Chapter 2
Trajectory Indexing and Retrieval

Ke Deng, Kexin Xie, Kevin Zheng and Xiaofang Zhou

Abstract The traveling history of moving objects such as a person, a vehicle, or an animal have been exploited in various applications. The utility of trajectory data depends on the effective and efficient trajectory query processing in trajectory databases. Trajectory queries aim to evaluate spatiotemporal relationships among spatial data objects. In this chapter, we classify trajectory queries into three types, and introduce the various distance measures encountered in trajectory queries. The access methods of trajectories and the basic query processing techniques are presented as another component of this chapter.

2.1 Introduction

Trajectories are the traveling history of moving objects such as a person, a vehicle, or an animal. Trajectories can be used for complex analysis across different domains. For example, public transport systems may go back in time to any particular instant or period to analyze the pattern of traffic flow and the causes of traffic congestions; in biological studies movements of animals may be analyzed when considering road networks to reveal the impact of human activity on wild life; or the urban planning of a city council may analyze the trajectories to predict the development of suburbs and provide support in decision making. Other applications include path optimization of

Ke Deng
The University of Queensland, Brisbane, Australia, e-mail: dengke@itee.uq.edu.au

Kexin Xie
The University of Queensland, Brisbane, Australia, e-mail: kexin@itee.uq.edu.au

Kevin Zheng
The University of Queensland, Brisbane, Australia, e-mail: kevinz@itee.uq.edu.au

Xiaofang Zhou
The University of Queensland, Brisbane, Australia, e-mail: zxf@itee.uq.edu.au

logistics companies, improvement of public security management, and personalized location-based services, etc.

Performing this complex analysis requires trajectory databases to support trajectory queries effectively. Trajectory queries can be classified into three types according to their spatiotemporal relationships: 1) trajectories and points (e.g. find all trajectories within 500m of a gas station between 9:00pm-9:30pm), 2) trajectories and regions (e.g. find the region which is passed by τ trajectories between 9:00pm-9:30pm), and 3) trajectories and trajectories (e.g. travelers who may take a similar path in the coming 30mins). Clearly, the spatiotemporal relationship in trajectory queries concerns not only the topological relationship such as passing a region, but also the distance measures between spatial objects from the simple such as Euclidean distances to the complex such as the similarities between trajectories. The background diversity of applications determines that the definition of distance measure is variable and should have a sound interpretation in certain application contexts. Several popular distance functions are introduced in this chapter.

Due to their historical nature, the size of trajectory databases is supposed to be very large. Thus, a critical aspect of trajectory databases is to support an effective trajectory index to accelerate query processing. In general, the spatiotemporal data index technique is an extension of the spatial data index with augmentation of a time dimension. However, the trajectory data have specific requirements to index techniques due to unique data characteristics, i.e. continuous long period of time, and due to unique query characteristics, i.e. often asking for information in an instantaneous/continuous time window. We present several representative trajectory indexes in this chapter with processing techniques for various trajectory query types.

This chapter is organized as follows. In section 2.2, we classify trajectory queries into three types. Section 2.3 focuses on the trajectory similarity measures/distance matrices used in trajectory queries. A number of trajectory indexes are discussed in section 2.4 and in section 5 we present the processing of trajectory queries with the support of index. This chapter is summarized in section 2.6.

2.2 Trajectory Query Types

Trajectories are the historical spatiotemporal data which are the foundation of many important and practical applications such as traffic analysis, behavior analysis and so on. The utility of trajectory data is built on various queries in a trajectory database. A typical trajectory query asks for the information against spatiotemporal relationships between trajectories and with other spatial data objects, i.e. points (P), regions (R) and trajectories (T). We generally classify the trajectory queries into three types:

- P-Query 1) asks for points of interest (POI) which satisfy the spatiotemporal relationship to specified trajectory segment(s) (e.g. top k nearest neighbors); or 2) asks for trajectories which satisfy the spatiotemporal relationship to a specified point/points.

- R-Query 1) asks for trajectories passing a given spatiotemporal region R; or 2) asks for spatiotemporal regions which are overlapped or frequently passed by trajectories.
- T-Query 1) asks for similar trajectories in a set of trajectories; or 2) asks for trajectories within a distance threshold (i.e. nearest approaching points of two trajectories).

In addition to spatiotemporal relationships, the query in a trajectory database may ask for trajectories which satisfy extra navigational conditions (e.g. top speed and direction within a certain area during a given time interval). To process such queries, the relevant trajectory segments extracted using R-query are examined against the navigational conditions. In trajectory queries, the absence of specification in time (time stamp or interval) implies that any character of object within a time dimension is acceptable. But, we treat this situation as a special case in trajectory queries.

2.2.1 P-Query

2.2.1.1 Ask for POI

P-query aims to find spatiotemporal point/points of interest which satisfy expected spatiotemporal relationships given segments of a trajectory/trajectories. A basic type of P-query is to retrieve the nearest neighbor of every point in a trajectory segment [5]. For example, you want to find all nearest gas stations along the specified trajectory segment from point s to point e. The result contains a set of $< POI, trajectory\_interval >$ tuples, such that POI is the nearest neighbor of all points in the corresponding trajectory interval. Some variants of P-query consider more practical factors in the context of a road network, that is, the traveling direction and destination along the trajectory [30, 29, 9]. The aim of such P-queries is to find one point from many, such as a set of gas stations, via which the detour from the original route (i.e. leave the route and come back after visiting the gas station) on the way to the destination is the smallest.

P-Query can also be an envelop query (or range query) which asks for all POIs (i.e. gas stations) whose distances (according to a distance metric) to a given trajectory is less than a threshold in a specified time interval.

2.2.1.2 Ask for Trajectory

P-query may ask for segment(s) of trajectory/trajectories when a given spatiotemporal point/points are specified. The single point based query [11, 26] looks for the nearest trajectories to only one point (e.g. a supermarket). Similarly, P-Query for trajectory may also ask for all trajectories which are within a proximity of a point such as within 500m in a time interval.

In the multiple-points trajectory query [10], given a small set of points with or without an order specified, the target is to find the trajectories from a trajectory database such that these trajectories best connect the designated points geographically. This query is useful when planning a traveling path passing several must-go locations such as sightsee points.

2.2.2 R-Query

2.2.2.1 Ask for Trajectory

Given a three dimensional spatiotemporal region (or block), R-query searches for trajectory segments that belong to the specified spatiotemporal region. This type of query can be used to support traffic analysis, and located-based services. As a basic operation in various applications, this query type has been studied extensively and various indexes have been proposed (TB, 3D-R tree, STR tree).

2.2.2.2 Ask for Regions

Even though trajectories are independent of each other, they often show common behavior such as passing a small region within a certain period of time. Identifying these regions with R-query is important in trajectory clustering [19, 17]. An application is to identify the regions which are more likely to be passed by a given user in a time window based on the many trajectories relevant to that user.

2.2.3 T-Query

2.2.3.1 Ask for Similar Trajectories

T-Query usually asks for similar trajectories in a trajectory database with attempt to classify/cluster trajectories. The trajectory classification/clustering can be used in many applications such as to predict the class labels of moving objects based on their trajectories and other features, to identify the traffic flow in road networks, or for instance, to identify interesting behavior patterns, etc. One version of T-query is to discover common sub-trajectories [20, 19] and another version targets the problem of long trajectories spreading over large geographic areas [18].

