An Efficient Indexing Scheme for Moving Objects' Trajectories on Road Networks*

Jae-Woo Chang and Jung-Ho Um

Dept. of Computer Eng., Chonbuk National Univ., Chonju, Chonbuk 561-756, Korea
jwchang@chonbuk.ac.kr, jhum@dblab.chonbuk.ac.kr

Abstract. Even though moving objects usually move on spatial networks, there has been little research on trajectory indexing schemes for spatial networks, like road networks. In this paper, we propose an efficient indexing scheme for moving objects' trajectories on road 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 both an insertion algorithm to store the initial information of moving objects' trajectories and one to store their segment information. We also provide a retrieval algorithm to find a set of moving objects whose trajectories match the segments of a query trajectory. Finally, we show that our indexing scheme achieves much better performance on trajectory retrieval than the leading trajectory indexing schemes, such as TB-tree and FNR-tree.

1 Introduction

Most of studies on spatial databases in the last two decades have considered Euclidean spaces, where the distance between two objects is determined by the ideal shortest path connecting them in the spaces [Se99]. 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, a gas station nearest to a given point in Euclidean spaces may be more distant 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 for emerging applications, such as location-based service (LBS) and Telematics [B02, PZM03, SJK03, SKS03]. First, Speicys et al. [SJK03] dealt with a computational data model for spatial network. Secondly, Shahabi et al. [SKS03] presented k-nearest neighbors (k-NN) query processing algorithms for SNDB. Finally, Papadias et al. [PZM03] designed a novel index structure for supporting query processing algorithms for SNDB.

Because moving objects usually move on spatial networks, instead of on Euclidean spaces, efficient index schemes are required to gain good retrieval performance on

* This work is financially supported by the Ministry of Education and Human Resources Development (MOE), the Ministry of Commerce, Industry and Energy (MOCIE) and the Ministry of Labor (MOLAB) though the fostering project of the Lab of Excellency.

J.X. Yu, M. Kitsuregawa, and H.V. Leong (Eds.): WAIM 2006, LNCS 4016, pp. 13 – 25, 2006.

their trajectories. However, there has been little research on trajectory indexing schemes for spatial networks, like road networks. In this paper, we propose an efficient indexing scheme for moving objects' trajectories on road 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 both an insertion algorithm to store the initial information of moving objects' trajectories and one to store their segment information. 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 Section 2, we introduce related work. In Section 3, we propose a signature-based indexing scheme for moving objects' trajectories. In Section 4, we provide the performance analysis of our indexing scheme. Finally, we draw our conclusions in Section 5.

2 Related Work

There has been little research on trajectory indexing schemes for spatial networks. So we overview both a predominant trajectory index structure for Euclidean spaces and a leading trajectory index structure for spatial networks. First, Pfoser et al. [PJT00] proposed a hybrid index structure which preserves trajectories as well as allows for R-tree typical range search in Euclidean spaces, called TB-tree (Trajectory-Bundle tree). The TB-tree has fast accesses to the trajectory information of moving objects, but it has a couple of problems in SNDB. Firstly, 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 case of a long time movement. This leads to a large volume of overlap with other objects' trajectories. Meanwhile, Frentzos [F03] 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 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 networks.

3 Efficient Indexing Scheme for Moving Objects' Trajectories

In this section, we will describe our indexing scheme not only for current trajectories of moving objects, but also for their past trajectories. In addition, we will present both insertion and retrieval algorithms for the trajectories of moving objects.

3.1 Indexing Scheme for Current Trajectories of Moving Objects

For indexing road networks, the TB-tree may lead to a large volume of overlap among the MBRs of its internal nodes. The FNR-tree usually leads to a great amount of storage overhead to maintain a great number of R-trees. To solve their problems, we propose a new signature-based indexing scheme which can have fast accesses to moving object trajectories. Figure 1 shows the structure of our trajectory indexing scheme.

