A New Signature-Based Indexing Scheme for Efficient Trajectory Retrieval in Spatial Networks

SUMMARY

Even though it is very important to retrieve similar trajectories with a given query trajectory, there has been a little research on trajectory retrieval in spatial networks, like road networks. In this paper, we propose an efficient indexing scheme for retrieving moving object trajectories in spatial networks. For this, we design a signature-based indexing scheme for efficiently dealing with the trajectories of current moving objects as well as for maintaining those of past moving objects. In addition, we provide an insertion algorithm for storing the segment information of a moving object trajectory as well as a retrieval algorithm to find a set of moving objects whose trajectories match the segments of a query trajectory. Finally, we show that our signature-based indexing scheme achieves at least twice better performance on trajectory retrieval than the leading trajectory indexing schemes, such as TB-tree, FNR-tree, and MON-tree.

key words: signature-based indexing scheme, similar trajectory retrieval, spatial network

1. Introduction

Spatial databases have been extensively studied in the last two decades, resulting in the development of numerous spatial data models, query processing techniques, and index structures for spatial data [1]. Most of the existing works consider Euclidean spaces, where the distance between two objects is determined by an ideal shortest path between them in the space. However, in practice, objects can usually move on road networks, where the network distance is determined by the length of the real shortest path connecting two objects on the network. For example, the gas station nearest to a given point in Euclidean space may be located at a much greater distance in a road network than another gas station. Therefore, the network distance is an important measure in spatial network databases (SNDB). Recently, there have been some studies on SNDB to address emerging applications such as location-based service (LBS), Telematics, and vehicle relationship management (VRM) [2]–[6]. First, Speicy et al. [2] dealt with a computational data model for spatial network. Second, Shahabi et al. [3] presented k-nearest neighbors (k-NN) query processing algorithms for SNDB. Finally, Papadias et al. [4] designed a novel index structure for supporting query processing algorithms for SNDB.

Since moving objects move on predefined spatial networks instead of on Euclidean spaces, they exhibit similarity with respect to their trajectories, i.e., traversed paths [5]. Therefore, two moving objects can be correlated based on their trajectories. In addition, an analysis of moving object trajectories is essential for intelligent location-based services. For example, an intelligent ridesharing application may try to find out long and sharable patterns in the trajectories of moving objects [6]. Therefore, it is very important to retrieve similar trajectories with a given query trajectory for such applications. However, there has been little research on trajectory retrieval in spatial networks, like road networks.

In this paper, we propose an efficient indexing scheme for retrieving moving object trajectories in spatial networks. For this, we design a new signature-based indexing scheme for dealing with the trajectories of current moving objects as well as for maintaining those of past moving objects, in an efficient manner. In addition, we provide an insertion algorithm for storing the segment information of a moving object trajectory. We also provide a retrieval algorithm to find a set of moving objects whose trajectories match the segments of a query trajectory. The rest of the paper is organized as follows. In Sect. 2, we introduce related work. In Sect. 3, we propose a new signature-based indexing scheme for moving object trajectories. In Sect. 4, we present both algorithms to insert and retrieve trajectories in our signature-based indexing scheme. In Sect. 5, we provide the performance analysis of our signature-based indexing scheme. Finally, we provide conclusions and future work in Sect. 6.

2. Related Work

There has been a little research on trajectory indexing schemes for spatial networks. They can be largely classified into two approaches: tree-based and filtering-based ones. First, we overview a couple of tree-based leading trajectory index structures for Euclidean spaces as well as for spatial networks. Pfoser et al. [7] proposed a hybrid index structure, called TB-tree (Trajectory-Bundle tree), which preserves trajectories as well as allows R-tree for typical range search in Euclidean spaces. Although TB-tree has fast accesses to the trajectory information of moving objects, it has a couple of problems in SNDB. First, because moving objects move on a predefined spatial network in SNDB, the paths of moving objects are overlapped due to frequently used segments, like downtown streets. This leads to a large volume of overlap among the MBRs of internal nodes. Secondly, because the TB-tree constructs a three-dimensional MBR including time, the dead space for the moving object trajectory is highly increased in the case of a long time movement.
This leads to a large volume of overlap with other objects’ trajectories. Meanwhile, Frentzos [8] proposed a new indexing technique, called FNR-tree (Fixed Network R-tree) for objects constrained to move on fixed networks in two-dimensional space. The general idea of the FNR-tree is to construct a forest of 1-dimensional (1D) R-trees on the top of a 2-dimensional (2D) R-tree. The 2D R-tree is used to index the spatial data of the network, e.g., roads consisting of line segments, while the 1D R-trees are used to index the time interval of each object movement inside a given link of the network. The FNR-tree outperforms the R-tree in most cases, but it has a critical drawback that the FNR-tree has to maintain a tremendously large number of R-trees, thus leading to a great amount of storage overhead to maintain it. This is because the FNR-tree constructs as large number of R-trees as the total number of segments in the network. It has a critical drawback that the FNR-tree has to maintain a tremendously large number of R-trees, thus leading to a great amount of storage overhead to maintain it. This is because the FNR-tree constructs as large number of R-trees as the total number of segments in the network, being greater than 1 million in some cases. Finally, Almeida and Güting [9] proposed a new index structure for moving objects on network, called MON-tree, by using both edge-oriented and route-oriented models. The MON-tree can store the complete trajectories of the objects moving in the network and can answer two kinds of queries: the range query and the window query. The MON-tree outperforms the FNR-tree in terms of both updating and querying, but it has a drawback that the MON-tree should still maintain a very large number of R-trees, like the FNR-tree.