2.2.3.2 Ask for Close Trajectories

Another task of T-trajectory is to find the closest trajectories. While the similarity of trajectories is based on the distance measure at a sequence of points, the closeness of trajectories refers to the shortest distance. Using this query, scientists may, for instance, investigate the behaviors of animals near the road where the traffic rate has been varied.

2.2.4 Applications

The booming industry of location-based services has accumulated a huge collection of users' location trajectories of driving, cycling, hiking, etc. In trajectory databases, a pattern can represent the aggregated abstraction of many individual trajectories of moving object(s). The trajectory based pattern has been widely used in many important and practical applications including traffic flow monitoring, path planning, behavior mining and predictive queries.

Traffic Flow Patterns Discovery of traffic flow patterns is important for applications requiring classification/profiling based on monitored movement patterns, such as targeted advertising, or resource allocation. The problem of maintaining hot motion paths, i.e. routes frequently followed by multiple objects over the recent past, has been investigated in recent years [28, 21].

To achieve this goal, [28] delegate part of the path extraction process to objects, by assigning to them adaptive lightweight filters that dynamically suppress unnecessary location updates and, , and thus, help reduce the communication overhead. [21] proposes a new density-based algorithm named FlowScan which finds hot routes in a road network. Instead of clustering the moving objects, road segments are clustered based on the density of common traffic they share. Another interesting problem concerns the road speed pattern based on the the traffic database that records observed road speeds under a variety of conditions (e.g., weather, time of day, and accidents) [12]. For example, if a road segment is a certain condition, then a speed factor is assigned, where speed factor is the speedup or slowdown for the road segment with respect to the base-speed. Speed factors are clustered to increase rule support. The concise set of rules is mined through a decision tree induction algorithm.

Behavior Mining In many applications, objects follow the same routes (approximately) over regular time intervals. For example, people wake up at the same time and follow more or less the same route to work every day and go shopping weekly. Thus, trajectory patterns can be viewed as concise descriptions of frequent behaviors, in terms of both space (i.e., the regions of space visited during movements) and time (i.e., the duration of movements). The discovery of hidden periodic movement patterns in spatiotemporal data, apart from unveiling important information to the data analyst, can facilitate data management substantially. The basic task is to map

the trajectories to human activities based on the spatiotemporal relationship, e.g., the trajectory of a person frequently remaining at a location from 9.00am to 7.00pm usually means that they work at that location.

Path Planning An effective path planning (or route recommendation) solution is beneficial to travelers who are asking directions or planning a trip. Historical traveling experiences can reveal valuable information on how other people usually choose routes between locations. The rationale behind this is that people must have good reasons to choose these routes, e.g., they may want to avoid those routes that are likely to encounter accidents, road construction, or traffic jams.

A problem is how to mine frequently traveled road-segment-sequences in order to obtain important driving hints that are hidden in the data [12]. An area-level mining computes frequent path-segments with support relative to the traffic in the area, e.g., lower support thresholds for edges in rural areas compared to those in big cities. While the road-segment-sequence may help in suggesting a drive turn at some intersection in a general case, they are not sufficient and accurate to discover a popular route to some specified destination. This is because simply counting the number of trajectories is not enough to discover the popular route between any two locations, for instance, there may be no records of direct trajectories. To overcome this difficulty, a so-called *transfer network* is generated from raw trajectories [8] developed a transfer network from raw trajectories as an intermediate result to capture the moving behaviors between locations, and to facilitate the search for the popular route.

Another interesting problem presented in [39] mines the history of travelers to find the sound traveling path when passing several locations, like culturally important places such as the Sydney Opera House, or frequented public areas like shopping malls and restaurants, and so on.

Predictive Query Given the recent movements of an object and the current time, predictive queries ask for the probable location of the object at some future time. A simple approach is based on the assumption that objects move according to linear functions. In practice movement is more complex, and individual objects may follow drastically different motion patterns. In order to overcome these problems, a nonlinear function can be modeled to accurately capture its movement in the recent past [32].

While [32] is more suitable for prediction of short time, [16, 23] accurately forecast locations when the forecast time is far away from the current time. The long term prediction uses previously extracted movement patterns named Trajectory Patterns, which are a concise representation of behaviors of moving objects as sequences of regions frequently visited within a typical travel time. It has been shown that prediction based on the trajectory patterns of an object is a powerful method.

2.3 Trajectory Similarity Measures

An important consideration in similarity-based retrieval of moving object trajectories is the definition of a distance function.

2.3.1 Point to Trajectory

The similarity between a query point q and a trajectory A is usually measured by the distance from q to the nearest point of A, i.e.,

$$D(q,A) = \min_{p' \in A} d(q,p')$$

where $d(,)$ is some distance function between two points, which can be either L_p-norms (if they are in Euclidean space) or network distance (if they are on spatial networks).

We can also extend a single query point to multiple query points. In this case we need a similarity function to score how well a trajectory connects the query locations, and which considers the distance from the trajectory to each query point. The work [10] has studied this kind of query and propose a novel similarity function

$$Sim(Q,A) = \sum_{q \in Q} e^{D(q,A)}$$

The intuition of using the exponential function is to assign a larger contribution to a closer matched pair of points while giving much lower value to those far away pairs. As a result, only the trajectory that is reasonably close to all the query locations can be considered as 'similar'.

2.3.2 Trajectory to Trajectory

The similarity between two trajectories is usually measured by some kind of aggregation of distances between trajectory points. Along this line, several typical similarity functions for different applications include Closest-Pair Distance, Sum-of-Pairs Distance [1], Dynamic Time Warping (DTW) [38], Longest Common Subsequence (LCSS) [39], and Edit Distance with Real Penalty (ERP) [10], Edit Distance on Real Sequences (EDR) [7]. It is worth noting that some of those similarity functions were originally proposed for time series data. But as trajectories can be regarded as a special kind of time series in multi-dimensional space, these similarity functions can also be applied to trajectory data.

2.3.2.1 Closest-Pair Distance

A simple way to measure the similarity between two trajectories is to use their minimum distance. To do so, we find the closest pair of points from two trajectories and calculate the distance between them. More formally, given two trajectories A and B, their Closest-Pair distance can be computed as follows:

$$CPD(A,B) = \min_{a_i \in A, b_j \in B} d(a_i, b_j)$$

2.3.2.2 Sum-of-Pairs Distance

Another similarity function between two trajectories was proposed by Agrawal et al. [1], who simply use the sum of distances between the corresponding pairs of points to measure the similarity. Let A, B be two trajectories with the same number of points, their distance is defined as follows:

$$SPD(A,B) = \sum_{i=1}^{n} d(a_i, b_i)$$

2.3.2.3 Dynamic Time Warping Distance

As we have seen, an obvious limitation of Euclidean distance is that, it requires the two trajectories to be of the same length, which is very unlikely for real life applications. More ideal similarity measures should have a certain flexibility on the lengths of two trajectories. Dynamic time warping (DTW) distance is the first one based on this motivation. The basic idea of DTW is to allow 'repeating' some points as many times as needed in order to get the best alignment.

Given a trajectory $A = \langle a_1, ..., a_n \rangle$, let $Head(A)$ denote a_1 and $Rest(A)$ denote $\langle a_2, ..., a_n \rangle$. The time warping distance between two trajectories A and B with lengths of n and m is defined as [38]:

$$DTW(A,B) = \begin{cases} 0, \ if \ n = 0 \ and \ m = 0 \\ \infty, \ if \ n = 0 \ or \ m = 0 \\ d(Head(A), Head(B)) + \min \begin{cases} DTW(A, Rest(B)) \\ DTW(Rest(A), B) \\ DTW(Rest(A), Rest(B))) \end{cases} \end{cases}$$

where $d(,)$ can be any of the distance functions defined on points.

