Recursive Partitioning Method for Trajectory Indexing

Elizabeth Antoine, Kotagiri Ramamohanarao, Jie Shao and Rui Zhang

Department of Computer Science and Software Engineering
The University of Melbourne, Victoria
Australia
Email: {eantoine, rao, jsh, rui}@csse.unimelb.edu.au

Abstract

A trajectory is defined as the record of time-varying spatial phenomenon. The trajectory database is an important research area that has received a lot of interest in the last decade, with the objective of trajectory databases being to extend existing database technology to support the representation and querying of moving objects and their trajectories. Querying in trajectory databases can be very expensive due to the nature of the data and the complexity of the query processing algorithms. Given also that location-aware devices, like the GPS, are present everywhere these days, trajectory databases will soon face an enormous amount of data. Consequently the performance in the presence of a vast amount of data will be a significant problem and efficient indexing schemes are required to support both updates and searches efficiently.

This paper provides the methodology for using the recursive partitioning technique for indexing trajectories in the unrestricted space, which is called the Recursively Partitioned Trajectory Index (RPTI). RPTI uses the two-level indexing structure, as does the state of art indexing scheme, SETI, and maintains separate indices for the space and time dimensions. We present the algorithms for constructing the RPTI and the algorithms for updates that include insertion and deletion. We also provide the results of the experimental study on the RPTI and have demonstrated that RPTI is better than SETI in handling trajectory-based queries and is competitive with SETI in handling coordinate-based queries. The structure of RPTI can be easily implemented by using any of the existing spatial indexing structures. The only design parameters required are the standard disk page size and maximum level of recursive partitioning.

Keywords: Indexing Trajectories, Recursive Paritioning Method, Trajectory and Coordinate-based Queries

1 Introduction

Location-aware devices such as the GPS, which is the sequel to satellite and atomic clock technologies, have led to a number of new applications, such as fleet management and location-based solutions. Mobile phones that are E911-enabled are used to locate the mobile phone user with a variation of a few hundred meters, which is helpful in providing location information during emergencies. In addition to outdoor location devices like the GPS, there are devices for determining the location indoors. The 3D ultrasonic location system [Ward and Jones, 1997], which is low-power and wireless, is one such system, and has led to new applications such as context-aware applications that respond to the user as he/she moves around.

Since the location-aware devices are used extensively these days, trajectory (or moving object) databases will soon be faced with the task of managing an enormous amount of data. The performance of the trajectory databases will significantly decline in the presence of a vast amount of data as the sequential access is time-consuming. Indexing structures are efficient in handling large data sets in various application domains, as they allow random look-ups; hence, indexing structures would be efficient in handling trajectory data as well. Traditional indexing structures like B-trees [Comer, 1979] are not efficient in ordering the multidimensional data and therefore cannot be used in spatial and trajectory databases.

Extensive work has been done on spatial indexing and R-tree variations have been found to be efficient in query processing [Guttman, 1984, Bentley and Friedman, 1979, Sellis et al., 1987, Beckmann et al., 1990, Kamel and Faloutsos, 1994]. In the domain of trajectory indexing, R-tree variations and extensions, such as, the three dimensional R-tree [Theodoridis et al., 1996], the TB tree [Pfoser et al., 2000], the STR tree [Pfoser et al., 2000] and the SETI tree [Chakka et al., 2003] have been introduced for indexing past locations of moving objects. In this paper we examine one such indexing structure that decouples the indexing of the space dimension from the time dimension and uses the recursive partitioning method for indexing in the space dimension. Similar to the SETI structure, the RPTI uses the R^*-tree to index the time intervals.

The sequel of the paper is organized as follows. Section 2 describes the related work, data model and queries. Section 3 explains the RPTI structure. Section 4 provides the algorithms for the RPTI structure. Section 5 details the experimental study of the RPTI structure. Section 6 gives the conclusion.

2 Related Work, Data Model and Queries

2.1 Related Work

In this section, we will be discussing some of the indexing structures that record past locations of the moving objects and those that organize motion in an unrestricted space. R-tree and its variations are found to be efficient in handling multidimensional data and have found their way into the commercial database systems. Hence, we will be discussing some of the R-tree variations and extensions that are used in indexing trajectories. We will be discussing the three

dimensional R-tree [Theodoridis et al., 1996] and the TB trees [Pfoser et al., 2000] which are efficient in handling coordinate-based queries. We will also discuss the SETI [Chakka et al., 2003] structure, which partitions the space into hexagons and then indexes the time dimension for every hexagon. This is useful in analysing the recursive partitioning of space in indexing trajectories.

The three dimensional R-tree [Theodoridis et al., 1996] are based on the R-tree index, which is widely used for indexing of spatial data. It is based on the previous efforts of the authors with regard to multimedia application modelling. The proposed indexing schemes was the first step towards a new direction of spatio-temporal indexing, while research has mainly focused on content-based image indexing - i.e., fast retrieval of objects using their content characteristics (colour, texture, shape). A three dimensional R-tree is a straightforward method of indexing trajectories. This method extends the R-tree to be used in a three dimensional space, with time as an extra spatial dimension (2D space + 1D time). Three dimensional R-trees are designed to handle coordinate-based queries, like the time-slice and time-interval queries, and is inefficient in answering trajectory-based queries. Based on the experimental study in [Chakka et al., 2003], SETI is shown to outperform the three dimensional R-tree. SETI partitions the space into static non-overlapping partitions and, for each partition, it uses the R^*-tree to build a sparse index over the time dimension [Chakka et al., 2003].