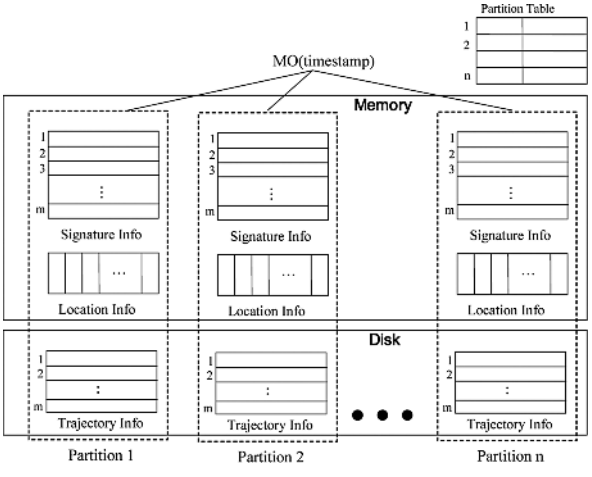

Fig. 1. Signature-based trajectory indexing scheme

The main idea of our trajectory indexing scheme is to create a signature of a moving object trajectory and maintain partitions which store the fixed number of moving object trajectories and their signatures together in the order of their start time. There are a couple of reasons for using partitions. First, because a partition is created and maintained depending on its start 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 parallel. As a result, our trajectory indexing scheme has three advantages. First, our indexing scheme is not affected by the overlap of moving objects' paths and never causes the dead space problem because it is not a tree-based structure like TB-tree. Secondly, our indexing scheme well supports a complex query containing a partial trajectory condition since it generates signatures using a superimposed coding. Finally, our indexing scheme can achieve very good retrieval performance because it can be easily adapted to a parallel execution environment.

Our trajectory indexing scheme consists of a partition table and a set of partitions. A partition can be divided into three areas: trajectory information, location information, and signature information. A partition table maintains a set of partitions which store trajectories for current moving objects. The partition table is resided in the main memory due to its small size. To answer a user query, we find partitions to be accessed by searching the partition table. An entry Ei for a partition i is Ei = <p_start_time, p_end_time, p_expected_time, final_entry_no> where p_start_time,

p_current_time, and p_end_time are the smallest start time, the largest current time, the largest end time of all the trajectories, respectively, and final_entry_no means the last entry number in a partition i. The trajectory information area maintains moving object trajectories which consist of a set of segments (or edges). A trajectory Ti for an object MOi is Ti =<MOi_{id}, #_past_seg, #_future_seg, #_mismatch, {sij,eid, start,end,ts,(te or v)}> where #_past_seg, #_future_seg, and #_mismatch are the number of past segments, expected future segments, and the number of mismatched segments between them, respectively. Here, sij and eid mean j-th segment of the trajectory for MOi and edge ID for an edge covering sij, respectively. Start and end mean the relative start and last location of sij in the edge of eid, respectively. ts, te, and v mean the start time, the end time, and the average speed of sij in the edge of eid, respectively. 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 start time, the current time, and the end time for a set of object trajectories. Location information, Ii, for the trajectory of an object MOi is Ii = <MOi_{id}, Li, strat_time, current_time, end_time > where Li is the location for MOi in the trajectory information area and start_time, current_time, and end_time mean the time when the first trajectory, the last segment, and the expected segment for MOi is inserted, respectively. To create a signature from a given object trajectory in an efficient manner, we make use of a superimposed coding because it is very suitable to SNDB applications where the number of segments for an object trajectory is variable [ZMR98]. In case the total number of object trajectories is N and the average number of segments per object trajectory is r, optimal values for both the size of a signature in bits (S) and the number of bits to be set per segment (k) can be calculated as ln $Fd = -(\ln 2)^2 * S/r$ and $k = S * \ln 2/r$ [FC84]. Here we assume that Fd (false drop probability that a trajectory signature seems to qualify, given that the corresponding object trajectory does not actually qualify) is 1/N. To achieve good retrieval performance, we store both the signature and the location information in the main memory.