2.3.2.4 Longest Common Subsequence

A common drawback of the previous two similarity functions is that they are relatively sensitive to noise, since all points including noises are required to match. Therefore it is possible to accumulate an overly large distance merely due to a single noisy point. To address this issue, Longest Common Subsequence (LCSS) has been proposed to measure the distance of two trajectories in a more robust manner. Its basic idea is to allow skipping over some points rather than just rearranging them. As a consequence, far-away points will be ignored, making it robust to noises. The advantages of the LCSS method are twofold: 1) some points can be unmatched, while in Euclidean distance and DTW distance all points have to be matched, even when they are outliers. 2) The LCSS distance allows a more efficient approximate computation.

Let A and B be two trajectories with lengths of n and m respectively. Given an integer δ and a distance threshold ε, the LCSS between A and B is defined as follows:

$$LCSS(A,B) = \begin{cases} 0, \ if \ n = 0 \ or \ m = 0 \\ 1 + LCSS(Rest(A), Rest(B)), \ if \ d(Head(A), Head(B)) \leq \varepsilon, \\ \qquad\qquad\qquad\qquad\qquad\qquad and \ |n-m| < \delta \\ \max(LCSS(Rest(A),B), LCSS(A, Rest(B))), otherwise \end{cases}$$

The parameter δ is used to control how far in time we can go in order to match a given point from one trajectory to a point in another trajectory. ε is a matching threshold to determine whether to take the point into account.

Based on the concept of LCSS, two similarity functions $S1$ and $S2$ have been proposed in [39].

Definition 2.1. The similarity function $S1$ between two trajectories A and B, given δ and ε, is defined as follows:

$$S1(\delta, \varepsilon, A, B) = \frac{LCSS(A,B)}{min(n,m)}$$

$S2$ is defined based on $S1$, but it allows the translations of trajectories. A translation of trajectory is a movement on all the points with the same offset. Let $\mathscr{F}$ be the set of all possible translations.

Definition 2.2. Given δ, ε and the family $\mathscr{F}$ of translations, $S2$ between two trajectories A and B is defined:

$$S2(\delta, \varepsilon, A, B) = \max_{f \in \mathscr{F}} S1(\delta, \varepsilon, A, f(B))$$

So both similarity functions take the range from 0 to 1. It is worth noting that $S2$ is an improvement over the $S1$, because by allowing translations, similarities between movements that are parallel in space but not identical can be detected.

2.3.2.5 EDR Distance

Although LCSS can handle trajectories with noises, but it is a very coarse measure since it does not differentiate trajectories with similar common subsequences, but focuses on different sizes of gaps in between. This has motivated the proposal of a new distance function, called Edit Distance on Real Sequence [7], to be proposed.

Definition 2.3. Given two trajectories A, B with lengths of n and m respectively, a matching threshold ε, the Edit Distance on Real sequence (EDR) between A and B is the number of insert, delete, or replace operations that are needed to change A into B.

$$EDR(A,B) = \begin{cases} n & if\ m = 0 \\ m & if\ n = 0 \\ \min\{EDR(Rest(R), Rest(S) + subcost, & otherwise \\ EDR(Rest(R), S) + 1, EDR(R, Rest(S)) + 1\} & \end{cases}$$

where

$$subcost = \begin{cases} 0, & if\ d(Head(A), Head(B)) \leq \varepsilon \\ 1, & otherwise \end{cases}$$

Compared to Euclidean distance, DTW, and LCSS, EDR has the following virtues:

- In EDR, the matching threshold reduces effects of noise by quantizing the distance between a pair of elements to two values, 0 and 1 (LCSS also performs the same quantization). Therefore, the effect of outliers on the measured distance is much less in EDR than that in Euclidean distance, DTW.
- Contrary to LCSS, EDR assigns penalties to the gaps between two matched sub-trajectories according to the lengths of gaps, which makes it more accurate than LCSS.

2.3.2.6 ERP Distance

All the similarity functions discussed before can be classified into two categories. The first one is the Euclidean distance, which is a metric but cannot support local time shifting. The second category includes DTW, LCSS, EDR which are capable of handling local time shifting but are non-metric. To tackle this problem, Edit distance with Real Penalty (ERP for short) [6] has been proposed, representing a marriage of L1-norm and the edit distance.

By analyzing DTW distance more carefully, it can be observed that the reason DTW is not metric is because, when a gap needs to be added, it repeats the previous points. Thus the difference between a point and a gap depends on the previous point. On the contrary, ERP uses real penalty between two matched points, but a constant value for computing the distance for unmatched points. As a consequence, ERP can support local time shifting, and is a metric.

Given two trajectories A, B with lengths of n and m respectively, a random point g, ERP distance is defined as follows:

$$ERP(A,B) = \begin{cases} \sum_1^n |s_i - g|, \ if \ m = 0 \\ \sum_1^m |r_i - g|, \ if \ n = 0 \\ \min \begin{cases} ERP(Rest(A), Rest(B)) + d(Head(A), Head(B)), \\ ERP(Rest(A), B) + d(Head(A), g), \ otherwise \\ ERP(A, Rest(B)) + d(Head(B), g) \end{cases} \end{cases} \quad (2.1)$$

2.4 Trajectory Indexes

It is important that trajectory databases can support the retrieval of trajectories of an arbitrarily large size. To extract qualitative information by querying databases containing very large numbers of trajectories, the efficiency depends crucially upon an appropriate index of trajectories. The trajectory database has specific requirements to index techniques due to both unique data characteristics, i.e. a continuous long period of time, and unique query characteristics, i.e. often asking for information in an instantaneous/continuous time window.

There are mainly three index approaches. The first is to use any multidimensional access method like R-tree indexes with augmentation in temporary dimensions such as 3D R-tree [25], or STR-tree [25]. The second approach uses multiversion structures, such as MR-tree [37], HR-tree [24], HR+-tree [33], and MV3R-tree [34]. This approach builds a separate R-tree for each time stamp and shares common parts between two consecutive R-trees. The third approach divides the spatial dimension into grids, and then builds a separate temporal index for each grid. This category includes SETI [4], and MTSB-tree [40].

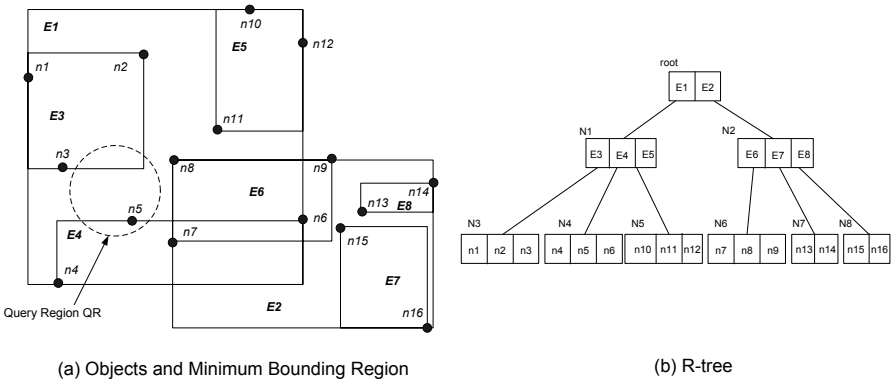
Here, we are only interested in the data structure indexing the trajectory in history. We ignore the indexes, like [31] [31] used, to update a large number of moving objects in a system, since they concern the latest location information about the moving objects.

