this indexing approach.

  We have also implemented a recently proposed trajectory indexing structure, the TB-tree [21], in SHORE, and in this paper we present the results comparing the performance of SETI with the TB-tree. The performance comparisons show that SETI also convincingly outperforms the TB-tree.

- Updates to trajectories tend to be append operations that add new segments to the ends of existing trajectories. To support high append rates, SETI keeps the last location of each object in an in-memory *front-line* structure. When a new location update for an object is available, we look up the last position of that object in the front-line structure and add the trajectory segment to the SETI index. Since we only use a sparse R*-tree index on the time dimension, such updates are very fast.

- The index scales well to handle large trajectory data sets because of the use of multiple sparse indices. We show that SETI scales well both with increasing number of mobile objects (actual users or devices) and increasing trajectory sizes (i.e. increasing number of segments per trajectory).

- Finally, SETI can be viewed as a logical indexing structure that can be built on top of an existing spatial indexing techniques, such as an R-tree. Consequently, implementing SETI is much easier than implementing a new physical indexing structure. In addition, existing techniques for concurrency control that have been developed for R-trees [15, 16] can be directly used by SETI.

The remainder of this paper is organized as follows: The problem definition and related work are presented in Section 2. The SETI indexing mechanism is presented in Section 3. Section 4 presents the experimental results based on an actual implementation of SETI, 3D R-trees and the TB-tree. Finally, Section 5 contains our conclusions and plans for future work.

2 Problem Definition and Related Work

2.1 Data Model

Trajectory data for moving objects is continuously changing between any two successive updates of the location of the mobile object. This poses a problem in representing the location of the object at all times because most conventional models for data representations are static in nature. A commonly used model for representing trajectory data approximates the motion of an object as a straight line segment between two consecutive updates [11, 14, 19, 21, 22, 24]. In this paper, we also use this model for representing trajectories. The position of a moving object is sampled at discrete times, and a series of straight lines connecting successive positions represents the movement of the object. Here after in this paper, this line is referred to as a *segment*, and the sequence of the connected segments for a single moving object is referred to as a *trajectory*. Furthermore, in this paper we will assume that an object moves

in a two-dimensional space, although extensions to higher-dimensions are fairly straightforward.

Stated more formally, a trajectory is represented as $trj(tid, \langle u_0, u_1, u_2 \dots u_n \dots \rangle)$, where tid is a unique trajectory id, and $\langle u_0, u_1, u_2 \dots u_n \dots \rangle$ is a sequence of points reflecting the positions of the moving object. Each point u_i is a three-tuple $u_i(x_i, y_i, t_i)$, where x_i and y_i represent the spatial position of the object along the x and y dimensions respectively, at time t_i. The only restriction on the sequence is that $u_i < u_{i+1}$ to ensure that the time parameter in the trajectory sequence is monotonically increasing.

A trajectory segment is represented as $s_i(tid, sid_i, u_{i-1}, u_i)$, where sid_i is a unique segment number for this segment of the trajectory (trivially one can set sid_i to i), and u_{i-1} and u_i are the two update end-points. The model can be easily extended to associate additional variables with each segment or update point; for example, in a trajectory data set of moving vehicles, each update point may have an additional reading recording the engine temperature at the time of the update.

2.2 Query Types

Queries on moving data can be broadly classified into two categories: queries that ask questions about the *future* positions of moving points, and queries that ask questions about the *historical* positions of moving objects. The former class of queries can be answered by storing current position, speed and the direction of the moving objects [1, 14, 24, 25]. For the second class of queries, Pfoser et al. [21] further classify historical queries into two different sub-classes: coordinate-based queries and trajectory-based queries. Coordinate-based queries include (a) time-interval, which select all objects within a given area and give time period, (b) time-slice queries, which select all the objects present in a given area at a time instant, and (c) nearest neighbor queries. Trajectory-based queries involve information about a trajectory such as topology and velocity.

In this paper, we focus on coordinate-based queries in general, and time-interval and time-slice queries in particular.

2.3 Related Work

An efficient indexing structure for trajectory data sets should essentially be scalable both in the number of moving objects it supports and the update rate of the objects. Most of the previous access structures in the literature are based on variations of the R-tree [12], and do not address scalability specifically. The earliest of these structures is the RT-tree which stores time in the nodes of a regular 2D R-tree [32]. Any temporal query has to search the entire R-tree as there is no discrimination in the index search along the temporal dimension. 3D R-trees [29] treat the temporal property simply as an additional dimension. The weakness of 3D R-tree is that the temporal dimension is treated in the same manner as the spatial dimensions. Since a bounding box in the index now also includes the time dimension, the overlap amongst the keys increases, and the dead space also increases. Consequently, the performance of the 3D R-tree degrades rapidly as the data set size increases. MR-trees [32] and HR-trees [17] are similar, as both maintain a separate R-tree for each time stamp. Duplication of unchanged nodes in consecutive R-trees is avoided to reduce the storage space. These indexing structures are efficient for evaluating time-slice queries, but search performance degenerates for time-interval queries. In addition, the storage space required by HR-trees is typically very high [18].

The MV3R-tree [27] is a hybrid structure that uses a multi-version R-tree (MVR) for time-stamp queries, and a small 3D R-tree for time-interval queries. MVR tree is three dimensional extension of multi-version B-trees [3]. A 3D R-tree is built on the leaf nodes of the MVR-tree. To keep the space requirement manageable, the two indices share the same leaf pages, which leads to a rather complex insert algorithm. In addition, the MVR-tree models temporal change as a discrete event; moving objects maintain the same position until the new position is updated. For the discrete event data model, the MV3R tree outperforms other indexing structures, such as the 3D R-tree and the HR tree. But the main drawback of this model is that it can not be used to represent the gradual change of position, which is required in the trajectory model that we use in this paper (see Section 2.1).