The TB-tree, introduced in [Pfoser et al., 2000], is fundamentally different from the access method they previously discussed (STR-tree [Pfoser et al., 2000]). The STR-tree introduces a new insertion/split strategy to achieve trajectory orientation, while not compromising the space discrimination capabilities of the index too greatly. Apart from this, the STR-tree is an R-tree-based access method. An underlying assumption when using the R-tree is that all inserted geometries are considered as parts of trajectories and the segments are stored independently in the R-tree and the STR-tree structures.

The TB-tree, provides an access method that strictly preserves trajectories, such that a leaf node only contains segments belonging to the same trajectory, thus the index is best understood as a trajectory bundle. As a drawback, line segments that are not part of the same trajectory but lie spatially close will be stored in different nodes. As the overlap increases, the space discrimination decreases and, thus, the classical range query cost increases. However, by giving up on space discrimination, there is a gain in trajectory preservation [Pfoser et al., 2000]. The structure of the TB-tree is actually a set of leaf nodes, each containing a partial trajectory, connected by a doubly linked list that preserves trajectory evolution. By visiting an arbitrary leaf node, these links allow us to retrieve the whole or part of the trajectory at minimal cost.

In spite of its clear advantages on trajectory-based query processing, the TB tree has a crucial drawback in its insertion strategy, as new trajectory data are always inserted to the right end of the tree, leading its performance to depend on the order of the data insertion. When an object enters an area where the position transmission system does not work, its information is locally stored and later transmitted to the central server, while other moving objects would have transmitted their positions. This violates the assumption that the trajectory data are inserted in the index in a purely chronological order [Manolopoulos et al., 2005]. Deletions are often ignored in the trajectory indexing structures and this is true with the TB trees as well. However, trajectories of objects that are no longer useful need to be deleted and this leaves holes in the TB-tree structure.

The Scalable and Efficient Trajectory Index (SETI) [Chakka et al., 2003] 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 and its variants, like the R^*-tree. 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. As the extent of the spatial dimensions does not change, an indexing structure could partition the spatial dimensions statically however, the number of partitions need to calculated prior to the processing. Within each spatial partition, the indexing structure only needs to index lines in a 1D (time) dimension. Consequently, such an approach will not exhibit the rapid degradation in index performance that is generally observed for 3D indexing techniques.

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 [Chakka et al., 2003]. 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: there are smaller index overheads and an improved insertion performance [Chakka et al., 2003]. The temporal indices also provide the temporal discrimination in searches.

There are several advantages for this technique and some are given below. As the objects that are actually indexed are one-dimensional time lines, the indexing structure does not suffer from the curse of dimensionality [Berchtold et al., 1998], which causes the performance of indexing structures to degrade rapidly as the number of dimensions increases. Updates to trajectories tend to 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, the last position of that object in the front-line structure is looked up and adds the trajectory segment to the SETI index. As only a sparse R^*-tree index is used 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. SETI can be viewed as a logical indexing structure that can be built on top of existing spatial indexing techniques, such as an R-tree. Consequently, implementing SETI is much easier than implementing a new physical indexing structure. 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. We provide the data model and discuss on the query processing in the following sections.

2.2 Data Model

When an object, in this example - a car, moves around space; the successive locations of the car are recorded

at specific time instances using the location-aware devices. The locations of the object are connected to form a sequence of line segments using the interpolation methods. The set of line segments (or polyline) represents the trajectory of the moving point object and is illustrated in Figure 1.

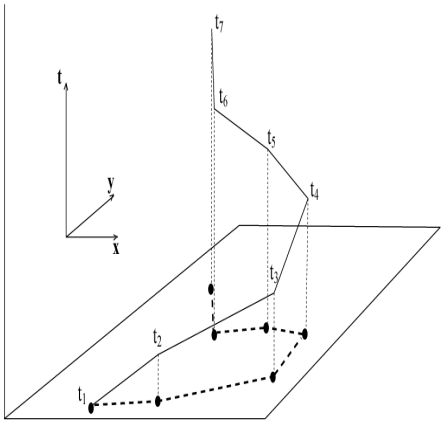

FIGURE 1: The Locations of the Spatial Object and the Corresponding Trajectory [Pfoser et al., 2000]

The motion of an object is approximated as the series of line segments where each line segment connects two consecutive update positions of the object [Güting et al., 2000, Kollios et al., 1999, Papadias et al., 2000, Pfoser et al., 2000, Pitoura and Samaras, 2001, Saltenis et al., 2000]. Each line segment 's' is represented by its segment id 'sid' and the connected line segments of an object are called its trajectory, and are identified by its unique trajectory id 'tid'. The representation of a trajectory (trj) is stated formally in [Chakka et al., 2003].