2.4.1 Augmented R-tree

R-Tree We first briefly introduce R-tree and then two extended versions of R-tree with augmentation of temporal dimension. R-tree [13] is efficient and simple. It has been widely recognized in the spatial database research community and found its way into commercial spatial database management systems. In addition, many trajectory indexing structures are variants or based on R-trees.

R-tree is a height-balanced data structure. Each node in R-tree represents a region which is the minimum bounding box (MBB) of all its children nodes. Each data entry in a node contains the information of the MBB associated with the referenced

children node. The search key of an R-tree is the MBB of each node. Figure 2.1 shows two views of an R-tree. In Figure 2.1(b), we see a tree structure, and in Figure 2.1(a), we see how the data objects and bounding boxes are distributed in space.



(a) Objects and Minimum Bounding Region (b) R-tree

Fig. 2.1 Two views of an R-tree Example.

The root has two data entries $E1$, $E2$ referring to the children nodes $N1$ and $N2$ separately. $N1$ represents the minimum bounding box of its children nodes $N3$, $N4$ and $N5$ and the information of the bounding box is contained in $E1$, as with $N2$. The spatial objects are referred by the entries of the leaf nodes in R-tree.

R-tree can be used in range query to retrieve all spatial objects within a specified query region, e.g., a sphere of 5km radius around a location. R-tree is traversed from the root. The query region is examined against the MBB in each entry visited. If they are contained/overlapped, this entry is accessed and its referring child node is visited. In this way, the nodes are visited only if its MBB is relevant (e.g., contained/overlapped) to the query region. When all relevant nodes have been visited, query processing terminates. For example in Figure 2.1, *QR* is a query region and $N1$ is visited since its MBB overlaps with *QR*; then for the same reason $N3$ and $N4$ are visited and $n5$ is found from $N4$. To search for a certain point using R-tree, is the same as with processing range query. For example, to search for $n7$, $N1$ and $N2$ are visited and then $n7$ is found in $N6$. Note that $N2$ is visited although $n7$ is not in the subtree of $N2$. The reason is that MBBs of R-tree nodes overlap each other and therefore multiple searching paths may be followed. A variant of R-tree, called R*-tree [2], has been developed to minimize node access by reducing overlapping of MBBs.

R-tree can also be used to process a nearest neighbor query for a given query point. There are two traverse strategies, depth-first and best-first [14]. In both strategies, the minimum distance from each MBB to the query point is used and denoted as *mindist*. Using the best-first traverse, the root node of R-tree is visited first. Each entry in the root node computes its *mindist* to the query point and is kept in a heap sorted in ascending order of *mindist*. Each time the first entry in the heap is expanded and replaced by its child nodes. Once the first entry is a leaf node, the object is

the first nearest neighbor. Using the depth-first traverse, a leaf node is visited first in some way and its *mindist* to the query point is used as a threshold T. Then, other nodes of R-tree which probably contain objects closer to the query point (i.e., $mindist < T$) are accessed and checked. If an object has $mindist < T$, T is updated with *mindist*. The query is returned when all such nodes have been visited. R-tree is also used in processing spatial join queries. More details can be found in [15, 3].

Inserting a new data point p to a R-tree takes the following steps. First, a leaf node L is selected in which to place p. The selection starts from the root of the R-tree and each time a subtree is chosen until the lead node is reached. Among several subtrees, we always chose the one which needs the least enlargement to include p. Second, we insert p into L if L has room for anther entry, otherwise a structure change happens by splitting L to two nodes. Third, the structure change is propagated upward. If the node split propagation caused the root to split, a new root is created and the R-tree grows taller as a consequence.

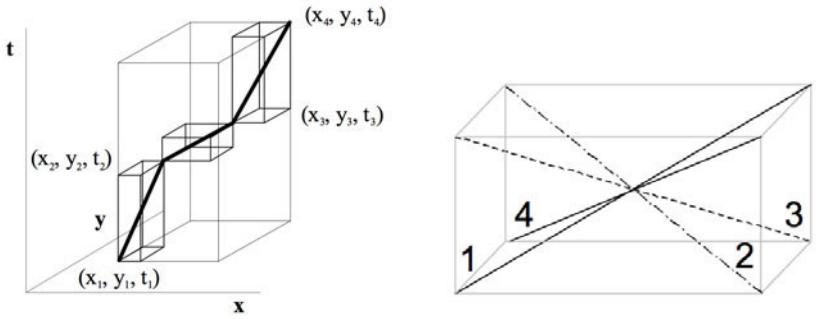
Deleting a data point p from an R-tree takes following steps. First, the leaf node L containing p is found and remove p from L. Second, L is eliminated if it has too few entries and relocate its entries. The change is propagated upward as necessary. Third, if the root node has one child after the tree has been adjusted, the child becomes the new root and R-tree is shortened as a consequence.

3D R-Tree In trajectory databases, traditional spatial queries need to consider the temporal aspect of the spatial objects. Two basic queries in temporal databases are *timestamp queries*, which retrieve all objects at a specific timestamp, and *interval queries*, which retrieve all objects lasting for several continuous timestamps. As a result, common queries for trajectory databases include finding the spatial objects according to some spatial requirement (e.g. trajectories within a region) for a given timestamp or interval.

In order to capture the temporal aspect of the spatial objects, R-tree can be simply modified to model the temporal aspect of the trajectory as an additional dimension. As a result, a 2 dimensional trajectory line segment can be bounded by a 3 dimensional MBB (i.e.minimum bounding box) as shown in Figure 2.4.1(a). The problem with using MBB to bound 3D line segments is that the MBB used to bound the actual line segment covers a much larger portion of space, whereas the line segment itself is small. As a result, the 3D R-tree have smaller discrimination capability because of the high overlap between MBBs.

Another aspect not captured in R-tree is knowledge about the specific trajectory to which a line segment belongs. To smoothen these inefficiencies, the 3 dimensional R-tree can be modified as follows. As can be seen in Figure 2.4.1(b), a line segment can only be contained in four different ways. This extra information is stored in four different ways in an MBB. This extra information is stored at the leaf level by simply modifying the entry format to $(id, MBB, orientation)$, where the orientation's domain is $\{1,2,3,4\}$. Assuming the trajectories are numbered from 0 to n, a leaf node entry is then of the form $(id, trajectory\#, MBB, orientation)$.

In 3D R-tree, the insertion and deletion of data points follows R-tree as discussed above.



(a) Results of mean filter (b) Results of median filter

Fig. 2.2 (a) Approximating trajectories using MBBs. (b) Mapping of line segments in a MBB [25]

STR-Tree The STR-tree (Spatio-Temporal R-Tree) is an extension of the 3D R-tree to support efficient query processing of trajectories of moving points. STR-tree and 3D R-tree differ in insertion/split strategy.

The insertion process is considerably different from the 3D R-tree. As already mentioned, the insertion strategy of the R-tree is based purely on the spatial least enlargement criterion. STR-tree improves this by not only considering *spatial closeness*, but also partial *trajectory preservation*, by trying to keep line segments belonging to the same trajectory together. As a result, when inserting a new line segment, the goal is to insert it as close as possible to its predecessor in the same trajectory. In order to achieve this, the STR-tree uses an algorithm *FindNode* to find which node contains the predecessor. As for the insertion, if there is room in this node, the new segment is inserted there, otherwise, a node split strategy has to be applied.