The TB-tree [21] is a trajectory bundle tree that is based on the R-tree. The main idea of the TB-tree indexing method is to bundle segments from the same trajectory into the leaf nodes of the R-tree. The R-tree insert algorithm is modified so that leaf nodes contain trajectory segments belonging to only one moving object. If segments of a single trajectory need more than one leaf node for representing the segments, then the leaf nodes that store the trajectory bundles for the same trajectory are connected by forward and backward pointers. This design aims to reduce the node accesses for retrieving a complete trajectory. Trajectory retrieval becomes very easy with this structure, which is important for topological queries. However, since the R-tree insertion is based solely on trajectory id rather than proximity in both the space and time dimensions, one can expect large overlaps among the minimum bounding box (MBB) keys in the internal nodes of the R-tree. However, updates to the index are made in chronological order thus reducing the overlap.

In many spatio-temporal data management systems, aggregate data is maintained rather than the actual data (e.g. traffic flow patterns). This aggregated data representation may also be used to address privacy concerns. Papadias et al. [19], propose the aggregate R-B-tree (aRB-tree) which maintains the spatial regions in a R-tree, and uses a B-tree

for each of the regions in the R-tree on the temporal dimension. The aggregate values for time stamps are maintained in the B-trees. Volatile regions are handled using Historical R-trees. By the very nature of aggregate indexes, aRB-trees are limited in the types of queries that they can support, and can not be extended to support the ones that we consider in this paper.

Song and Roussopoulos [26] consider the problem of updates for a large number of moving objects in a system that stores only the latest location information. They assume that only approximate location information is required, and partition the working space into non-overlapping cells. They solve the update problem by storing only the cell number for each object, and updating the cell number associated with an object only if the object moves to a different cell. This approach is not adequate for dealing with trajectory data sets for two reasons: it does not provide accurate position information, and it does not support spatio-temporal queries since it does not store the detailed trajectory data of moving objects.

To the best of our knowledge, previous works on trajectory indexing have not considered the efficiency of the index operation that appends new segments to existing trajectories.

3 SETI

3.1 Description

Trajectory data sets of line segments exhibit characteristics that are different from general three dimensional data sets. Rather than using an R-tree for storing 3-D line segments, these differences can be exploited to design a more efficient indexing structure. In a 3D R-tree, the temporal and the spatial dimensions are treated equally. However, for trajectory data sets, there are important differences in the characteristics of these dimensions. More specifically, the boundaries of the spatial dimensions remain constant or change very slowly over the lifetime of the trajectory data set growth, whereas the time dimension is continually increasing. Since the extent of the spatial dimensions does not change, an indexing structure could partition the spatial dimensions statically (see Section 3.5 for a discussion on adapting dynamic partitioning). Within each spatial partition, the indexing structure only needs to index lines in a 1-D (time) dimension. Consequently, such an approach will not exhibit the rapid degradation in index performance that is generally observed for 3-D indexing techniques. The SETI indexing mechanism capitalizes on this observation to achieve good spatial and temporal *discrimination*[1].

In SETI, spatial discrimination is maintained by logically partitioning the spatial extent into a number of non-overlapping spatial *cells*. Each cell contains only those trajectory segments that are completely within the cell. If a trajectory segment crosses a spatial partitioning boundary then that segment is split at the boundary, and inserted into both cells (see Section 3.2 for more details).

[1]The term discrimination has frequently been used in the literature before and refers to the effectiveness of an indexing structure in identifying a candidate set of index entries with few false positives.

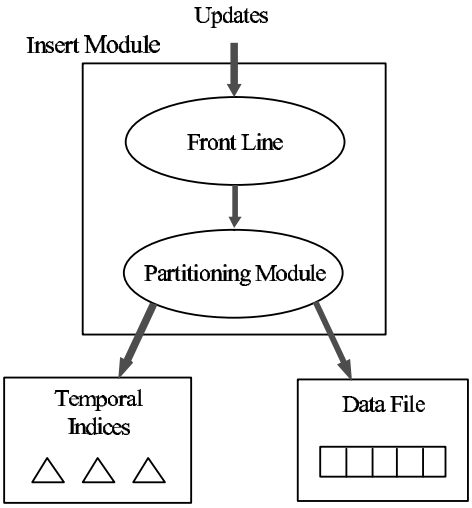

Figure 1: Schematic Diagram of the Insert Procedure

Each trajectory segment is stored as a tuple in a data file, with the restriction that any single data page only contains trajectory segments that belong to the same spatial cell. The *lifetime* of a data page is defined as the minimum time interval that completely covers the time-spans of all the segments stored in that page. The lifetime values of all pages that are logically mapped to a spatial cell are indexed using an R*-tree. These *temporal indices* are sparse indices as only one entry for each data page is maintained instead of one entry for each segment. Using sparse indices has two distinct advantages: smaller index overheads and improved insert performance. The temporal indices also provide the temporal discrimination in searches.

In most of the trajectory applications, location updates arrive in chronological order. This order of updates essentially makes the R-tree indices *clustered*, with the entries being clustered by the end-times of the trajectory segments. This clustering property results in additional index efficiencies when fetching trajectories that overlap with a time-range specified in a query. Note that even if the trajectory updates arrive in near-chronological order, the resulting index is nearly clustered, and has most of the performance benefits of a fully clustered index.

3.2 Insert

Figure 1 illustrates the insert procedure in SETI. The key to the performance of the insert algorithm in SETI is the use of an in-memory structure, called the *front-line*, which maintains the last updated location of all moving objects. The front-line is a cache of the last positions of all objects, and these positions are organized in a hash structure indexed

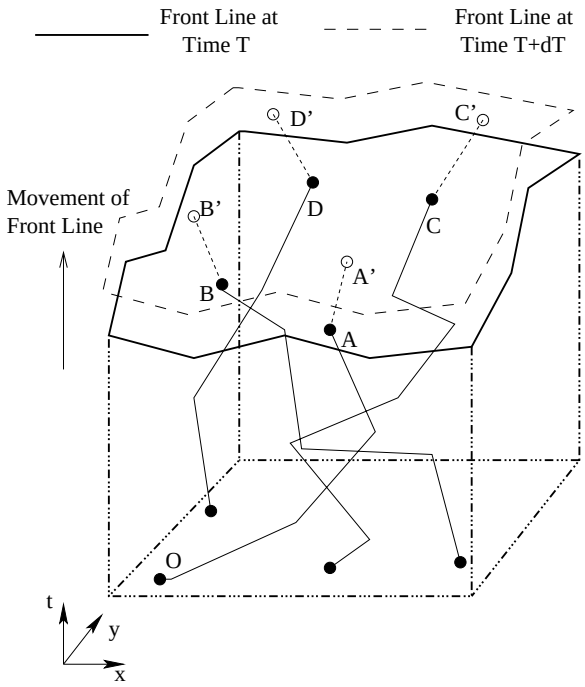

Figure 2: Movement of the Front-Line Structure

on the unique id associated with the moving object. When a new update for a moving object is presented to the system, the front-line structure is consulted to pull out the last known location for the moving object. Then a trajectory segment based on the previous and the new location is constructed and inserted into the SETI index. The front-line structure is updated with the new location of the moving object.