A trajectory is represented as:

$$trj\,(\,tid,\,(\,u_0,\,u_1\,,u_2\,,\ldots,\,u_n\,,\,\ldots)\,)$$

$(u_0, u_1, u_2, \ldots, u_n, \ldots)$ is a sequence of points representing the positions of the moving object. Each point u_i is given by (x_i, y_i, t_i) where (x_i, y_i) represents the position of the moving object in space recorded at the time t_i. $(u_0, u_1, u_2, \ldots, u_n, \ldots)$ are sequenced in the increasing order of time $(u_i < u_{i+1})$.

The trajectory segment of an object is represented by:

$$s_i\,(\,tid,\,sid,\,u_{i-1},\,u_i\,)$$

(u_{i-1}, u_i) are the two update positions of the object and s_i represents the line segment that connects the two update positions. The insertion procedure of the RPTI structure has u_i as its input parameter and the procedure constructs the line segment based on the previous location u_{i-1} and inserts the new line segment s_i, as represented above with some additional variables. These will be discussed in the following sections. Given the above formal definition of trajectories, we provide a brief note on query types in the next section.

2.3 Queries

Querying in trajectory databases can be very expensive due to the nature of the data and the complexity of the query processing algorithms. Given also that location-aware devices, like the GPS, are ubiquitous these days, trajectory databases will soon face an enormous amount of data. Consequently the performance in the presence of a vast amount of data will be a significant problem. Queries on moving objects can be broadly classified into two categories: queries that ask questions about the future positions of moving objects; and queries that ask questions about the historical positions of moving objects [Pfoser et al., 2000, Theodoridis et al., 1996, Frentzos, 2003, De Almeida and Güting, 2005, Chakka et al., 2003]. We focus on the queries relating to the historical positions of moving objects.

The historical queries are further classified into coordinate-based queries and trajectory-based queries. Coordinate-based queries include the time-slice queries, which select all the objects that lie within a given area at a given time instant; time-interval queries, which include the objects that lie within a given area and given time period; and nearest neighbour queries. The trajectory-based queries are further classified into topological queries and navigational queries.

Topological queries examine the whole or part of the trajectory of an object. For example, the predicate 'collision' examines two trajectories if they intersect in a given area at a specific time instant. The navigational query involves the information derived from the trajectories, such as the speed and the heading of an object. The average or top speed of an object is obtained by the fraction of travelled distance over time. The heading or the direction of travel of the object is computed by determining the vector between two specified positions [Pfoser et al., 2000]. The following are examples of the coordinate and trajectory-based queries; and their respective SQL statements [Erwig and Schneider, 2002].

Coordinate-based queries:

Time-interval Query: "find all objects that are present within a given area during a given time interval" (range query in space and time dimension). Example for time-interval queries is given below.
'Id' represents the unique identifier of the cars. 'Line' is the additional information that is stored about the travel; 'trajectory' as already mentioned, represents, the moving path of the cars. The above query represents the query window at the time dimension. It extracts the information about all the cars that were present at the time period 'P', without any clause in the space dimension.

Q1: Where exactly were the cars during the time period P?

```
SELECT Id, Line, trajectory AS Stretch
FROM Cars WHERE Trip present P;
```

Time-slice Query: "find all objects that are present within a given area during a given time instant" (range query in space with a zero extent in time dimension). Example for time-slice queries is given below.

'Loc' represents a point (x, y) in the 2D space and time instant 'I' represents the point (t) at the time dimension. It extracts the 'Id' of all the cars present at a particular location (Loc) in the space dimension and at a particular time instant (I) in the time dimension.

Q2: Give the IDs of all the cars present in the location X at time instant I?

```
SELECT Id from Cars C WHERE Trip
passes Loc AND Trip present I;
```

Trajectory-based queries:
"find the average speed of a moving object given its trajectory id". Example for trajectory query is given below.

'Car' is a relation and 'h' represents the car that has the Id, 'C'. The following query retrieves the whole trajectory information for the car 'C' and calculates the average speed of the car given by '*average_speed*' and 'h.route' represents the trajectory Id.

Q1: What is the car's (C) average speed?

SELECT h.route, *average_speed* FROM Car h WHERE h.id = C;

In order to answer the queries efficiently, the data needs to be represented appropriately. Due to the tremendous increase in the amount of data that are stored in the database system, the sequential access becomes time-consuming and the need for an efficient storage and access mechanism arises. Indexes provide the basis for rapid random look-ups and, for the indexing to be worthwhile, the process of creating the index must not be time-consuming, otherwise the queries could be answered more efficiently without an index. While dealing with the proximity queries in spatial data, the database management system needs to sort the spatial data efficiently, based on the space occupied by the data, to enable random loop-ups. Such techniques are known as spatial indexing methods [Samet, 1995]. The same is true for trajectory queries.

3 Recursively Partitioned Trajectory Index (RPTI)

In trajectory data sets that represent the past locations of moving objects, the extent of the spatial dimensions change very slowly over the lifetime of the trajectory data set growth, whereas the time dimension is constantly increasing. Similar to the SETI structure [Chakka et al., 2003], the RPTI indexing structure partitions the space by exploiting the static space dimension. Instead of partitioning the space into non-overlapping spatial cells, space is partitioned hierarchically. The number of levels is denoted by L and at every level $l (l = 0, \ldots, L-1)$, the number of partitions is 4^l and each level is composed of $2^l - 1$ equally spaced lines in each dimension. A line segment intersected by any partitioning line of level l belongs to level $l-1$.