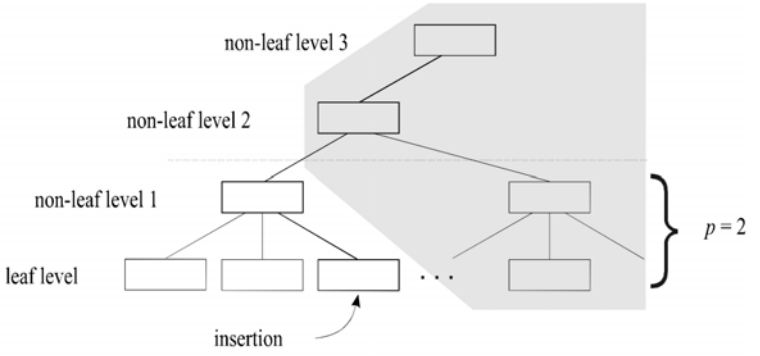


Fig. 2.3 STR-tree Insertion [25]

The ideal characteristics for an index suitable for object trajectories would be to decompose the overall space according to time, the dominant dimension in which "growth" occurs, while simultaneously preserving trajectories. An additional parameter is introduced by [25], called the *preservation parameter*, p, that indicates the number of levels to be "reserved" for the preservation of trajectories. When a leaf node returned by *FindNode* is full, the algorithm checks whether the $p-1$ parent nodes are full (in Figure 2.3, for $p=2$, only the node draw in bold at non-leaf level 1 has to be checked). In case one of them is not full, the leaf node is split. In case all of the $p-1$ parent nodes are full, another leaf node is selected on the subtree including all the nodes further on the right of the current insertion path (the gray shaded tree in Figure 2.3). A smaller p decreases the trajectory preservation and increases the spatial discrimination capabilities of index. The converse is true for a larger p. The experimental results show that the best choice of a preservation parameter is $p=2$ [25].

TB-Tree The TB-tree (Trajectory-Bundle 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 structure of the TB-tree is a set of leaf nodes, each containing a partial trajectory, organized in a tree hierarchy. In other words, a trajectory is distributed over a set of disconnected leaf nodes. Figure 2.4 shows a part of a TB-tree structure and a trajectory illustrating this approach. The trajectory symbolized by the gray band is fragmented across six nodes $c1$, $c3$, etc. In the TB-tree, these leaf nodes are connected through a linked list.

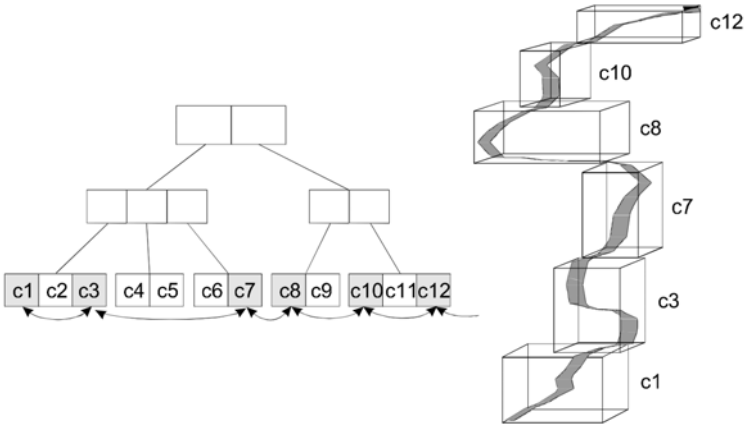


Fig. 2.4 The TB-tree structure [25]

To insert a trajectory into TB-tree, the goal is to "cut" the whole trajectory of a moving object into pieces, where each piece contains M line segments, with M being the fanout. The process starts by traversing the tree from the root and steps into every child node that overlaps with the MBB of the new line segment. The leaf node containing a segment connected to the new entry (stage 1 in Figure 2.5) is chosen. In case the leaf node is full, in order to preserve the trajectory, instead of a split, a new leaf node is created. As shown in the example, the tree is traversed until a non-full parent node is found (stages 2 through 4). The right-most path (stage 5) is chosen to insert the new node. If there is room in the parent node (stage 6), the new leaf node is inserted as shown in Figure 2.5.

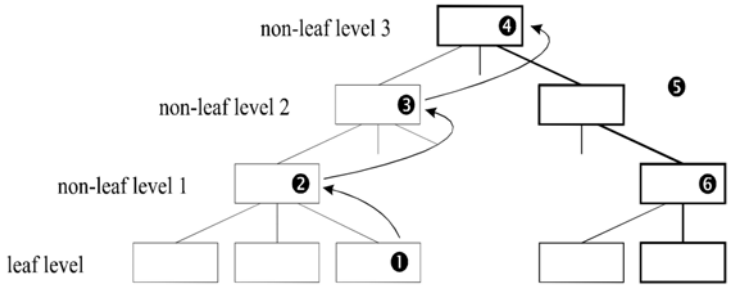


Fig. 2.5 Insertion into TB-tree [25]

2.4.2 Multiversion R-trees

Another solution, different to augmenting a temporal dimension to R-tree, is to create an R-tree for each timestamp in the history, and then index the R-trees with a one dimensional indexing structure (e.g. B-tree) to support time slice and interval queries. As a result, one can simply use the B-tree to find the corresponding R-trees for the corresponding temporal constraints and then process the spatial object in those trees with the spatial aspect of the queries.

Obviously, creating an R-tree for each timestamp is not practical due to the storage overhead. Instead of creating a complete R-tree to index the spatial objects for each of the timestamps, the space consumption can be saved by only creating the part of the R-tree that is different from the previous timestamp, i.e. if consecutive R-trees share branches when the objects do not move, and new branches are only created otherwise. The indexing structures that employ this strategy are MR-tree [37] (Multiversion R-tree), HR-tree [24] (Historical R-tree) and HR+-tree [33].

Figure 2.6 shows an example of HR-tree which stores spatial objects for timestamp 0 and 1. Since all spatial objects in A_0 do not move, the entire branch is shared

by both trees R_0 and R_1. In this case, it is not necessary to recreate the entire branch in R_1, instead, a pointer is created to point branch A_0 in R_0.

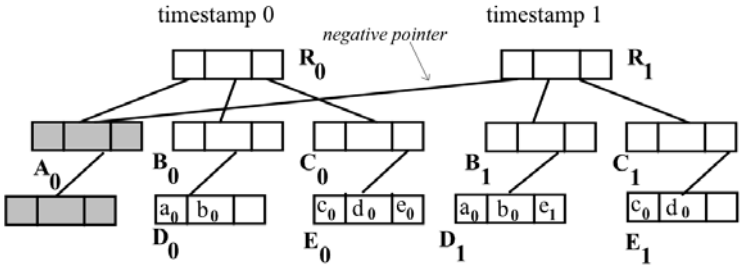


Fig. 2.6 Example of HR-tree [33]

HR+-tree improves HR-tree by allowing entries of different timestamps to be placed in the same node. This means that objects within the same node with a small change of position may still be shared with different R-trees. According to [33], HR+-tree consumes less than 20% of the space required by HR-tree yet answers interval queries several times faster, with similar query processing time for timestamp queries.