Next, we introduce a couple of filtering-based trajectory index structures for spatial networks. Chang et al. [10], [11] proposed a trajectory index structure for moving objects in Location-based Services (LBS), which made use of signatures for a filtering technique. They also provided a retrieval algorithm to find a set of moving objects whose trajectories match the segment of a given query trajectory. But, the trajectory index structure has a limitation that it focuses on handling the trajectories of only current moving objects. To deal with the trajectories of past moving objects as well as current ones, Chang et al. [12] proposed a signature-based indexing scheme for moving objects’ trajectories on spatial networks. The proposed signature-based indexing scheme can get access to moving objects’ trajectories through two entries, one for current moving objects and the other for past ones. However, the signature-based indexing scheme has two problems. First, as current objects are moved into past ones, its index structure for past object trajectories is continuously growing forever. So, it may lead to dramatic performance degradation on trajectory retrieval and may at last cause the operational failure of retrieving similar trajectories. Next, while current objects are moving on a road network, the signature-based indexing scheme maintains their trajectories in a secondary storage. So, it causes tremendous update cost for current object trajectories.

3. New Signature-Based Trajectory Indexing Scheme

3.1 Architecture of Trajectory Indexing Scheme

To design an efficient indexing scheme for moving object trajectories, the following requirements should be considered. First, because moving objects change their locations continuously on road networks, the amount of trajectory information for a moving object is generally very large. So it is necessary to handle a large amount of trajectory data in an efficient manner. Secondly, because the trajectories of moving objects are stored in a cumulative way along time, a user can issue a query to retrieve a moving object trajectory on any past time from a large amount of trajectory data. So it is necessary to deal with the trajectories of current moving objects as well as past objects in an effective manner. The current moving objects mean objects which are moving on a road network, and thus their trajectories are growing continuously. Whereas the past moving objects mean objects which have already moved on a road network, and thus their trajectories are fixed. Finally, because moving objects probably exhibit similarity with respect to their traversed paths, they can be correlated based on their trajectories. For example, LBS (location-based services) applications, like an intelligent ridesharing application, may try to find out long and sharable patterns in the trajectories of moving objects [6]. Therefore, it is very important to retrieve similar trajectories with a given query trajectory for LBS applications. However, there has been little research to meet the above three requirements. Because the TB-tree [7] fails to satisfy even the first requirement, it is not a proper index structure to deal with moving object trajectories. Though both of the FNR-tree [8] and MON-tree [9] meet the first and the second requirements, they do not support similar trajectory search for moving objects, and so they fail to meet the final requirement. Meanwhile, the filtering-based trajectory index structure [10], [11] meets the first and the third requirements, but it does not satisfy the second requirement.

To meet the above three requirements, our previous work presented a signature-based indexing scheme for moving objects’ trajectories on spatial networks [12]. However, the presented indexing scheme has two critical problems when it is applied to real applications, as mentioned in Sect. 2. To solve the problems, we propose a new signature-based indexing scheme by extending it in two folds. At first, in order to prevent the continuous growing of an index structure for past object trajectories, we provide two index structures for past object trajectories, i.e., one for the trajectories of past objects probably used by a user query and the other for the trajectories of past objects hardly used. We build a time-based B+-tree index structure only for the trajectories of probably used past objects, whereas the previous work builds an index structure for the trajectories of all the past objects. Secondly, in order to minimize the update cost of current object trajectories, we maintain current object trajectories in a main memory. Whereas, the previous work maintains them in a secondary storage. Figure 1 shows the structure of our new trajectory indexing scheme which can handle not only the trajectories of current moving objects, but also those of past moving objects, in such a way that it can be efficiently used for real applications. Our trajec-
of partitions between p and n are used for current ob-
...the trajectories of hardly used past objects, where i