At each level, for every partition an indexing structure like the R^*-tree is used to index lines in the time dimension. The number of R^*-trees at every level is given by 4^l. Similar to the SETI structure, the RPTI indexing mechanism achieves good spatial and temporal discrimination. Discrimination represents the ability to identify the candidate set of index entries with few false hits. The Recursively Partitioned Trajectory Index (RPTI) structure is illustrated in Figure 2. Each segment of the trajectory is stored as a tuple in the level files. The insertion and deletion strategies are explained in detail in the following section. For each level file, a space-filling curve, such as the z-ordering curve [Orenstein and Merrett, 1984], is used to calculate the z-order value for each line segment. The line segment is stored as a tuple in its corresponding level file, with the restriction that any data page will only contain line segments that belong to the same partition.

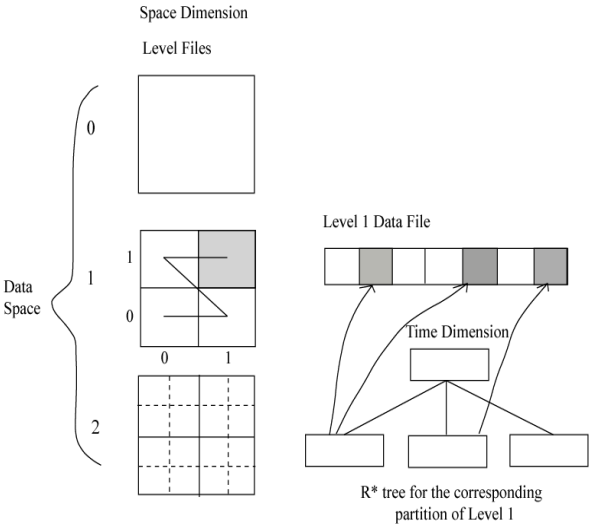

FIGURE 2: Recursively Partitioned Trajectory Index (RPTI) Structure

For every partition, an R^*-tree is used to index the $life-time$ of every data page. The $life-time$ of every page covers the time-spans of all the line segments stored in that page. Any spatial indexing structure other than R^*-tree can be used for this purpose. As mentioned in the SETI structure [Chakka et al., 2003], the R^*-tree index is sparse as there is only one entry for each data page [Chakka et al., 2003]. This makes the searching efficient but, during insertions and deletions, the life time of the data page might change. If the life time of a data page changes, then the corresponding R^*-tree should be updated.

4 Algorithms for RPTI Structure

This section explains in detail the insertion, deletion and search process for the RPTI structure and presents the relevant algorithms.

4.1 Insertion

RPTI maintains a $front-line$ structure similar to the one presented in SETI, which has the last update location of every moving object. The last update position of all the trajectories is maintained in a hash structure that is indexed by the trajectory id '(tid)', which is unique for every moving object.

When there is an update in the position of the moving object, say '*obj*', the hash structure is used to get the information of the last updated position based on its '*tid*'. A line segment is constructed connecting the last update position and the new position, and this is inserted into the RPTI structure. The level of the line segment '*Level*', with its z-order value '*zvalue*', is computed and the new line segment is inserted into the corresponding level file. The line segment tuple inserted into the level files is of the form:

$$s_i\,(\,tid,\ sid,\ u_{i-1},\ u_i,\ zvalue\,)$$

The SETI structure results in line segments intersecting spatial cell boundaries and is handled by splitting the line segments in the space and time dimensions. However, splitting the line segments leads to additional computation during updates and searches, where the split line segments have to be merged. The number of partitions is an important factor in any partitioning strategy that affects the area covered by the partitions. If the partitions cover large areas, then the space discrimination of the index is reduced. This

can, however, be tolerated when the number of line segments that fall inside the partitions is small. If the partitioning is done very finely, then the number of segments that cross the boundary of the spatial cells increases, which increases the computational time [Chakka et al., 2003] for SETI.

RPTI structure avoids the splitting of line segments by inserting the line segments at their appropriate level, and as is illustrated in Figure 3. The line segments of a given trajectory may be found in more than one level. An in-memory structure called $track - trajectory$ is maintained to hold the information about every trajectory. The tuple inserted to the $track - trajectory$ file is of the form (tid, $Level$, $zvalue$). When a new line segment is inserted to the RPTI structure, the $track - trajectory$ is consulted; no updates occur if the level file and z-order value for the trajectory matches with the level file and z-order value of the new line segment, otherwise a new tuple with a trajectory id, level file and z-order value is inserted into the $track - trajectory$. The updates also occur if the 'tid' is not found in the $track - trajectory$ file. More than one entry for a trajectory denotes the distribution of the line segments of the corresponding trajectory over the level files.

By doing this, the whole trajectory can be retrieved by consulting the $track - trajectory$ file and the problem of a line segment spanning over multiple partitions is eliminated; a problem which is prevalent in the SETI structure. Such a $track - trajectory$ structure can also be maintained for SETI but the splitting of line segments cannot be avoided, which increases the computational time for handling trajectory-based queries that require the whole or part of the trajectory. As mentioned in [Chakka et al., 2003], about 8% of the line segments cross the partition boundaries and this increases not only the index size but also leads to additional computation, which is eliminated in RPTI.