If persistence of the front-line structure is important, the front-line structure can be implemented using a persistent index structure, such as a hash or a B+-tree index.

3.2.1 Example of the Insert Procedure

We now illustrate the insert procedure using the example shown in Figure 2. This figure shows four trajectories that belong to four different moving objects. Point A is the latest location information for moving object O, maintained in the front-line structure. When the object O moves to a new location, A', an update request is sent to the *insert module* shown in Figure 1. Now the segment AA' represents the movement of object O between the two updates. The insert module then determines the particular spatial cell that the location A' belongs to using the *partitioning module*.

If the segment AA' spans multiple spatial cells, it is split into a number of smaller segments as shown in the Figure 3. In the figure, the spatial extent is partitioned using regular hexagons, which are shown in dashed lines. (As discussed later in Section 3.5, other adaptive spatial partitioning strategies can also be used with SETI).

The long segment AA' is broken into two smaller segments: AX and XA', with location X being the intersection

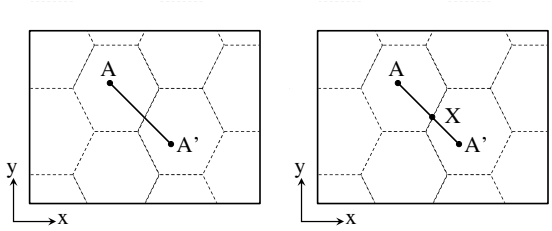

Figure 3: Splitting Segments During Insert

of the segment AA' with the cell partition boundary. Location X becomes a *logical update* location, whereas locations A and A' are actual update points. The segments AX and XA' are inserted into the corresponding spatial cells. Note that this techniques does not alter the location data for the moving object because logically the two stored segments, AX and XA', still represent the single segment AA'. The original segment can be retrieved by merging AX and XA' which will have the same segment identifier (see Section 2.1).

Even though Figure 3 shows the split in two dimensions, in actuality the segment is split in three dimensions, including the temporal dimension. The time of the logical update at the spatial boundary is determined by interpolating the times between the two successive updates (locations A and A').

3.3 Search

Figure 4 shows the steps in the search algorithm. The input to the search algorithm for a time-interval query is a three dimensional query box, which consists of a spatial predicate box and a temporal predicate range. For a time-slice query, the temporal predicate range consists of only a single value.

The search algorithm executes the following steps:

1. *Spatial Filtering*: In this step the spatial partitions that overlap with the spatial predicate box is computed and a candidate cell list is produced.

2. *Temporal Filtering*: For each of the cells in the candidate cell list, the corresponding temporal index is probed with the temporal predicate range. This step generates a list of pages whose lifetimes overlap with the temporal predicate. For time-slice queries, those data pages whose lifetimes contain the predicate timestamp are fetched.

3. *Refinement Step*: Each page in the candidate list of pages is processed as follows. If the page belongs to a spatial cell that is completely inside the spatial predicate box, then only the temporal predicate range is applied. If the temporal predicate range contains the lifetime of the page, then all the segments on the page qualify for the result, and no additional refinement steps are needed. For all the other pages query

predicates are applied for each segment on the page. This step produces a list of segments that overlap with the query box.

4. *Duplicate Elimination*: The set of segments produced in the refinement step may have multiple segments belonging to the same trajectory (or moving object). If only unique trajectory identifiers are required as part of the result, then the duplicates must be eliminated. This is done by maintaining a bitmap of trajectory id's. Each bit in the bitmap corresponds to a unique trajectory id. If a segment is qualified through the refinement step, the corresponding trajectory id bit is set in the bitmap. If the bit is already set, the segment is discarded. At the end of this step, the bitmap is scanned producing a list of trajectory ids corresponding to bit positions that are turned on.

   If the search algorithm needs to produce both the trajectory id and the segment number of the matching segment, the (trajectory id, segment number) pairs are written out to a file, which is then sorted to eliminate duplicates.

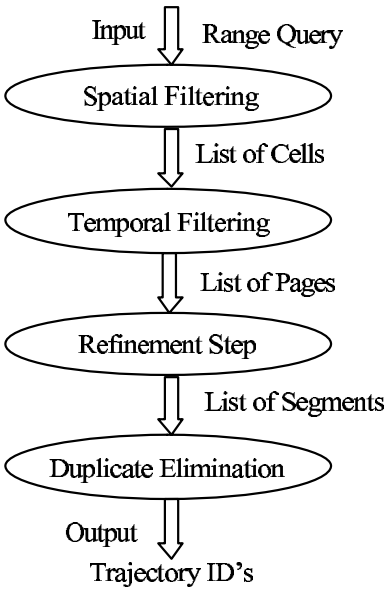

Figure 4: Schematic Diagram of the Search Algorithm

3.4 Deletes and Updates

There are two types of deletions: either a complete trajectory is deleted, or a particular segment of trajectory is deleted. If a segment is deleted, then a time-interval query is issued using the bounding box of the particular segment as the predicate. This operation retrieves the required segment data, which is then deleted from the data page. If the lifetime of the data pages changes, the corresponding entry in the temporal R-tree is updated.

To delete an entire trajectory, all the segments of that trajectory must be identified. This information is maintained using an auxiliary composite B+-tree index on the trajectory id and the segment number of the trajectory. Once all the segments of the trajectory are identified, they are then deleted one at time as outlined above.

Modifications to entire trajectories are treated as deletion followed by an insertion.