\begin{equation}
A \text{ disc table's entry for a compact disc } j \text{ is } TD_{j} = \{e_{i}, \# \text{past } \text{seg}, \# \text{future } \text{seg}, (s_{ij}, e_{i}, \text{start, last, tb, te}) > \text{ where the first three entries mean an identifier for } MO_{i}, \text{ the number of past segments}, \text{ and the number of expected future segments, respectively. In addition, } s_{ij}, e_{i}, \text{ start and last mean a } j\text{-th segment of } MO_{i}\text{'s trajectory, an edge ID for an edge covering } s_{ij}, \text{ a relative start location of } s_{ij} \text{ in the edge of } e_{i}, \text{ and a last location of } s_{ij}. \text{ te means a beginning time when entering into } s_{ij} \text{ and } te \text{ means an end time when leaving } s_{ij}. \text{ The location information area contains the location of an object trajectory stored in the trajectory information area. This allows for accessing the actual object trajectories corresponding to potential matches to satisfy a query trajectory in the signature information area. The location information area also allows for filtering out irrelevant object trajectories based on the time condition of a query trajectory because it includes the beginning time, the current time, and the end time for a set of object trajectories. Location information, } Li, \text{ for the trajectory of an object } MO_{i} \text{ is } Li = \langle MO_{id}, \#_p past \text{ seg}, \#_f future \text{ seg}, (s_{ij}, e_{i}, \text{start, last, tb, te}) \rangle > \text{ where } Li \text{ is a location stored for } MO_{i} \text{ in the trajectory information area and the last three entries mean the time when the first segment, the last segment, and the expected segment for } MO_{i} \text{ are inserted, respectively. The signature information area contains the signatures of moving object trajectories stored in the trajectory information area. We will explain how to generate a signature from an object trajectory in the next section.}

3.1.2 Signature Generation

The main idea of our trajectory indexing scheme is to create partitions which store the fixed number of moving object trajectories together with the signatures generated from them. There are a couple of reasons for using partitions. First, because a partition is created and maintained depending on its beginning time, it is possible to efficiently retrieve the trajectories of moving objects on a given time. Next, because a partition can be accessed independently to answer a trajectory-based query, it is possible to achieve better retrieval performance by searching partitions in a parallel way.

A partition can be divided into three areas; trajectory information, location information, and signature information. The trajectory information area maintains moving object trajectories which consist of a set of segments (or edges). Trajectory information, Ti for a moving object MO_{i} can be represented as Ti = \langle MO_{id}, \#_p past \text{ seg}, \#_f future \text{ seg}, (s_{ij}, e_{i}, \text{start, last, tb, te}) \rangle > \text{ where the first three entries mean an identifier for } MO_{i}, \text{ the number of past segments}, \text{ and the number of expected future segments, respectively. In addition, } s_{ij}, e_{i}, \text{ start and last mean a } j\text{-th segment of } MO_{i}\text{'s trajectory, an edge ID for an edge covering } s_{ij}, \text{ a relative start location of } s_{ij} \text{ in the edge of } e_{i}, \text{ and a last location of } s_{ij}. \text{ te means a beginning time when entering into } s_{ij} \text{ and } te \text{ means an end time when leaving } s_{ij}. \text{ The location information area contains the location of an object trajectory stored in the trajectory information area. This allows for accessing the actual object trajectories corresponding to potential matches to satisfy a query trajectory in the signature information area. The location information area also allows for filtering out irrelevant object trajectories based on the time condition of a query trajectory because it includes the beginning time, the current time, and the end time for a set of object trajectories. Location information, } Li, \text{ for the trajectory of an object } MO_{i} \text{ is } Li = \langle MO_{id}, Li, \text{begin } time, current \text{ time, end } time > \text{ where } Li \text{ is a location stored for } MO_{i} \text{ in the trajectory information area and the last three entries mean the time when the first segment, the last segment, and the expected segment for } MO_{i} \text{ are inserted, respectively. The signature information area contains the signatures of moving object trajectories stored in the trajectory information area. We will explain how to generate a signature from an object trajectory in the next section.}

To create a signature from a given object trajectory, we make use of a superimposed coding technique [13] because it is very suitable for SNDB applications where the number of segments for a moving object trajectory can be variable. When the total number of object trajectories is N and the average number of segments per object trajectory is r, the optimal values for both the size of a signature in bits (S) and the number of bits to be set per segment (k) [14] can be calculated as ln Fd = -(ln 2)2 * S/r and k = S * ln 2/r. Here
we assume that $F_d$ is 1/N where $F_d$ is false drop probability that a trajectory signature seems to qualify, given that the corresponding object trajectory does not actually qualify.