The updates to the corresponding R^*-tree do not happen frequently. It is only updated when the inserted line segment tuple falls in a new data page. As already mentioned, R^*-tree is used to index only the $life - time$ of data pages and, hence, there are infrequent updates to the R^*-trees. However, if the insertion changes the $life - time$ of the data page, the corresponding index entry in the R^*-tree is updated. Algorithm 1 illustrates the insertion process.

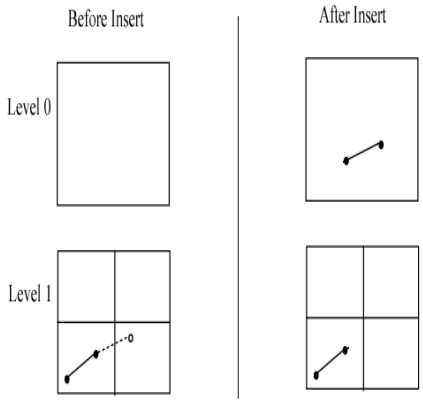

FIGURE 3: Insertion in RPTI

Algorithm 1: InsertSegment

input: New position 'u_i' of object 'obj',
Trajectory id 'tid' of object 'obj'
begin
Consult $front - line$ and retrieve last position 'u_{i-1}' using 'tid';
if *no entries found for 'tid'* **then**
Insert the tuple (tid, u_i) in $front - line$;
exit;
end
Insert the tuple (tid, u_i) in $front - line$;
Construct new segment 's_i' connecting last and new positions.;
Compute '$Level$' and '$zvalue$' for s_i ;
`/* (see Section 3) */`
$old - lifetime \leftarrow$ life time of data page where s_i is to be inserted ; `/* (see Section 3) */`
Insert tuple 's_i' (tid, sid, u_{i-1}, u_i, $zvalue$) to level file '$Level$';
$new - lifetime \leftarrow$ life time of data page after insertion ; `/* (see Section 3) */`
Consult $track - trajectory$ and retrieve the tuple for trajectory id, 'tid';
if *no entries found for 'tid'* **then**
Insert tuple (tid, $level$, $zvalue$) ;
else
for *each entry that represent 'tid'* **do**
if *level in each entry* $\neq$ *Level AND z-order in each entry* $\neq$ *zvalue* **then**
Insert tuple (tid, $level$, $zvalue$) ;
end
end
end
if $old - lifetime \neq new - lifetime$ **then**
Consult the corresponding R^*-tree based on '$Level$' and '$zvalue$';
if *no* R^**-tree found* **then**
Construct R^*-tree with index entry ($new - lifetime$, $ptr\_to\_the\_datapage$)
else
Retrieve index entry $old - lifetime$ and update with $new - lifetime$;
end
end
end

4.2 Deletion

Deletion of a trajectory can be efficiently done using the RPTI structure. Deletion is often ignored while proposing a trajectory index because of the assumption that deleting the trajectory of a moving object is meaningless after the transmitted positions are recorded. However, deletions are necessary when the trajectory of a moving object is no longer useful. Deletions include deleting a particular segment of a trajectory or deleting the whole trajectory of a moving object.

A list of trajectory Ids with their associated level file '$Level$' and '$zvalue$' is maintained in the $track - trajectory$ file. Given the trajectory id and the segment id, deleting a particular segment or deleting the entire trajectory becomes straightforward. The deletion process is illustrated in Algorithm 2. Deleting trajectory segments from a data page changes the $life - time$ of the data page and hence the respective R^*-tree indices need to be updated if the deleted segment tuple changes the $life - time$ of the data page. The worst case scenario for deletion would re-

quire reading one disk page for each trajectory line segment when retrieving a single trajectory. This is due to the fact that the trajectory line segments are organized based on the spatial and temporal relations, which results in the trajectory line segments belonging to a trajectory to be placed in different disk pages. This is similar to the SETI structure; however, the computational time for deletion in RPTI is reduced by avoiding the splitting of line segments across partition boundaries.

Algorithm 2: DeleteSegment

input: Trajectory id '*tid*' of object '*obj*'
begin
  Consult $track - trajectory$ and retrieve the tuple for the trajectory id, '*tid*';
  Retrieve the data page that holds the segment tuple of trajectory '*tid*' based on the *Level* and *zvalue*;
  $old - lifetime \leftarrow$ life time of retrieved data page;
  Delete all the segment tuple of '*tid*' or a particular segment of '*tid*' if '*sid*' is given;
end
$new - lifetime \leftarrow$ life time of data page after deletion;
if $old - lifetime \neq new - lifetime$ **then**
  Consult the corresponding R^*-tree based on '*Level*' and '*zvalue*';
  Retrieve index entry $old - lifetime$ and update with $new - lifetime$;
end

4.3 Search

The search is executed in the following steps for the coordinate-based queries.
Step 1: In the space dimension, the partitions that intersect the query window are found for every level.
Step 2: At each level, for every partition that intersects, the temporal index is searched based on the temporal predicates and the relevant data pages are retrieved. As mentioned earlier, the life-time of every data page is maintained in the R^*-tree.
Step 3: If the data pages that are retrieved belong to the partitions that are completely inside the query window, then all the trajectory id of the segments that belong to the page are returned. Data pages retrieved might have more than one segment belonging to a trajectory, in which case the trajectory id is returned only once. However, if the segment id is needed, it can also be listed. When the data pages belong to the partitions that are not completely inside the query window, the spatial predicate is applied to each segment tuple in the data page.