3.5 Spatial Partitioning and Spatial Skew

A good spatial partitioning is one in which the number of moving objects per cell is fairly uniform. Producing a good partitioning strategy is challenging as the distribution of the objects may be non-uniform, and the distribution may change over time. Partitioning strategies may be static or dynamic. In a static partitioning strategy, as described above, the partition boundaries are fixed, whereas in a dynamic partitioning strategy the partition boundaries may change over time.

A crucial parameter for any partitioning strategy is the number of partitioning cells, which in turn affects the area covered by the spatial cells. If the spatial cells cover large areas, then the spatial discrimination of the index is reduced, which can adversely affect the search performance. On the other hand, if the spatial extent in partitioned very finely, then then number of segments that cross a cell boundary increases, which in turn increases the partitioning overhead.

Intuitively one might expect a static partitioning of the space to become inefficient when the underlying object densities (the number of moving objects per cell) is skewed. As some cells become overloaded, the performance of the corresponding temporal R-tree is expected to deteriorate. However, this inefficiency does not usually arise because the temporal R-Tree are sparse indexes. As a result, the temporal indices can grow to index a large number data pages with only a small deterioration in the performance of the index. In addition, the chronological (or near-chronological) update order makes the R-Tree grow in one direction, resulting in key values at the leaf level that have very little overlap. In general, the performance of an R-Tree starts to degrade if the overlap amongst the keys becomes very large, which then requires traversing multiple paths during the search operation. However, in SETI, such degradation is usually not seen for the temporal R-trees, because of the chronological arrival order. With the chronological arrival order, even as the data set being indexed by a temporal R-tree increases rapidly, the overlap among the keys in the index does not increase as rapidly, reducing the impact of any partitioning skew.

Under other circumstances, such as random arrival of updates or topological change of the spatial extent, we may need to repartition the spatial dimensions. There are a number of alternatives available. The simplest is to split the partitions that are heavily populated, and rebuild the corresponding temporal R-tree indices. The resulting spatial cell

boundaries are now not uniform, and can be indexed using a spatial index structure like a Quad-tree [9]. Another more complex alternative is to keep track of the history of splits on the spatial partitions using a structure like the Historical R-Tree [17]. Existing R-tree indices on the time dimension are not rebuilt, but future appends to the cells that are split are inserted into new temporal R-tree indices. With small changes, the search algorithm for SETI can be adapted to handle this dynamic partitioning strategy.

3.6 Other Spatial Partitioning Methods

While in the previous section, we have described a static, uniform, non-overlapping partition of the spatial dimension, it is fairly straight-forward to adapt SETI to accommodate other spatial partitioning strategies, such as partitioning into non-uniform cells and overlapping spatial partitioning strategies. In the interest of space, we do not consider these options further in this paper.

4 Experimental Evaluation

In this section we present the experimental results evaluating the behavior of SETI and two other trajectory indexing structures: the 3D R*-tree and the TB-tree [21]. The 3D R*-tree simply stores each trajectory segment as a 3-D line segment in a R*-tree. The TB-tree is essentially an R-tree index, with the modification that the leaf nodes only contain trajectory segments from the same trajectory.

4.1 Implementation Details and Experimental Platform

The experimental platform is a Intel Pentium III 600MHz machine that is configured with 384 MB of main memory, and a 60GB IBM Deskstar 7200 RPM Ultra ATA/100 disk, running Debian Linux version 2.4.13.

The software that we use for our experiment is a system, called COMET, that we are currently building to research issues in continuous management of evolving trajectory data sets. This system uses SHORE [7] is its storage manager, and all indexing techniques that we examine in this section are implemented in SHORE. SHORE currently supports 2D R*-trees, so the R*-tree key definition in SHORE was modified in a fairly straight-forward fashion to support the 3D R*-trees. We implemented the TB-trees as an additional indexing mechanism in SHORE. Finally, SETI is implemented using the existing 2D R*-trees in SHORE. The spatial extent is partitioned using a uniform rectangular grid, and the front-line structure is implemented using a hash index that is mapped to a single large SHORE object for persistence. In all our experiments, we use a buffer pool of 64MB. The disk page size used in all the experiments is 2KB. Values in all the dimensions, including the time dimension, are represented using 4-byte integers.

All trajectory segments are stored as separate tuples in a single SHORE file. Each tuple has a unique trajectory id, a segment number, and the two end points of the trajectory segments.

4.2 Data Sets and Queries

Since no real trajectory data sets are currently freely available, we generated synthetic data sets using two different methods: the first method uses the GSTD [28] data generator, and the second method uses the network data generator [6].

The GSTD data generator produces trajectory data sets for a specified number of moving objects, with a specified number of segments per moving object. The GSTD generator has numerous knobs for changing the distribution of the initial positions of the moving objects, the direction of the movements, trajectory segment length, etc. We experimented with a number of data sets produced by varying some of these parameters, and found that the results using the default uniform distributions were representative of these other data sets too. In the interest of space, we only present results for GSTD data sets that were generated using the default data generation parameter values.

The network data generator models the movements of users moving in a network of paths (such as roads). It takes as input a map data set and then generates trajectories for users moving in the paths in the map. The network data set that we use in this paper was generated using the TIGER [30] data files for the road network in San Joaquin County, CA. In this data set, the distribution of the moving objects is based on the density of the road networks, with a larger population of mobile objects in areas of dense road networks.

In the real world, mobile users often remain stationary at a specific location for a long period of time. The resulting trajectory segment has a long span on the temporal dimension, which increases the life times of data pages that are indexed by the temporal R-trees. Intuitively, this characteristic should lead to a deterioration in the performance of the temporal indices. To explore the effect of such data sets, we modified the network data generator to simulate cell phone users. The simulation time period was thirty days, and in each day every user moves along the San Joaquin County road network for roughly the first eight hours of the day. For the remaining part of the day, the users remain stationary. The number of users is varied from one to ten thousand, which produces data sets with a half million to five million trajectory segments.

Most of the results presented in this section use data sets generated with GSTD[2]. However, we also present two results using the San Joaquin network data set.

[2]We make this choice since GSTD has been extensively used in most previous work in this area, making it easier for a reader to put these results in perspective.