**Definition 3.1:** Let a moving object trajectory in a spatial network be $T = \{s_1, s_2, \ldots, s_n\}$ where $s_i$ is a segment (edge) composing $T$ and $1 \leq i \leq n$. The signature of a moving object trajectory ($T$) can be obtained by superimposing (OR-ing) the signatures of all the segments ($s_i, 1 \leq i \leq n$) of $T$.

For example, a moving object trajectory consists of three segments, like $T_1 = \{s_1, s_2, s_3\}$. We assume that the signatures of $s_1$, $s_2$, and $s_3$ are 01000001, 00110000, and 01000010 where $S$ is 8 bits and $k$ is 2. Thus the signature of $T_1$ is obtained by superimposing them as follows.

- signature of $s_1$: 01000001
- signature of $s_2$: 00110000
- signature of $s_3$: 01000010

Using a signature generated for an object trajectory, we can support partial-match retrieval.

**Definition 3.2:** Let a query trajectory in a spatial network be $Q = \{s_1, s_2, \ldots, s_m\}$ where $s_i$ is a segment composing $Q$ and $1 \leq i \leq m$. If the signature of $Q$ is contained by the signature of $T$ in bit positions set to 1, $Q$ has a possibility of partially matching with $T$.

For example, we assume that a query trajectory $Q$ consists of $s_2$ and $s_3$. Thus the signature of $Q$ is obtained by superimposing both signatures of $s_2$ and $s_3$, that is, 01110011. Because the bit positions set to 1 in the signature of $Q$, i.e., 2, 3, 4, and 7, are also set to 1 in the signature of $T$, the signature of $Q$ is contained by the signature of $T$ in bit positions set to 1. Thus $Q$ has a possibility of the partial match with $T$ and actually $Q$ partially matches with $T$.

As a result, our new trajectory indexing scheme can meet all of the above three requirements. Firstly, because our indexing scheme is a signature-based technique, it is not affected by the overlap of moving objects’ paths and never causes the dead space problem, thus satisfying the first requirement. Secondly, our indexing scheme maintains partitions for past objects’ trajectories by using a time-based B-tree, it can efficiently retrieve the trajectories of moving objects on any past time. So, our scheme meets the second requirement. Finally, since our indexing scheme generates signatures by using the superimposed coding, it is capable of supporting partial-match trajectory retrieval, thus leading to the fulfillment of the final requirement.

### 3.2 Partition Movement

To answer trajectory-based queries for past time, it is necessary to efficiently search the trajectories of past moving objects which no longer move on road networks. The trajectories of moving objects can be divided into three groups; one for current object trajectories, another for past object trajectories being frequently used, and the last for past object trajectories being hardly used. Based on the groups, Fig. 2 shows an overall architecture for storing moving object trajectories, which consists of three structures: current object trajectory structure in a main memory (COSM), past object trajectory structure in a secondary storage (POSS), and past object trajectory structure in a tertiary storage (POST). When a current moving object trajectory in COSM is no longer changed due to the completion of the object movement, the object trajectory should be moved from COSM to POSS. Meanwhile, as a past object is getting older, its trajectory is hardly used for a trajectory-based query. Thus, the past object’s trajectory should be moved from POSS to POST. The POSS is stored into a secondary storage, like a disk, whereas POST is maintained in a tertiary storage, like a compact disc (CD-RW) or a magnetic tape.

#### 3.2.1 Partition Movement between COSM and POSS

To move current object trajectories from COSM to POSS, we should consider such requirements as retrieval of past object trajectories in an efficient way, accessing of the small number of partitions to answer a trajectory-based query, and construction of an efficient time-based index structure. To satisfy the first requirement, we make use of a bit-sliced method [13] for constructing a signature-based indexing scheme in POSS, instead of using a bit-string method in COSM. In the bit-sliced method, we create a fixed-length signature slice for each bit position in the original signature string. That is, we can store a set of the first bits of all the trajectory signatures into the first slice, a set of the second bits into the second slice and so on. When the number of segments in a query trajectory is $m$ and the number of bits assigned to a segment is $k$, the number of page I/O accesses for answering the query in the bit-sliced method is less than $k \times m$. Therefore, when the number of segments in a query trajectory is small, our indexing scheme requires the small number of page I/O accesses due to the small number of signature slices needed for the query.

To satisfy the second requirement, we should maintain all the partitions in POSS, using the following definition.

**Definition 3.3:** Let the beginning time of partition $i$ be $T(P_i)$ and the end time of partition $i$ be $E(P_i)$ where $P_i$ is maintained in POSS. If $T(P_i) < T(P_j)$, then $E(P_i) \leq E(P_j)$ where $i < j$.