The search is done as illustrated in the steps above and it produces the list of segments or trajectories that overlap with the query window. The search process of RPTI is quite similar to that of the SETI structure. However, the duplicate elimination that occurs because of the splitting of line segments that cross cell boundaries is not found in RPTI.

The search process for the trajectory-based queries, where the whole or part of the trajectory is retrieved, is straightforward and is illustrated in Algorithm 3.

Algorithm 3: SearchTrajectory

input: Trajectory id '*tid*' of object '*obj*'
begin
  Consult $track - trajectory$ and retrieve the tuples for the trajectory id, '*tid*';
  Retrieve the data page that holds the segment tuples of trajectory '*tid*' based on the *Level* and *zvalue*;
  List all the segment tuples of '*tid*' or a particular segment of '*tid*' if '*sid*' is given;
end

Similar to the case of deletion, the worst case scenario for trajectory-based queries would require reading one disk page for each trajectory line segment when retrieving a single trajectory. This is due to the fact that the trajectory line segments are organized based on spatial and temporal relations, which results in the trajectory line segments belonging to a trajectory needing to be placed in different disk pages. This is similar to the SETI structure; however, the computational time for trajectory-based queries in RPTI is reduced by avoiding the splitting of line segments across partition boundaries. The following section provides the experimental results for the RPTI structure for both the coordinate and trajectory-based queries.

5 Experiments and Results for RPTI

In order to assess the performance benefits of RPTI, we compare it with our prototype implementation of the SETI structure. The SETI structure performs better than the three dimensional R-tree and the TB tree for both the time-slice and time-interval queries as described in [Chakka et al., 2003]. In the following, we present the experimental results for the time-interval queries and trajectory queries. The experiments were conducted on an Intel®Core™2 Duo Processor at 2.00GHz that has the main memory capacity of 2.0GB. The Windows experience rating index for the processor is 4.7 out of 5.

Synthetic data sets are generated using the GSTD generator of spatiotemporal datasets [Theodoridis et al., 1999] to create trajectories of moving objects. For a specified number of moving objects, the GSTD data generator produces specified number of segments per moving object. The number of trajectories for the data sets is kept as 1K and for every trajectory the number of segments is varied to form 5 GSTD data sets. The total number of segment tuples for 5 GSTD data sets are 4M, 8M, 12M, 16M and 20M respectively and are represented as GSTD-4M, GSTD-8M, GSTD-12M, GSTD-16M and GSTD-20M. The size of the level files for the 5 GSTD data sets is 144MB, 288MB, 432MB, 576MB and 720MB respectively.

The number of partitions is restricted for the SETI structure, as the splitting of line segments that cross the spatial cell boundary degrades the performance of the index. The recursive partitioning method in the RPTI structure eliminates the problem of splitting line segments by moving the line segments that cross the boundary to a higher level. The RPTI method performs better when the spatial dimension is partitioned very finely, as the partitioning forces the line segments that cross the boundary to be moved to the higher level and this result in a uniform distribution of line segments at the lower levels. This is illustrated in Figure 4. For the purpose of understanding the distribution of line segments over level files, we have plotted the average number of line segments per partition for the data set of 1M line segments (number

of trajectories and the number of line segments per trajectory is kept as 1K).

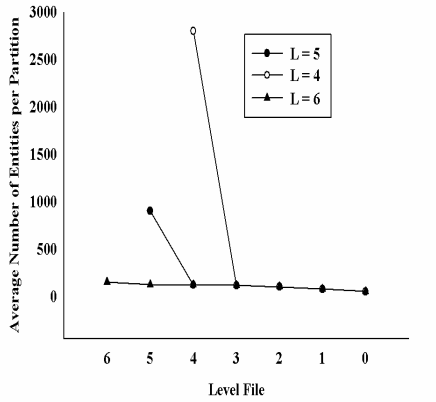

FIGURE 4: Distribution of Line Segments across Level Files for Varying L (Level until the Recursive Partitioning is done)

5.1 Performance of Insertion

The experimental results for the insertion performance of the GSTD data set with 4M segments (GSTD-4M) is given in Figure 5. For the SETI structure, the insertion of line segments involves the splitting of line segments if they cross spatial cells. But for the RPTI structure, the insertion is straightforward and it involves the update of the $track - trajectory$ file if the trajectory id is not found in the file or if the line segment falls in a new level or a new partition.

After the level files are generated and the time indices are constructed, we present the performance of inserting 1K segments as an average time for inserting a single line segment. The result demonstrates that the RPTI structure outperforms the SETI for insertion, and that the reason for the increase in average time for the SETI is due to the splitting process, which is eliminated in our recursive partitioning method.