Both HR-tree and HR+-tree are efficient in timestamp queries. Although, they are better than using an R-tree for each timestamp, however, the extensive duplication of objects still leads to considerable space cost, and relatively poor performance on interval queries. MV3R-tree [34] utilizes both multiversion B-Trees and 3D R-trees to overcome disadvantages. A MV3R-tree consists of two structures: a multiversion R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaves of the MVR-tree in order to process interval queries. Figure 2.7 shows an overview of MV3R-tree.

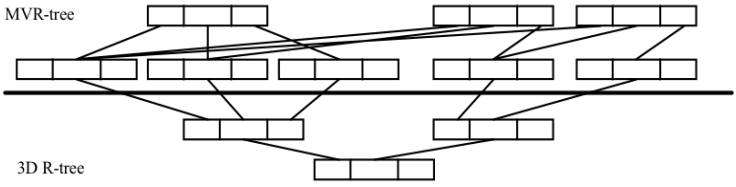


Fig. 2.7 MV3R-tree [34]

2.4.3 Grid Based Index

In a 3D R-tree, the temporal and the spatial dimensions are treated equally, i.e. a bounding box in the index includes the time dimension. This increases the chance of overlap amongst the keys. Consequently, the performance of the 3D R-tree degrades rapidly as the data set size increases. However, for trajectory data, the temporal and the spatial dimensions have important different characteristics. More specifically, the boundaries of the spatial dimensions remain constant or change very slowly over the lifetime of the trajectories, whereas the time dimension is continually increasing. This observation motivates the grid based index which partitions the spatial dimensions statically. Within each spatial partition, the indexing structure only needs to index lines in a 1-D (time) dimension. Such an approach will not exhibit the rapid degradation in index performance observed for 3-D indexing techniques. The SETI indexing mechanism (Scalable and Efficient Trajectory Index) [4] is the first grid based index.

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. 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. 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, the partition boundaries are fixed, whereas in a dynamic partitioning strategy the partition boundaries may change over time.

A variant of SETI is MTSB-tree (Multi Time Split B-tree). Similar to SETI, MTSB-tree partitions the spatial space into cells, and maintains a temporal access method corresponding to each cell. A trajectory is stored in all cells it intersects. Different from SETI where an R*-tree is used to index the time dimension within each cell, MTSB-tree uses the TSB-tree (Time Split B-tree) [22] to index the time dimension within each cell. Compared to R*-tree, the advantage of using TSB-tree is that it provides results in order of time. So, MTSB is more suitable for processing queries where trajectories are close in spatial space as well as in time. Another variant of SETI is CSE-tree (Compressed Start-End tree) [36] where different indexes are used to index time dimension within each cell; $B+$-tree for frequently updated data and sorted dynamic array for rarely updated data.

2.5 Query Processing

2.5.1 Query Processing in Spatial Databases

Since trajectories are spatial objects themselves, in this section, we first review the traditional query processing techniques for spatial objects, in particular, the spatial range search as well as nearest neighbor search.

As introduced earlier, spatial objects are usually organized by R-trees in spatial databases. Since computing the spatial relationship (e.g. distances, containments, etc) between spatial objects is expensive, a query processing algorithm typically employs a filter-and-refine approach. The *filter* step uses relatively cheap computation cost to find a set of candidate objects that are likely to be the results. The *refine* step is to further identify the actual query result from the small set of candidates. The overall efficiency of a spatial query processing algorithm mainly depends on the effectiveness of the filtering step.

```
N ← the root node of the R-tree;
Q ← a priority queue initialized with one element ⟨N⟩;
C ← ∅;
while Q is not empty do
    E ← Q.pop_first();
    if E is a leaf node then
        if filter(E, O_q) then
            C ← C ∪ {E};
        end
    else
        foreach E' ∈ E.children do
            if filter(E', O_q) then
                Q.append(E');
            end
        end
    end
end
return refine(C);
```

Algorithm 1: Breadth-First Spatial Query Processing

Algorithm 1 and 2 show the general framework of the filter-and-refine approach for spatial query processing. The only difference is how the nodes of the R-tree are traversed. Generally speaking, a depth-first search will consume a relatively small amount of memory, while a breadth-first search can find all the candidates more efficiently. The function $refine(C)$ is usually implemented to evaluate the actual spatial relationship between each element of C and the query object. The function $filter(E, O_q)$ is a function based on $refine(C)$ to eliminate the spatial objects that are impossible to be returned. For example, given a spatial region QR, a *spatial range* query is to search for all the objects that overlaps with QR. The function The

```
N ← the root node of the R-tree;
Q ← a priority queue initialized with one element ⟨N⟩;
C ← ∅;
while Q is not empty do
    E ← Q.pop_first();
    if E is a leaf node then
        if filter(E, O_q) then
            C ← C ∪ {E};
        end
    else
        foreach E' ∈ E.children do
            if filter(E', O_q) then
                Q.insert_first(E');
            end
        end
    end
end
return refine(C);
```

Algorithm 2: Depth-First Spatial Query Processing

function $filter()$ is to check whether the MBB of the node and the query region overlaps, while $refine()$ is to examine whether the candidate actually overlaps with the query region. As shown in figure 2.1, only the spatial objects n_1, n_2, n_3, n_4, n_5 and n_6 need to be passed to the refine process, since the MBBs of the other objects do not overlap with *QR* and would thus be eliminated in the filter step.

2.5.2 P-query

2.5.2.1 Querying trajectories by a given point

This kind of query aims to find a set of trajectories whose nearest distance to a given query point is below a certain threshold (i.e., range query) or belongs to the top-k (i.e., k nearest neighbor query), based on some distance function. To answer such a query efficiently, one can index either the line segments or the sampled positions of trajectories using any multidimensional indexing structure (e.g. R-trees), and then search for the line segments or sampled points in the filter-and-refine paradigm. But it is worth noting a subtle difference compared to the traditional query processing for point objects. Whenever we encounter a line segment or a point, the corresponding trajectory should be marked as "visited" so that it will not be processed repeatedly when the other parts of the same trajectory are also reached.

2.5.2.2 Querying points by a given trajectory

A well known query that falls into this category is the Continuous Nearest Neighbor Search [35]. Given a set $\mathscr{P}$ of points, and a line segment $q = \overline{s,e}$, a continuous nearest neighbor (CNN) query retrieves the nearest neighbor from $\mathscr{P}$ for every position on q. The resulting output by such a query contains a set of $\langle R,T\rangle$ tuples, where R is a point in $\mathscr{P}$, and T is an interval of $\overline{s,e}$ within which R is the nearest neighbor of q. Figure 2.8 shows an example of the CNN query. Let $\mathscr{P} = \{a,b,c,d,f,g,h\}$ be the set of points to be considered. Given a query line segment $\overline{s,e}$, the CNN returns $\{\langle a,\overline{s,s_1}\rangle, \langle c,\overline{s_1,s_2}\rangle, \langle f,\overline{s_2,s_3}\rangle, \langle h,\overline{s_3,e}\rangle\}$ as the result. The objective of a CNN query is to split the line segments such that each of them consists of only one nearest neighbor.