If the condition of the definition 3 is not satisfied
among partitions in POSS, query processing may be inefficient depending on the time window distribution of partitions in POSS, even for queries with the same time window. For example, assuming that there are six partitions as shown in Fig. 3, three queries with the same time window can be answered by accessing two, four, and two partitions in POSS, respectively. Actually, if all the trajectories of the partition i have completed their movements earlier than those of the partition i − 1, the partition i should move from COSM to POSS earlier than the partition i − 1, thus leading to the dissatisfaction of the above condition. To prevent it, we require a strategy to store partitions such that if all the trajectories of the partition i are no longer changed, but those of the partition i − 1 are changed, we exchange the trajectories in the partition i − 1 with those having the smallest end time in the partition i and then move the partition i − 1 from COSM to POSS.

To satisfy the final requirement, we construct a time-based B+-tree by using the end time of a partition as a key, so as to have fast access to the partitions in POSS. Figure 4 shows the time-based B+-tree structure. A record of a leaf node in the time-based B+-tree has \( p_{\text{start\_time}}, p_{\text{end\_time}}, \text{Pid}, \text{PLoc} \geq t \) where the first two entries mean the smallest beginning time and the largest end time of the trajectories for a partition in POSS, respectively, and Pid and PLoc mean its partition ID and its location, respectively. When a query is issued to find object trajectories with a time window \([t1, t2]\), it first gets a starting leaf node by searching the time-based B+-tree using \( t1 \), and then obtains records to satisfy the condition, \( p_{\text{end\_time}} \geq t1 \) AND \( p_{\text{start\_time}} \leq t2 \). The search space for processing the query with \([t1, t2]\) ranges from \( p_{a} \) to \( p_{b} \). Here \( p_{a} \) is the leaf node obtained by searching the B+-tree with key = \( t1 \), and when we follow the leaf nodes of the sequence set from \( p_{a} \), \( p_{b} \) is the first leaf node that fails to satisfy the above condition. This allows for the minimum of page I/O accesses for answering the query.

3.2.2 Partition Movement between POSS and POST

To move past objects’ trajectories from POSS to POST, we should find past objects which are probably used by a user query. For this, we define PTD as follows.

**Definition 3.4:** Let PTD a past time duration from the current time for differentiating between POSS and POST. The trajectories of past objects created within PTD are probably used by a user query for computing similar trajectories, while those of past objects created out of PTD are hardly used by a user query.

We can define PTD depending on specific application scenario. For example, for an intelligent ridesharing application, a user may want to find some similar trajectories for a short period of time from the current time. For an intelligent travel time prediction application, a user usually wants to obtain the similar trajectories of past objects for a long period of time, e.g., one year, from the current time.

When partitions which are generated by past objects created out of PTD are resided in POSS, we first move those partitions from POSS to POST. Then, we remove their keys, i.e., the end times of the partitions, from the time-based B+-tree structure. At last, when there is room enough to insert the partitions moved to POST into the most recently created compact disc, we update the first entry of the disc table because the disc table’s entries are maintained in the reverse order of a compact disc creation time. When there is no room, we create a new compact disc and store the partitions moved to POST into it. Then, we insert into the disc table a new entry for the newly created compact disc. At this time, we check if the largest time of partitions for the last entry of the disc table is out of a disappearing past time duration from the current time (DPTD) or not. If it is true, we destroy the least recently created compact disc and delete the last entry for the compact disc from the disc table. We can also define DPTD depending on specific application scenario. Figure 5 shows the partition movement between COSM and POSS as well as that between POSS and POST. First, \( r \) partitions are newly created in COSM due to the insertion of new moving objects’ trajectories. Secondly, because all the trajectories of the partitions from \( p \) to \( p + q - 1 \) have no longer changed, \( q \) partitions are moved from COSM to POSS. Thirdly, because all the trajectories of the partitions from \( i + k \) to \( i + (k + 1) - 1 \) have been hardly used by a user query, \( i \) partitions are moved from POSS to POST and are stored into a compact disc \( k + 1 \). Finally, because the largest time of all the partitions in the compact disc \( 1 \) has been out of DPTD, the compact disc \( 1 \) is destroyed and thus the last entry of the disc table is deleted.
4. Algorithms for Moving Object Trajectories

Here we provide both a segment insertion algorithm and a trajectory retrieval algorithm.