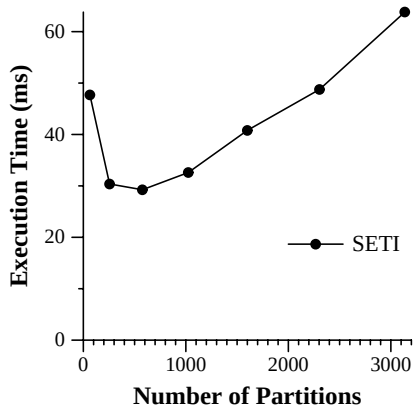

Figure 5: Effect of Number of Spatial Partitioning Cells, GSTD(1K, 4M), 0.1% Time-interval Query

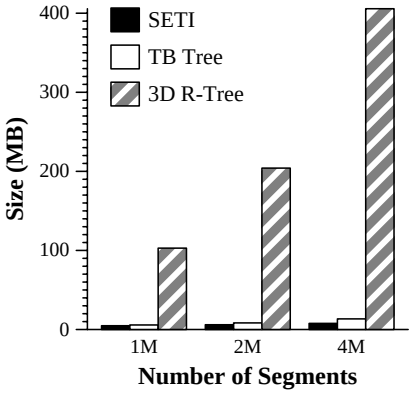

Figure 6: Index Sizes, GSTD(1K, X)

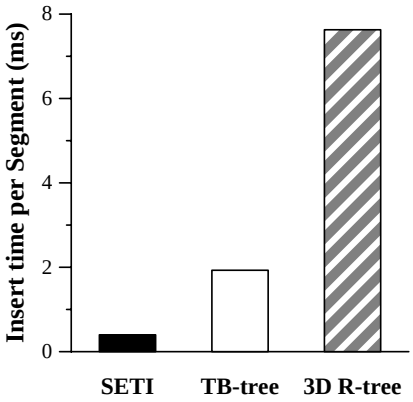

Figure 7: Comparing Insert Performance, GSTD(1K, 4M), 10K Inserts

For the GSTD data sets, we use a number of different data sets to examine the effect of increasing the number of mobile users, and the effect of increasing the number of trajectory segments per mobile object. We denote the GSTD data sets using the nomenclature GSTD(n_T, n_S) where n_T is the number of trajectories (moving objects) in the data set, and n_S is the *total* number of trajectory segments in the data set. Similarly, NetSJ(n_T, n_S) is used to represent the network data set, and NetSJ-CP(n_T, n_S) is used to represent the data set with cell phone users moving in the SanJoaquin County road network.

In the experiments, we use two types of queries: time-interval queries and time-stamp queries. The predicate in a *time-interval* query is a 3D box of equal *normalized* widths in each dimension. Since the extent along each dimension can be different, the range of the query box in each dimension is normalized by the actual extent of that dimension. The selectivity of a time-interval query is equal to the volume of the query box in a universe of unit length in each dimension. In this paper we present results using query selectivities of 0.01%, 0.1% and 1%. Note that a 1% query corresponds to a range predicate with a selectivity of 21.6% along any single dimension.

For a *time-slice* query, the time interval is a single time-stamp value, and the range along each of the spatial dimension is the same. The selectivity of the query is the area of the spatial predicate box in a universe of unit length along each of the spatial dimensions.

For each of the data points in the query performance graphs presented below, we generated a set of hundred random queries. Each query in a set has the same selectivity, but the starting values of the predicates are chosen randomly. We ran each of these thousand queries starting with a cold buffer pool, and then computed the average *per query* execution time. The graphs presented below show this average query execution time.

4.3 Effect of Number of Spatial Partitions

In the first experiment, we examine the effect of the number of spatial partitions on the search performance of SETI. Figure 5 shows these results for the GSTD(1K, 4M) data set and a 0.1% time-interval query.

For a small number of cells, less than 250 in Figure 5, each cell covers a large portion of the spatial extent. Consequently, many moving objects are mapped to each cell, which reduces the spatial discrimination of the index. As a result, the search process produces a large number of false positives in the filter steps, which leads to poor performance. As the number of cells increases, spatial discrimination increases, improving the performance of the index. However, as the number of cells increases, the probability that a trajectory segment will cross a spatial cell boundary also increases. This behavior leads to an increase in the replication in the index, which then starts to degrade the performance of the index. In addition, as the number of cells increases, the number of R-trees and the total space consumed by all the R-trees also increases, which leads to greater competition for buffer pool space. However, within the range of 200-1200, the degradation in performance is very gradual as the the number of cells is increased. The index performs best when the number of partitions is around 600 cells, and in the range 200-1200 cells, the performance of the search is within 20% of the best case.

We observed a similar behavior for other queries and other data sets, and in the interest of space we omit these additional results. For the remaining experiments presented in this paper, we keep the number of cells constant at 400 (20x20).

4.4 Index Sizes

Figure 6 shows the sizes of SETI, TB-tree and 3D R-tree indexes for the three GSTD data sets: GSTD(1K, 1M), GSTD(1K, 2M), and GSTD(1K, 4M). The three data sets shown in Figure 6 correspond to increasing number of seg-

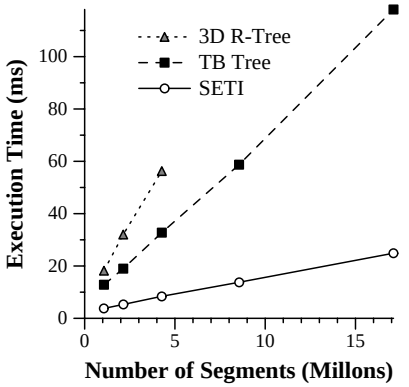

Figure 8: Scaling with Number of Segments, GSTD(1K, X), 0.01% Time-interval Query

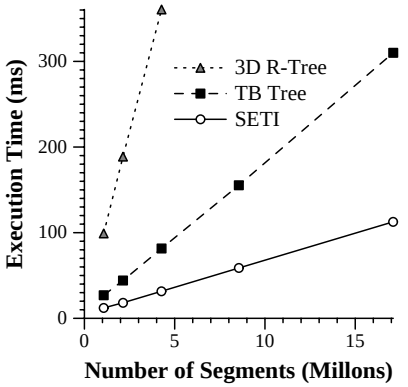

Figure 9: Scaling with Number of Segments, GSTD(1K, X), 0.1% Time-interval Query

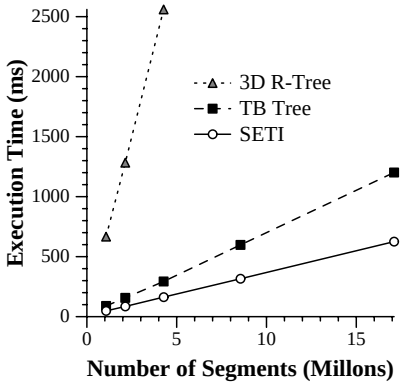

Figure 10: Scaling with Number of Segments, GSTD(1K, X), 1.0% Time-interval Query

ments per trajectory. The sizes shown in the figure only include the size of the index file. The size of the data file for the three data sets is 32MB, 64MB and 128MB respectively. For SETI, about 8% of the segments were split as they crossed some spatial partition boundaries, so the data file sizes are about 8% larger than for the other indices (as described in Section 3.2 the logical split points are stored in the data file).