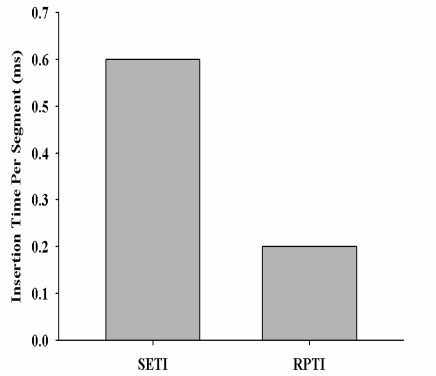

FIGURE 5: Insertion Performance

5.2 Time-interval Queries

The performance of RPTI for the time-interval queries tends to be competitive with the SETI structure. The slight increase in computation time is because of the recursive partitioning of space in RPTI; each level file needs to be checked for the relevant partitions. The number of partitions to be examined for false positives is higher as the partitions that are not completely enclosed by the query window tend to be found in almost every level. The number of segments per trajectory is varied and experiments were conducted on the GSTD data set with 1K trajectories. The query window in the experiments was chosen to have 0.1% and 1.0% query selectivity. The total number of line segments in the data sets is 4M, 8M, 12M, 16M and 20M respectively. The results show that the RPTI structure is competitive with the SETI for the time-interval queries. Figures 6 and 7 illustrate the results of the experiments by varying the number of line segments per trajectory and the query window size.

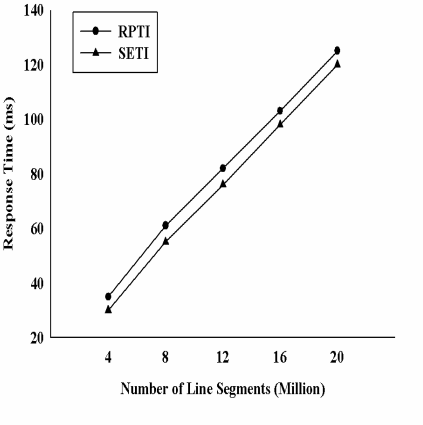

FIGURE 6: Time-interval Query Performance for 0.1% Query Selectivity

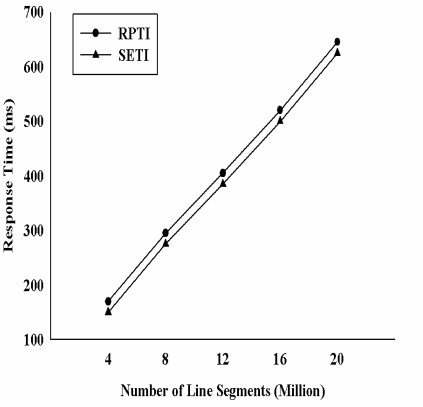

FIGURE 7: Time-interval Query Performance for 1.0% Query Selectivity

If the query window in the space dimension is large, then almost every partition at the higher levels intersects with the query window and this increases the number of false positives. The duplicate elimination in the SETI structure results in further computation and this is eliminated in RPTI. Though the number of partitions for checking false positives is higher in RPTI, the increase in computational time is compensated by the elimination of duplicates.

5.3 Time-slice Queries

In this section, we provide the experimental results of the time-slice query for the SETI and RPTI structures. Similar to the results of the time-interval query, the increase in response time for RPTI is due to the query window being executed at every level. However, the results show that the RPTI structure is competitive with the SETI. As already mentioned, the time-slice query executes the range query in the space dimension with a zero extent in the time dimension.

Figure 8 illustrates the experimental results for the time-slice queries. The number of segments per trajectory is varied and experiments were conducted on GSTD data set with 1K trajectories. The total number of line segments in the data sets is 4M, 8M, 12M, 16M and 20M respectively. During the search process, if the data pages belong to the partitions that are not completely inside the query window, the spatial predicate is applied to each segment tuple in the data page.

The number of partitions that are not completely inside the query window is high for the RPTI structure, as the area of the partition is large in the higher levels. Almost every partition in the higher levels tends to intersect the query window and, hence, the spatial predicate is applied to each segment tuple in the level files at the higher level. When the query window size is of 0.01% selectivity, the results of the RPTI and SETI tend to be similar. As with the time-interval queries, the time-slice query requires the look-up of partitions at every level and this results in the slight increase in response time of the RPTI over the SETI.

However, the performance of the RPTI greatly depends on the query window size and the position of the query window. The variation in the number of partitions that will be for false positives at every level is dependent on the query window size and the position of the query window. The results shown in Figure 8 are for the query window size of 0.01% selectivity, however there might be an increase in the response time of RPTI if the query window size is large.

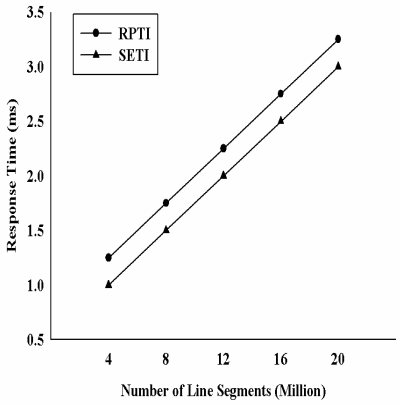

FIGURE 8: Time-interval Query Performance for 1.0% Query Selectivity

5.4 Trajectory-based Queries

The RPTI structure performs better than the SETI structure for the trajectory queries, as the search process for retrieving the segments of the trajectory is quite straightforward. The SETI structure, however, requires the search process that involves duplicate elimination.