4.1 Insertion Algorithm

We provide a segment insertion algorithm for current moving objects’ trajectories because only current moving objects make their trajectories to be stored as they move on a spatial network. The segment insertion algorithm first checks whether a segment to be stored is the first segment of a moving object’s trajectory or not. If it is the first one, the algorithm finds the last partition in the partition table and obtains an available entry (E) in the last partition. Then it stores the segment into the entry E of the trajectory information area in the last partition. The algorithm also stores begin_time (BeginT), current_time (CurrentT), and end_time (ExpectedET) into the entry E of the location information area in the last partition. Here both StartT and CurrentT are assigned to the beginning time of the first segment to be stored and ExpectedET is assigned to NULL. At last, the algorithm stores <MOid, Loc, StartT, CurrentT, ExpectedET > into the entry E of the location information area. It updates the CurrentT of the entry E in the location information area and the CurrentT of the partition P’s entry in the partition table. Meanwhile, the algorithm checks whether or not the number of segments in P is the maximum and all the objects in P have issued their last segments. If true, the algorithm calls a sub-algorithm (Insert_POST) to insert the partition P in main memory into POSS in the secondary storage. Finally, it stores the SigTS into the entry E of the signature information area in the partition P and increases the number of actual segments (#actual_seg) of E by one. Figure 6 shows a segment insertion algorithm for a moving object’s trajectory.

In order to insert a partition into POSS, the Insert_POST algorithm first opens both the signature file and the trajectory file which have been created, as shown in Fig. 7. Then it generates a chunk of bit-sliced signatures by transposing all the signatures of the signature information area in P, and stores the chunk to a location (SLoc) of the signature file. In addition, for each trajectory information record in the partition, the algorithm stores the record into the trajectory file and stores both its signature and its location into the signature file. Finally, the algorithm inserts SLoc into the time-based B+-tree with a key of the end time for the partition.

In order to insert into POST a set of partitions which are generated by past objects created out of PTD (i.e., SP), the Insert_POST algorithm first deletes SP in POSS and remove the keys of SP from the time-based B+-tree structure, as shown in Fig. 8. Then, it checks if there is room enough to store SP in the most recently created compact disc or not. If there is enough room, the Insert_POST algorithm stores SP in the compact disc and updates the first entry of the disc table. Otherwise, the algorithm creates a new compact disc and stores SP in the newly created compact disc. Then, it inserts into the disc table a new entry for the newly created compact disc. At last, the algorithm checks if the largest time of partitions for the last entry of the disc table is out of DPTD or not. If it is true, the algorithm deletes the last entry from the disc table and destroys the least recently created compact disc.

4.2 Retrieval Algorithm

The retrieval algorithm for moving object trajectories finds a set of objects whose trajectories match the segments of a query trajectory. For this, it is necessary to find a set of partitions satisfying the time interval (TimeRange) represented by < lower, upper > of a given query (Q). Figure 9 shows a partition finding algorithm (Find_partition) to generate a list of partitions (partitionList), given a time range (TimeRange). Based on the TimeRange, the algorithm searches both the trajectories of current moving objects by accessing the partition table of COSM and those of past moving objects by using the time-based B+-Tree of POSS. The search cases can be determined by comparing the TimeRange (T) with the p_end_time (PEtime) of the last partition in POSS as well.
Algorithm Segment_Insertion(MOi, TrajSeg, ST)
/* TrajSeg contains a segment of a trajectory for a moving object (MOi) to be stored with an object trajectory’s start time, ST */
1. Generate a signature SigTS from TrajSeg
2. if (TrajSeg is the first segment of its trajectory) {
3. BeginT = CurrentT = th of TrajSeg
4. Obtain final_entry_no of the entry for the last partition, P, in the partition table
5. E = final_entry_no + 1 // E is the next entry available in P
6. Obtain the location, Loc, of the entry E in the trajectory info area for inserting object trajectory
7. ExpectET = NULL
8. Store <MOi, T, TrajSeg> into the entry E, pointed by Loc, of the trajectory information area in P
9. Store <MOi, T, BeginT, CurrentT, ExpectET> into the entry E of the location information area in P
10. Store <StartT, CurrentT, ExpectET, NE> into the entry for P in the partition table }
11. else {
12. Locate a partition P covering ST in partition table
13. Locate an entry E covering ST for MOi in the location info area and get its location, Loc, in the trajectory info area
14. Obtain #actual seg of the trajectory info entry E for the MOi in P and increment #actual seg
15. Insert TrajSeg into #actual seg+1-th trajectory segment of E
16. SigTS = SigTS | signature stored in E of the signature info area
17. CurrentT = te of TrajSeg
18. Store CurrentT into the current_time of the entry E of the location info area and store CurrentT into the p_current_time of the partition P entry in the partition table
19. if(TrajSeg is the last segment of its trajectory) {
20. Check a condition on whether # of objects in P is the maximum and every object has issued the last segment
21. if(Condition is True) call Insert_POSS(P) }
22. } //else
23. Store SigTS into the entry E of the signature info area in P
24. Increase #actual seg of E by one
End Segment_Insertion

Fig. 6 Segment insertion algorithm for moving objects.