As seen in Figure 6, SETI index sizes are smaller than the TB-tree because SETI uses sparse indices, and the R-trees in SETI only index on the time dimension. The 3D R-tree requires the most amount of space because average page occupancy is about 68%. The TB-tree has close to a 100% page occupancy. For SETI, the average page occupancy is between 55-65%, with the leaf pages having close to a 100% occupancy.

4.5 Insert Performance

In this section, we evaluate the performance of an index operation that appends new segments to existing trajectories. For this experiment we use the GSTD(1K, 4M) data set. After building the indices, we inserted 10K additional segments. Figure 7 shows the average time taken to insert a single trajectory segment over these 10K inserts.

As seen in Figure 7, inserts in SETI are nearly five times faster than the TB-tree and nineteen times faster than the 3D R-tree. For the TB-tree, the insert operation is executed in two steps. First, the last leaf node for the trajectory is located to check if there is space to accommodate the new trajectory segment in that leaf node. Then, the new trajectory segment is inserted either into the last leaf node, or in a newly created leaf node. Since the TB-tree index is not a sparse index, splits at the leaf nodes are more frequent than in SETI, leading to poor relative performance. Inserts into the 3D R-tree are costly because of the page splits. The key representation in the 3D R-tree is also larger than in SETI, leading to a smaller fanout and larger number of splits.

4.6 Time-interval Queries

In this section, we present the performance of the index structures when evaluating time-interval queries.

4.6.1 Varying Number of Segments

For this experiment we generated a number of GSTD data sets, each with 1K number of moving objects, and varied the total number of segments from 1M to 17M. Figures 8, 9 and 10 plot the performance of the indices for the query selectivities of 0.01%, 0.1% and 1%.

As shown in these figures, the 3D R-tree has the worst performance of the three indexing structures. The 3D representation of the segments in the 3D R-tree index results in large dead space in the keys used to represent the leaf nodes, and large amounts of overlap among the keys in the internal nodes. As a result, the search performance degrades as multiple paths have to be traversed to evaluate the query. The performance degradation is more rapid with larger query selectivities (observe the rapid performance degradation in Figure 10). The 3D R-tree also take a longer time to build the index for the same reasons. For data sets with more that 4M segments, the index build time was over six hours, and we aborted these tests. Since it is clear from this experiment that the 3D R-tree performs poorly, for the rest of the paper we do not consider the 3D R-trees.

Going back to Figures 8, 9 and 10, we observe that SETI also outperforms the TB-tree. One of the reasons for this behavior is that the R-trees in SETI are more efficient that the R-tree in the TB-tree. The R-trees in SETI only index on the time dimension, whereas the keys in the R-tree of the TB-tree are 3-dimensional. With higher dimensional keys, the dead space and overlap in the R-tree increases, which degrades the search performance. In addition, the sparse index of SETI is more I/O efficient.

SETI also has a significant advantage over the TB-tree with respect to the CPU cost. For the temporal indices that correspond to spatial cells that are fully contained in the

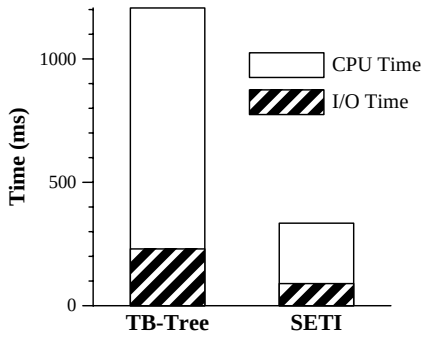

Figure 11: CPU and I/O components, GSTD(1K, 4M), 0.1% Time-interval Query

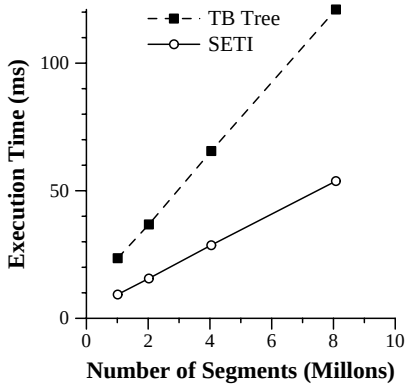

Figure 12: San Joaquin Dataset, NetSJ(1K, X), 0.1% Time-interval Query)

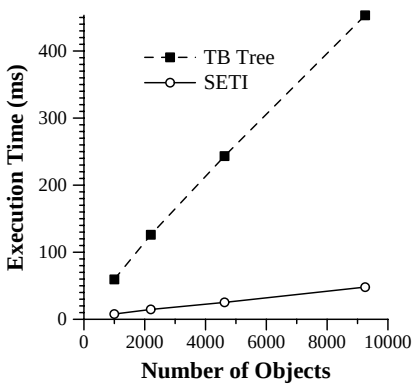

Figure 13: Long Update Intervals, NetSJ-CP(X, X×0.5M), 0.1% Time-interval Query

query box, only the temporal predicates need to be checked (refer to Section 3.3). CPU costs have been shown to constitute a big portion of spatial query evaluation when using 2D spatial objects [20], and not surprisingly, the CPU component is also very significant when working with 3D trajectory data sets.

The CPU and the I/O contributions for both SETI and the TB-tree are shown in Figure 11 for a sample 0.1% time-interval query on the GSTD(1K, 4M) data set. This figure shows that the CPU costs are much higher for the TB-tree. We observed a similar trend for other query sizes and data sets.