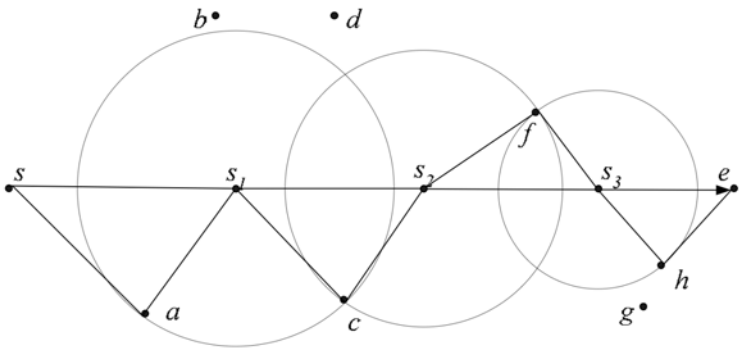


Fig. 2.8 Example of Continuous Nearest Neighbor[35]

Let *SL* be the list of split points. It can be seen that the start and end points of the line segment constitute the first and last elements in *SL*. To avoid multiple database scans, the strategy is to start with an initial *SL* that contains only two split points s and e with their covering points set, and then incrementally updates *SL* during query processing.

As illustrated in figure 2.9, the initial *SL* consists of s and e only. The nearest neighbor of the segment starting with s must be a nearest neighbor of s. The process starts by finding the nearest neighbor (i.e. a) of s and then draws a circle centered at s and e with the radius of $\overline{s,a}$ and $\overline{e,a}$, respectively. At this stage, only the data points within the circle need to be processed, because the data points outside the circles have greater distances to the line segment than a. The next step is to find the nearest neighbor of e from the data points within the circles and then draw a circle centered at e with radius $\overline{e,c}$. After that, the perpendicular bisector of $\overline{a,c}$ is computed and intersects with $\overline{s,e}$ at s_1. The above process will repeat, i.e., finding the nearest neighbor of s_1 and performing the perpendicular bisection, until no new point within the circles becomes the nearest neighbor again.

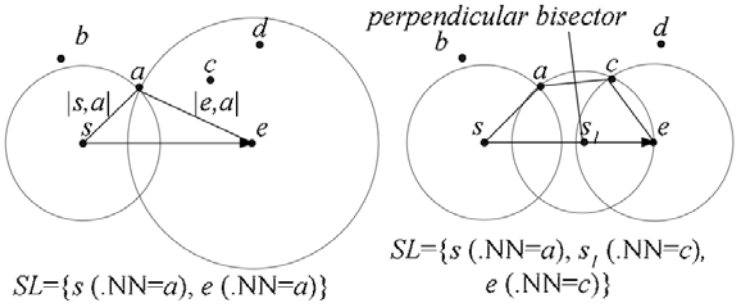


Fig. 2.9 Updating SL [35]

2.5.3 T-Query

Given a trajectory, a T-query finds trajectories that satisfy a given distance function to a query trajectory. One of the approaches for addressing this type of query is to treat the query trajectory as multiple points.

2.5.3.1 Finding trajectories from multiple points [10]

Given multiple query points, a k-BCT query finds k closest trajectories to the set of query points. The distance $Dist_q$ between a query location q_i and a trajectory $R = \{p_1, p_2, \ldots, p_l\}$ is defined as:

$$Dist_q(q_i, R) = \min_{p_j \in R} \{Dist_e(q_i, p_j)\} \tag{2.2}$$

Given a set of query locations Q, the distance between Q and a trajectory R is defined as follows:

$$Dist(Q, R) = \sum_{i=1}^{m} e^{-Dist_q(q_i, R)} \tag{2.3}$$

The approach adopted to evaluate this query is to search the nearby trajectories for each query location separately by using the k-Nearest Neighbor search [27, 7] algorithm, and then merge the results for the exact k-BCT [10]. Given a set of query locations $\{q_1, q_2, \ldots, q_m\}$, first of all the λ-NN of each query location is retrieved:

$$\lambda - NN(q_1) = \{p_1^1, p_1^2, \ldots, p_1^\lambda\}$$

$$\lambda - NN(q_2) = \{p_2^1, p_2^2, \ldots, p_2^\lambda\}$$

$$\ldots$$

$$\lambda - NN(q_3) = \{p_3^1, p_3^2, \ldots, p_3^\lambda\}$$

The set of scanned trajectories that contain at least one point in λ-NN(q_i) form a candidate set C_i for the k-BCT results. Note the cardinality $|C_i| \leq \lambda$, as there may be several λ-NN points belonging to the same trajectory. By merging the candidate sets generated by all the λ-NN(q_i), we get totally f different trajectories as candidates:

$$C = C_1 \cup C_2 \cup \cdots \cup C_m = \{R_1, R_2, \ldots, R_f\}$$

For each trajectory $R_x (x \in [1, f])$ within C, it must contain at least one point whose distance to the corresponding query location is determined. For example, if $R_x \in C_i (C_i \subseteq C)$, then the λ-NN of q_i must include at least one point of R_x, and the shortest distance from R_x to q_i is known. Therefore, at least one matched pair of points between R_x and some q_i can be discovered, and then a lower bound LB of the distance for each candidate $R_x (x \in [1, f])$ can thereafter be computed by using the found matched pairs:

$$LB(R_x) = \sum_{i \in [1,m] \wedge R_x \in C_i} \left(\max_{j \in [1,\lambda] \wedge p_i^j \in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} \right) \tag{2.4}$$

Here $\{q_i | i \in [1,m] \wedge R_x \in C_i\}$ denotes the set of query locations that have already been matched with some point on R_x, and the p_i^j which achieves the maximum $e^{-Dist_e(q_i, p_i^j)}$ with respect to q_i is the point on R_x that is closest to q_i, i.e., $\max_{j \in [1,\lambda] \wedge R_x \in C_i (e^{-Dist_q(q_i, R_x)})}$. Obviously it is not greater than $\sum_{i=1}^{m} e^{-Dist_q(q_i, R_x)}$, because it only takes those matched pairs found so far into account. Thus $LB(R_x)$ must lower bound the exact distance $Dist(Q, R_x)$ defined in equation 2.3. On the other hand if $R_x \notin C_i$, then none of the trajectory points has been scanned by $\lambda - NN(q_i)$ yet.

The upper bound UB of the distances for candidate trajectories in C can be derived by following the same rational:

$$UB(R_x) = \sum_{i \in [1,m] \wedge R_x \in C_i} \left(\max_{j \in [1,\lambda] \wedge p_i^j \in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} \right) + \sum_{i \in [i,m] \wedge R_x \notin C_i} (e^{-Dist_e(q_i, p_i^\lambda)}) \tag{2.5}$$

Having both the lower and upper bound, a filter and refine strategy can be adapted by using λ-NN to find the results. In case that λ-NN returns enough candidates for finding the k nearest trajectories to the query point, the refine process will be invoked to evaluate the exact result. Otherwise, λ is modified to λ^2 in order to include more candidates.

2.5.4 R-Query

R-query searches for trajectories or trajectory segments that belong to a specified space-time window. This type of query can be further classified into timestamp and time window (interval) queries. In general, timestamp queries find the spatial objects

within a region at a particular time instant in the past, whereas time window queries find spatial objects within a region for a period of continuous time instants.

2.5.4.1 R-Query with multi-version R-trees

There are many indexing structures that organize the movements of spatial objects by building an R-tree for each timestamp. In this section, we choose HR+-tree [33] as a representative for processing this type of query.

For timestamp queries, the process search is firstly directed to the root whose jurisdiction interval covers the timestamp. After that, it proceeds to the appropriate branches considering both the spatial and temporal extents (lifespan). The processing of interval queries is more complicated; since a node can be shared by multiple branches, it may be visited many times during the search. Sometimes, it is not necessary to traverse the same node if it has already been visited.