Algorithm Insert_POSS(P)
/* P is a partition to be stored into POSS */
1. Open both the signature file and the trajectory file
2. Generate bit-sliced signatures by transposing all the signatures of the signature information area in P
3. Append the bit-sliced signatures for P to the signature file and get the start location to be stored, SLoc
4. Append the location information for P to the signature file
5. for (each trajectory information record in P) {
6. Append the record to the trajectory file and get the start location to be stored, TLoc, and the record length, Len
7. Append a pair of TLoc and Len to the signature file
8. } //for
9. Insert SLoc into a time-based B+tree with a key of ExpectET for P
End Insert_POSS

Fig. 7 Partition insertion algorithm into POSS.

as that with the p_start_time (CStime) of the first partition in COSM as follows.
1. If T.lower > PEtime, both T.lower and T.upper are ranged in COSM
2. If T.upper ≤ PEtime AND T.upper < CStime, both T.lower and T.upper are ranged in POSS
3. If T.lower ≤ PEtime AND T.upper ≥ CStime, the range of T.lower and T.upper are within both COSM and POSS simultaneously

Fig. 8 Partition insertion algorithm into POST.

Algorithm Insert_POST(Sp)
/* Sp is a set of partitions being generated by past objects created out of PTD */
1. Delete Sp in POSS
2. Remove the keys of Sp from the time-based B+tree structure
3. Check if there is room enough to store Sp in the most recently created compact disc or not
4. if (there is enough room) {
5. Store Sp in the most recently created compact disc
6. Update the first entry of the disc table }
7. else {
8. Create a new compact disc
9. Store Sp in the newly created compact disc
10. Insert a new entry for the newly created compact disc into the disc table
11. Let LTP be the largest time of partitions for the last entry of the disc table
12. if (LTP is out of DPTD) {
13. Delete the last entry from the disc table
14. Destroy the least recently created compact disc }
15. } //else
End Insert_POST

Fig. 9 Partition finding algorithm.

T.lower and T.upper are ranged in POSS
3. If T.lower ≤ PEtime AND T.upper ≥ CStime, the range of T.lower and T.upper are within both COSM and POSS simultaneously

For the first case, it performs the sequential search of the partition table in COSM and finds a list of partitions (partList) to satisfy a condition, i.e., end_time (or current_time) ≥ T.lower AND start_time ≤ T.upper. Because the partition table of COSM is resident in a main memory, the cost for searching the partition table is low. For the second case, it finds a starting leaf node by searching the time-based B+tree with key = t1 and obtains partList to satisfy the above condition by searching the next leaf nodes from the starting node in the sequence set. For the last case, it gets two lists of partitions to satisfy
Algorithm Retrieve(QSegList, TimeRange, MOidList) /* MOidList is a set of ids of moving objects containing a set of query segments, QSegList, for a given range time, TimeRange */
1. Qsig = 0, #qseg = 0, partList = Ø
2. t1 = TimeRange.lower, t2 = TimeRange upper
3. for each segment QSj of QSegList { 
4.  Generate a signature QSSt from QSj
5.  QSig = QSig | QSSt, #qseg = #qseg + 1 
6.  call Find_partition(TimeRange, partList)
7.  for each candidate entry Ek of CanList { 
8.     Obtain a set of candidate entries, CanList, examining the signatures of signature information area in Ek
9.     for each candidate entry Ek of CanList { 
10.    Let s,e,c be start_time, end_time, current_time of the entry Ek of location information area
11.   #matches = 0
12.   Obtain the first segment ESj of the entry Ek of the trajectory information area, TEk, and obtain the first query segment QSj of QSegList
13.   while(RESi # NULL and QSj # NULL) { 
14.      if(#matches = #matches + 1) 
15.          Obtain the next segment ESj of TEk
16.          Obtain the first segment QSj of QSegList 
17.         #matches = #matches + 1
18.         if(#matches >= #qseg) MOidList = {TEk’s MOid} 
19.   } // end of while
20. } // end of for - partList
End Retrieve

Fig. 10 Retrieval algorithm for moving object trajectories.

the TimeRange in both COMS and POSS. It finally obtains partList by merging the two lists of partitions acquired from both COSM and POSS.

By using the partition finding algorithm, Fig. 10 shows a retrieval algorithm (i.e., Retrieve) which finds a set of objects whose trajectories match a query trajectory. It generates a query signature (QSig) from a query trajectory’s segments. Next, it calls the partition finding algorithm to obtain parList satisfying TimeRange. For each partition of the partition list, it searches the signatures in the signature information area and acquires a list of candidates (CanList). For the entries corresponding to the candidates, it determines whether or not their start_time, end_time, and current_time satisfy a condition, i.e., end_time (or current_time) ≥ T.lower AND start_time ≤ T.upper. Finally, it determines whether or not the query trajectory matches the object trajectories corresponding to the candidate entries. If true, the algorithm inserts the object’s ID into a result list (MOidList).