4.6.2 Network Data Set

Figure 12 shows the results for network based data set NetSJ(1K, X), where X is varied from 1M to 8M. For this experiment we use a 0.1% time-interval query. The results are similar for other queries, and in the interest of space additional results are omitted in this presentation.

The results follow the same trend as the GSTD data set. Unlike the GSTD data set where all the objects are moving with an uniform speed at all times, in this data set users vary speed and the update rate. The concentration of users also varies, as the road network is not distributed uniformly. There are slow moving objects in urban areas (where the road density is high) and fast moving objects on the freeways and the country side roads. This experiment demonstrates that SETI continues to be effective for non-uniformly distributed trajectory data sets too. The main reason for this counterintuitive result is that even though the index looses spatial discrimination (because of many objects being clustered in a spatial cell), the temporal discrimination increases. Since dense cells have many more objects than an average cell, they also generate many more updates. Each update produces a trajectory segment that is indexed by the temporal R-tree for that cell. With many more of these trajectory segments in a dense cell, the segments tend to be be better clustered in the time dimension. Consequently, the lifetimes of pages in dense pages is generally smaller, which leads to better temporal discrimination. Thus for dense cells, the decrease in performance due to larger R-trees is offset by the improvement in search performance caused by the increase in temporal discrimination.

Figure 13 shows the results for the data set simulating cell phone users moving in the Joaquin County road network. Recall from Section 4.2 that this data set has many segment with long time spans. This characteristic affects the performance of temporal indices by increasing the lifetimes of data pages in which this segment is stored. The result in Figure 13 shows that SETI temporal indices do not deteriorate with the introduction of long segments. For the time periods in which the objects are stationary, only one segment is inserted into the index for each object. The lifetimes of the pages that contain such segments with long time spans is also very long. However, there are many more pages with short time spans, and as a result, the degradation in the search performance is not severe. However, the performance of the TB-Tree deteriorates because the lower-level non-leaf nodes and the leaf nodes contain large minimum bounding box keys. As a result, the searches are slow, and a large number of false positives are produced, which leads to poor search performance.

4.6.3 Varying Number of Objects

We also examined the scalability of the index search performance when the number of moving objects is increased (in contrast to increasing the number of trajectory segments per moving object as was done in the experiments presented in Section 4.6.1). In this experiment, the number of mobile objects range from 10K to 160K, and the number of segments per trajectory is kept constant at 100, which results in the total number of segments increasing from 1M to 16M. In the interest of space, we only present the result for a 0.1% time-interval query, which is representative of the performance that we observed with other queries too.

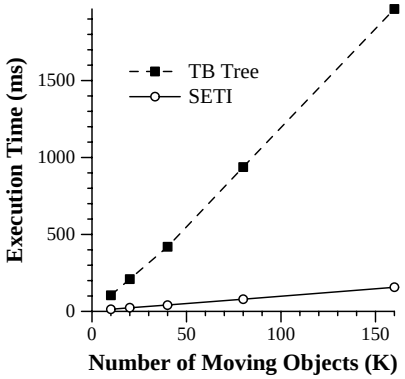

Figure 14: Scaling with Number of Objects, GSTD(X, $100 \times X$), 0.1% Time-interval Query

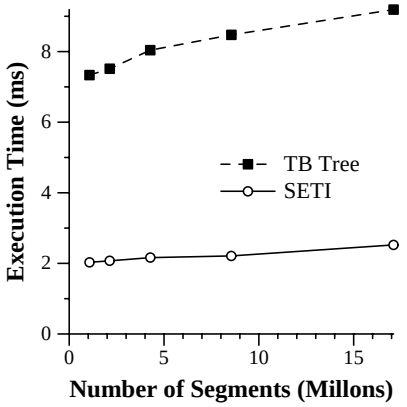

Figure 15: Scaling with Number of Segments, GSTD(1K, X), 1% Time-slice Query

This result is shown in Figures 14. This experiment demonstrates that SETI continues to outperform the TB-tree index when the data set has a large number of trajectories.

4.7 Time-slice Queries

In this section, we evaluate the performance of the index structures for evaluating time-slice queries. While we ran experiment with an number of time-slice queries with selectivities between 0.1% and 4%, in the interest of space we only present the results for a 1% time-slice query. The results for the other queries are similar.

Figures 15 plots the performance of SETI and the TB-tree for a 1% time-slice queries, for a number of different GSTD data sets. This figure show that SETI outperforms the TB-tree index even for the time-slice query, in which the temporal discrimination affects query performance more than the spatial discrimination. SETI performance advantages come from both I/O and CPU efficiencies. Across the data points in this experiment, the number of disk I/Os incurred by SETI is 45-80% lower than that for the TB-tree, and SETI's portion of the CPU costs are 53-80% lower than that for the TB-tree.

5 Conclusions and Future Work

In this paper we have proposed and evaluated a new trajectory indexing mechanism, called SETI. Unlike previously proposed trajectory indexing mechanisms, SETI decouples the indexing of the spatial and the time dimensions which leads to greater search and update efficiencies. Based on an implementation in SHORE, we have demonstrated that SETI outperforms a 3D R-tree and the TB-tree for both time-interval and time-slice queries. SETI is also capable of supporting faster rates for operations that append new segments to existing trajectories. SETI has the added advantage of being a logical indexing structure that can be easily built over existing spatial indexing structures, such as the R-tree. This property makes it much easier for any database developer to implement SETI.

As part of the future work, we plan on investigating the impact on performance of various adaptive partitioning strategies. We also plan on investigating the use of SETI for evaluating trajectory queries, which require fetching entire trajectories, and computing derived values such as average speed. The work presented in this paper has focused only on historical queries, and as part of our future work, we plan on investigating extensions to the front-line structure in SETI to answer queries on the future positions of moving objects.

Acknowledgements

We would like to thank Richard A. Hankins for providing valuable feedback on a earlier version of this paper. In addition, we would like to thank the anonymous reviewers for their comments; in particular, one of the anonymous reviewers provided very valuable detailed comments and suggestions that has helped improve the presentation and quality of this paper.