We also present the performance of the RPTI structure for trajectory queries by varying the number of moving object trajectories in the data set. The number of moving objects varies from 40K to 160K and the number of segments per moving object is kept as 100. The number of segments in the data set ranges from 4M to 16M. We present the response time of the retrieving segment id of 100 trajectories given the trajectory id. The experimental results show that the RPTI structure is better in answering trajectory-based queries than the SETI, as is illustrated in Figure 9. For a fair comparison, we maintained a separate file for SETI, similar to the $track - trajectory$ in the RPTI, that keeps track of the cells associated with a trajectory.

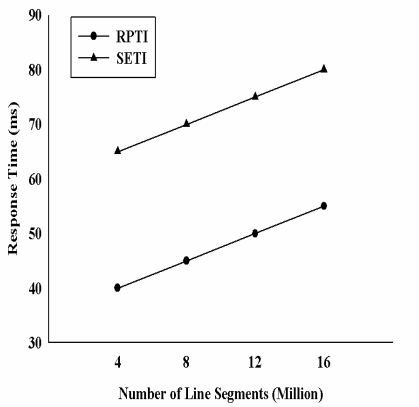

FIGURE 9: Trajectory Query Performance Scaling with the Number of Moving Objects

Deletion of trajectories is similar to trajectory queries where the whole or part of the trajectory is retrieved based on the trajectory id and, as mentioned earlier, the response time for deletion using the RPTI structure is less compared to the SETI. Deletion of trajectories in SETI involves deleting the line segments that cross the boundary twice and updating the R^*-tree twice if the deletion changes the $life - time$ of the data pages.

The only design parameters required are the standard disk page size and maximum level of recursive partitioning. However, in SETI, the number of spatial partitions, which is a crucial parameter in any spatial partitioning strategy, is highly dependent on the distribution of data sets.

6 Conclusion

We have discussed the use of the recursive partitioning technique for trajectory indexing. Similar to the SETI structure, the RPTI decouples the space and time dimension. Contrary to the SETI structure, the RPTI recursively partitions the space and keeps track of the segments of a trajectory. Hence, the RPTI is better than SETI in handling trajectory-based queries and is competitive with the SETI in handling coordinate-based queries. The R-tree structure or its variations can be used to index the time dimension at every level. As, the RPTI is efficient in retrieving all the segments of the trajectory, deletion of trajectories is efficiently done. The time-interval queries require the look-up of partitions at every level and the structure increases the number of false positives during the search process. Hence, the response time of the RPTI for time-interval queries and time-slice queries is slightly higher than the SETI. The structure of the RPTI can be easily implemented by using any of the existing spatial indexing structures.

Acknowledgements

This work is sponsored in part by National ICT Australia (NICTA).

References

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles.

In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990. ACM Press.

Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Survey*, 11(4):397–409, 1979.

Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 142–153, June 1998.

V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with seti. In *In Proceedings of Conference on Innovative Data Systems Research (CIDR)*, 2003.

Douglas Comer. The ubiquitous b-tree. *ACM Computing Survey*, 11(2):121–137, 1979.

Victor Teixeira de Almeida and Ralf Hartmut Güting. Indexing the trajectories of moving objects in networks. *Geoinformatica*, 9(1):33–60, 2005.

Martin Erwig and Markus Schneider. *STQL – A Spatio-Temporal Query Language*, chapter 6. Mining Spatio-Temporal Information Systems. Kluwer Academic Publishers, 2002.

Elias Frentzos. Indexing objects moving on fixed networks. In *In Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD)*, pages 289–305, Santorini Island, Greece, July 2003. Springer.

Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD'84, Proceedings of Annual Meeting*, pages 47–57, June 1984.

Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, Santiago de Chile, Chile, September 1994. Morgan Kaufmann.

George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 261–272, Philadelphia, Pennsylvania, June 1999. ACM Press.

Yannis Manolopoulos, Alexandros Nanopoulos, and Apostolos N. Papadopoulos. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer, 1 edition, September 2005. ISBN 1852339772.

Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84: Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, Waterloo, Ontario, Canada, April 1984. ACM.

Dimitris Papadias, Yufei Tao, Panos Kalnis, and Jun Zhang. Indexing spatio-temporal data warehouses. In *Proceedings of 18th International Conference on Data Engineering (ICDE)*, pages 166–175, San Jose, CA, March 2000. IEEE Computer Society.

Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB 2000: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 395–406, Cairo, Egypt, September 2000. Morgan Kaufmann.

Evaggelia Pitoura and George Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, 2001.

Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 331–342, Dallas, Texas, USA, May 2000. ACM.

Hanan Samet. Modern database systems: The object model, interoperability, and beyond. In *Modern Database Systems*. ACM Press and Addison-Wesley, 1995. ISBN 0-201-59098-0.

Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, Brighton, England, September 1987. Morgan Kaufmann.

Yannis Theodoridis, Michalis Vazirgiannis, and Timos K. Sellis. Spatio-temporal indexing for large multimedia applications. In *ICMCS '96: Proceedings of the 1996 International Conference on Multimedia Computing and Systems*, pages 441–448, 1996.

Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, pages 147–164, Hong Kong, China, July 1999. Springer.

Andy Ward and Alan Jones. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, 1997.