One approach to avoid duplicate visits is to store the old spatial extents into the new entry. However, storing such information, will significantly lower the fanout of the R-tree. Another solution is to perform (interval) queries in a breadth-first manner. Specifically, we start with the set of roots whose associated logical trees will be accessed. By examining the entries in these nodes, we can decide the nodes that need to be visited at the next level. Instead of accessing these nodes immediately, we save their block addresses and check for duplicates. Only when we have finished all the nodes at this level, will the ones at the next level be searched via the address information saved. Obviously the duplicate visits will be naturally avoided in this way.

2.5.4.2 R-Query with 3D R-trees

With 3D R-tree, the timestamp and interval queries can be treated as 3D range queries. A timestamp query is to find the spatial objects that overlap with a circle parallel to the x-y plane, while an interval query is to find spatial objects that are contained by a cylinder. In such cases, traditional spatial query processing approaches can then be adopted to answer these queries.

2.6 Summary

In trajectory databases, trajectory queries play an important role in complex analysis in various applications. According to the spatial data types involved, the trajectory queries can be classified into three types: point related trajectory query (P-query), region related trajectory query (R-query) and trajectory related trajectory query (T-query). The evaluation of trajectory queries has to address problems from two aspects. The first aspect is on effectiveness where a proper definition of dis-

tance measure between spatial objects is essential in order to accurately capture the spatiotemporal relationships. The second aspect is about efficiency which requires a proper index to speed up the query processing. This chapter introduces fundamental knowledge about these two aspects and discusses the processing techniques for different types of queries. These techniques and knowledge form the background for further study of this book.

References

1. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. FODO pp. 69–84 (1993)
2. Beckmann, N., Kriefel, H., Schneider, R., Seeger, B.: The r^* tree: An efficient and robust access method for points and rectangles. In 9th ACM-SIGMOD Symposium n Principles of Database Systems **6**(1), 322–331 (1990)
3. Brinkhoff, T., Kriegel, H.P., Seeger, B.: Efficient processing of spatial joins using r-trees. ACM SIGMOD Conference pp. 237–246 (1993)
4. Chakka, V.P., Everspaugh, A., Patel, J.M.: Indexing large trajectory data sets with seti. In Proc. of the Conf. on Innovative Data Systems Research(CIDR) (2003)
5. Chen, L.: Robust and fast similarity search for moving object trajectories. VLDB
6. Chen, L., Ng, R.: On the marriage of lp-norms and edit distance. In: VLDB, pp. 792–803 (2004)
7. Chen, L., Ozsu, M.T., Oria, V.: Robust and fast similarity search for moving object trajectories. SIGMOD (2005)
8. Chen, Z., Shen, H.T., Zhou, X.: Discovering popular routes from trajectories. ICDE (2011)
9. Chen, Z., Shen, H.T., Zhou, X., Yu, J.X.: Monitoring path nearest neighbor in road networks. SIGMOD (2009)
10. Chen, Z., Shen, H.T., Zhou, X., Zheng, Y., Xie, X.: Searching trajectories by locations - an efficiency study. SIGMOD (2010)
11. Frentzos, E., Gratsias, K., Pelekis, N., Theodoridis, Y.: Algorithms for nearest neighbor search on moving object trajectories. Geoinformatica **11**(2), 159–193 (2007)
12. Gonzalez, H., Han, J., Li, X., Myslinska, M., Sondag, J.P.: Adaptive fastest path computation on a road network: a traffic mining approach. VLDB (2007)
13. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, SIGMOD '84, pp. 47–57. ACM, New York, NY, USA (1984). DOI http://doi.acm.org/10.1145/602259.602266. URL http://doi.acm.org/10.1145/602259.602266
14. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. TODS **24**(2), 265–318 (1999)
15. Huang, Y.W., Jing, N., Rundensteiner, E.A.: Spatial joins using r-trees: Breadth-first traversal with global optimizations. VLDB **24**(2), 396–405 (1997)
16. Jeung, H., Liu, Q., Shen, H.T., Zhou, X.: A hybrid prediction model for moving objects. ICDE (2008)
17. Jeung, H., Yiu, M.L., Zhou, X., Jensen, C.S., Shen, H.T.: Discovery of convoys in trajectory databases. VLDB (2008)
18. Lee, J.G., Han, J., Li, X., Gonzalez, H.: Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. PVLDB **1**(1), 1081–1094 (2008)
19. Lee, J.G., Han, J., Whang, K.Y.: Trajectory clustering: A partition-and-group framework. SIGMOD (2007)
20. Lee, J.G., Han, J., Whang, K.Y.: Trajectory clustering: a partitionand-group framework. SIGMOD (2007)

21. Li, X., Han, J., Lee, J.G., Gonzalez, H.: Traffic density-based discovery of hot routes in road networks. SSTD (2007)
22. Lomet, D., Salzberg, B.: The performance of a multiversion access method. SIGMOD (1990)
23. Monreale, A., Pinelli, F., Trasarti, R., Giannotti, F.: Wherenext: a location predictor on trajectory pattern mining. SIGKDD (2009)
24. Nascimento, M., Silva, J.: Towards historical r-trees. In: Proceedings of the 1998 ACM symposium on Applied Computing, pp. 235–240. ACM (1998)
25. Pfoser, D., Jensen, C., Theodoridis, Y.: Novel approaches to the indexing of moving object trajectories. VLDB (2000)
26. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches in query processing for moving object trajectories. VLDB pp. 395–406 (2000)
27. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: Acm Sigmod Record, vol. 24, pp. 71–79. ACM (1995)
28. Sacharidis, D., Patroumpas, K., Terrovitis, M., Kantere, V., Potamias, M., Mouratidis, K., Sellis, T.: On-line discovery of hot motion paths. EDBT (2008)
29. Shang, S., Deng, K., Xie, K.: Best point detour query in road networks. ACM GIS (2010)
30. Shekhar, S., Yoo, J.S.: Processing in-route nearest neighbor queries: a comparison of alternative approaches. ACM GIS (2003)
31. Song, Z., Roussopoulos, N.: Seb-tree: An approach to index continuously moving objects. Proceedings of International Conference of Mobile Data Management (2003)
32. Tao, Y., Faloutsos, C., Papadias, D., Liu, B.: Prediction and indexing of moving objects with unknown motion patterns. SIGMOD (2004)
33. Tao, Y., Papadias, D.: Efficient historical r-trees. In: ssdbm, p. 0223. Published by the IEEE Computer Society (2001)
34. Tao, Y., Papadias, D.: Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In: VLDB, pp. 431–440 (2001)
35. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: Proceedings of the 28th international conference on Very Large Data Bases, pp. 287–298. VLDB Endowment (2002)
36. Wang, L., Zheng, Y., Xie, X., Ma, W.Y.: A flexible spatio-temporal indexing scheme for large-scale gps track retrieval. MDM (2008)
37. Xu, X., Han, J., Lu, W.: Rt-tree: An improved r-tree indexing structure for temporal spatial databases. In: Int. Symp. on Spatial Data Handling
38. Yi, B.K., Jagadish, H., Faloutsos, C.: Efficient retrieval of similar time sequences under time warping. ICDE (1998)
39. Zheng, Y., Zhang, L., Xie, X., Ma, W.Y.: Mining interesting locations and travel sequences from gps trajectories. WWW (2009)
40. Zhou, P., Zhang, D., Salzberg, B., Cooperman, G., Kollios, G.: Close pair queries in moving object databases. Proceedings of ACM GIS (2005)