5. Performance Analysis

We implement our trajectory indexing scheme under Pentium-IV 2.0 GHz CPU with 1 GB main memory. For our experiment, we use a road network consisting of 170,000 nodes and 220,000 edges [15]. To generate moving objects in the road network, we make use of Brinkhoff’s algorithm because it is the most widely used for the experiments of spatial network databases [16]. The Brinkhoff’s algorithm provides the network-based approach, which generates moving objects uniformly in a road network, as well as the region-based approach, which generates skewed data to simulate a real-word application, as shown in Fig. 11. For our experiment, we generate both 100,000 moving objects based on the network-based approach and 100,000 moving objects based on the region-based approach.

For our performance comparison, we compare our indexing scheme with TB-tree, FNR-tree and MON-tree whereas we exclude the previous filtering-based trajectory index structures. This is because the trajectory index structure [10], [11] has a limitation that it focuses on handling the trajectories of only current moving objects while the signature-based indexing scheme [12] have a critical problem that its index structure for past object trajectories is continuously growing forever. We do our performance analysis in terms of insertion time, storage overhead, and retrieval time. First, Table 1 shows insertion performance to store the trajectories of moving objects generated based on the network-based approach and the region-based approach, respectively. It is shown that our indexing scheme achieves over one order of magnitude better insertion performance than TB-tree, FNR-tree and MON-tree. This is because both FNR-tree and MON-tree construct extremely great number of R-trees. Secondly, we measure storage overhead over the size of 100,000 moving object trajectories, i.e., 105 Mbyte, as shown in Table 1. It is shown that the storage overhead of our indexing scheme is about one twentieth of those of TB-tree, FNR-tree and MON-tree.
The indexing scheme is constructed based on a signature file which basically needs a low storage overhead.

Next, we measure retrieval time for answering queries whose trajectory contains 2 to 20 segments, in case of using data generated based on the network-based approach as shown in Fig. 12. When the number of segments in a query is 2, it is shown that our indexing scheme requires about 0.06 sec while MON-tree, FNR-tee and the TB-tree needs 0.10, 0.12, and 8.5 sec, respectively. Thus, our indexing scheme shows about twice better performance than MON-tree and FNR-tee, while it achieves two order of magnitude better performances than TB-tree. The TB-tree achieves the worst retrieval performance due to a very large amount of overlap in its internal nodes. As the number of segments in queries increase, the retrieval time of our indexing scheme is very slightly increased while those of competitors are rapidly increased. The reason is why our indexing scheme creates a query signature combining all the segments in a query and it searches for potentially relevant trajectories of moving objects by using the query signature as a filter. When the number of segments in a query is 20, it is shown that our indexing scheme achieves three times better retrieval performance than MON-tree and FNR-tee while it achieves over two order of magnitude better performance than TB-tree. When the number of segments is 20, our indexing scheme achieves four times better retrieval performance than MON-tree, over one order of magnitude better than FNR-tee, and about three order of magnitude better than TB-tree. Because our indexing scheme is a signature-based indexing structure, it hardly depends on the data distribution. Whereas, because the competitors generate a large number of candidates in general when the number of segments in a query is great, they require more time to obtain a result set from a given query.

6. Conclusions

Because moving objects usually moves on spatial networks, instead of on Euclidean spaces, efficient index structures are needed to gain good retrieval performance on their trajectories. However, there has been a little research on trajectory indexing schemes for spatial network databases. Therefore, we proposed an efficient indexing scheme for moving objects’ trajectories in spatial networks. For this, we first designed a signature-based indexing scheme for efficiently dealing with the trajectories of current moving objects as well as maintaining those of past moving objects. In addition, we provided both an insertion algorithm for storing the segment information of a moving object trajectory and a retrieval algorithm for finding a set of moving objects whose trajectories match the segments of a query trajectory. Finally, we showed that our indexing scheme achieved at least twice better retrieval performance than the leading trajectory indexing schemes, such as TB-tree, FNR-tree, and MON-tree. As future work, we need to extend our indexing scheme to a parallel environment to achieve better retrieval performance, due to the characteristic of signature files [13].

Acknowledgments

This work was financially supported by the Ministry of Education, Science Technology (MEST) and Korea Industrial Technology Foundation (KOTEF) through the Human Resource Training Project for Regional Innovation. This work
was also supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. 2009-0059417).

References