References

[1] AGARWAL, P. K., ARGE, L., AND ERICKSON, J. Indexing Moving Points. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Dallas, Texas, USA, May 2000), pp. 175–186.

[2] BAHL, P., AND PADMANABHAN, V. N. RADAR: An In-building RF-based User Location and Tracking System. In *INFOCOM (2)* (2000), pp. 775–784.

[3] BECKER, B., GSCHWIND, S., OHLER, T., SEEGER, B., AND WIDMAYER, P. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal 5*, 4 (1996), 264–275.

[4] BECKMANN, N., KRIEGEL, H. P., SCHNEIDER, R., AND SEEGER, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM-SIGMOD Conference* (Atlantic City, NJ, USA, June 1990), pp. 322–331.

[5] BERCHTOLD, S., BÖHM, C., AND KRIEGEL, H.-P. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM-SIGMOD Conference* (Seattle, WA, USA, June 1998), pp. 142–153.

[6] BRINKHOFF, T. Generating Network-Based Moving Objects. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management* (Berlin, Germany, July 2000), pp. 253–255.

[7] CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O., WHITE, S., AND ZWILLING, M. J. Shoring up Persistent Applications. In *Proceedings of the 1994 ACM-SIGMOD Conference* (Minneapolis, Minnesota, USA, May 1994), pp. 383–394.

[8] *FCC Wireless 911 Requirements*. Federal Communications Commission, 2001.

[9] FINKEL, R., AND BENTLEY, J. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica, Springer Verlag 4* (1974), 1–9.

[10] GETTING, I. A. The Global Positioning System. *IEEE Spectrum 30*, 12 (Dec. 1993), 36–47.

[11] GÜTING, R. H., BÖHLEN, M. H., ERWIG, M., JENSEN, C. S., LORENTZOS, N. A., SCHNEIDER, M., AND VAZIRGIANNIS, M. A Foundation for Representing and Quering Moving Objects. *ACM Transactions on Database Systems 25*, 1 (Mar. 2000), 1–42.

[12] GUTTMAN, A. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM-SIGMOD Conference* (Boston, MA, June 1984), pp. 47–57.

[13] HARTER, A., HOPPER, A., STEGGLES, P., WARD, A., AND WEBSTER, P. The Anatomy of a Context-Aware Application. In *Mobile Computing and Networking* (Seattle, WA, USA, August 1999), pp. 59–68.

[14] KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. J. On Indexing Mobile Objects. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, USA, May 1999), pp. 261–272.

[15] KORNACKER, M., AND BANKS, D. High-Concurrency Locking in R-trees. In *Proceedings of the 21st VLDB Conf.* (Zurich, Switzerland, September 1995), pp. 134–145.

[16] KORNACKER, M., MOHAN, C., AND HELLERSTEIN, J. M. Concurrency and Recovery in Generalized Search Trees. In *Proceedings of the 1997 ACM-SIGMOD Conference* (Tucson, Arizona, USA, May 1997), pp. 62–72.

[17] NASCIMENTO, M., AND SILVA, J. Towards Historical R-trees. In *In Proceedings of ACM Symposium on Applied Computing (ACM-SAC)* (1998), pp. 235–240.

[18] NASCIMENTO, M. A., SILVA, J. R. O., AND THEODORIDIS, Y. Evaluation of Access Structures for Discretely Moving Points. In *Spatio-Temporal Database Management* (Edinburgh, Scotland, September 1999), pp. 171–188.

[19] PAPADIAS, D., TAO, Y., KALNIS, P., AND ZHANG, J. Indexing Spatio-temporal Data Warehouses. In *In Proceedings of International Conference on Data Engineering* (San Jose, CA, USA, February 2002), pp. 166–175.

[20] PATEL, J. M., AND DEWITT, D. J. Partition Based Spatial–Merge Join. In *Proceedings of the 1996 ACM-SIGMOD Conference* (Montreal, Quebec, Canada, June 1996), pp. 259–270.

[21] PFOSER, D., JENSEN, C. S., AND THEODORIDIS, Y. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proceedings of the 26st VLDB Conf.* (Cairo, Egypt, September 2000), pp. 395–406.

[22] PITOURA, E., AND SAMARAS, G. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering 13*, 4 (2001), 571–592.

[23] PRIYANTHA, N. B., CHAKRABORTY, A., AND BALAKRISHNAN, H. The Cricket Location-support System. In *Mobile Computing and Networking* (Boston, MA, USA, August 2000), pp. 32–43.

[24] SALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LOPEZ, M. A. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM-SIGMOD Conference* (Dallas, Texas, USA, May 2000), pp. 331–342.

[25] SISTLA, A. P., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. Modeling and Querying Moving Objects. In *Proc. of the 13th International Conference on Data Engineering* (Birmingham U.K, April 1997), pp. 422–432.

[26] SONG, Z., AND ROUSSOPOULOS, N. Hashing Moving Objects. In *Mobile Data Management* (Hong Kong Polytechnic University, Hong Kong, January 2001), pp. 161–172.

[27] TAO, Y., AND PAPADIAS, D. MV3R-Tree: A Spatio-temporal Access Method for Timestamp and Interval Queries. In *The VLDB Journal* (2001), pp. 431–440.

[28] THEODORIDIS, Y., SILVA, J. R. O., AND NASCIMENTO, M. A. On the Generation of Spatiotemporal Datasets. In *Advances in Spatial Databases, 6th International Symposium* (1999), Lecture Notes in Computer Science, Springer, pp. 147–164.

[29] THEODORIDIS, Y., VAZIRGIANNIS, M., AND SELLIS, T. K. Spatio-Temporal Indexing for Large Multimedia Applications. In *International Conference on Multimedia Computing and Systems* (1996), pp. 441–448.

[30] U. S. BUREAU OF THE CENSUS. *TIGER/Line Files(TM), 1992 Technical Documentation*.

[31] WARD, A., JONES, A., AND HOPPER, A. A New Location Technique for the Active Office. *IEEE Personnel Communications 4*, 5 (Oct. 1997), 42–47.

[32] XU, X., HAN, J., AND LU, W. RT-tree: An Improved R-tree Index Structure for Spatio-temporal Databases. In *Proceedings of the 4th International Symposium on Spatial Data Handling* (1990), pp. 1040–1049.