3.2 Indexing Scheme for Past Trajectories of Moving Objects

To answer trajectory-based queries with a 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 two groups: one being frequently used for answering queries based on current object trajectories (COTSS) and the other for answering queries based on past object trajectories (POTSS). Figure 2 shows an overall architecture of indexing schemes for moving object trajectories. When a current moving object trajectory in COTSS is no longer changed due to the completion of the object movement, the object trajectory should be moved from COTSS to POTSS. The signature and the location information areas of COTSS are resided in the main memory for fast retrieval, whereas all of three areas of POTSS are maintained in the secondary storage. To move current object trajectories from COTSS to POTSS, we should consider three requirements: retrieval of past object trajectories in an efficient way, accesses 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 [ZMR98] for constructing a signature-based indexing scheme in POTSS, instead of using a bit-string method in COTSS. In the bit-sliced method, we create a fixed-length signature slice for each bit position in the original signature string. That is, we store a set of the first bit positions of all the trajectory signatures into the first slice, a set of the second bit positions 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*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. Figure 2 shows the movement of a partition from COTSS to POTSS. The partitions from 1 to i-1 have been moved to POTSS and k partitions are newly created in COTSS due to the insertion of new moving object trajectories. Because all the trajectories of the partition i-1 have no longer changed, the partition i-1 has just moved from COTSS to POTSS.

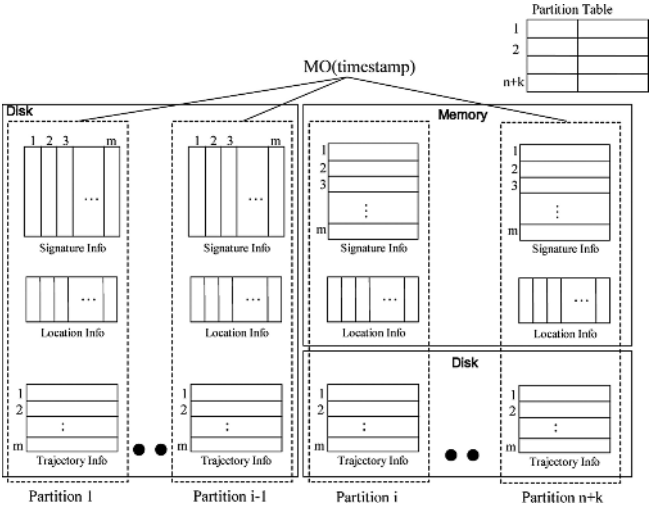

Fig. 2. Movement of partitions from COTSS to POTSS

To satisfy the second requirement, we maintain all the partitions in POTSS so that they can hold the condition that if start_time(partition i)<start_time(partition i+1), end_time(partition i) ≤end_time(partition i+1). If this condition is not satisfied among partitions in POTSS, query processing may be inefficient depending on the time window distribution of partitions in POTSS, even for queries with the same time window. For example, assuming that there are six partitions with their start and their end time as shown in Figure 3, three queries with the same time window can be answered by accessing two, four, and two partitions in POTSS, 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 COTSS to POTSS earlier than the

partition i-1, 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 trajectories being changed in the partition i-1 with those having the smallest end time in the partition I and then move the partition i-1 from COTSS to POTSS.

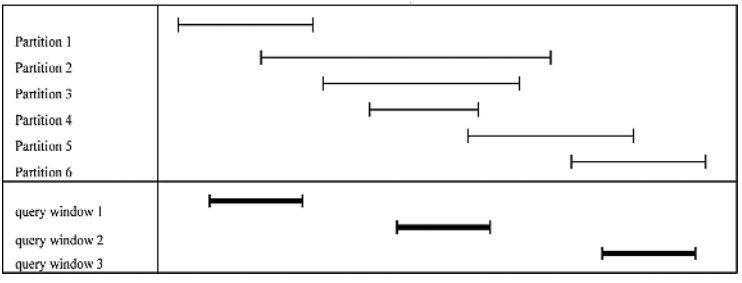

Fig. 3. Example of three queries with the same time window

To satisfy the final requirement, we construct a B+-tree by using the end time of a partition as a key so as to have fast accesses to partitions in POTSS. Figure 4 shows the time-based B+-tree structure. A record, Rec, of a leaf node in the time-based B-tree is <p_start_time, p_end_time, Pid, PLoc> where p_start_time and p_end_time mean the smallest start time and the largest end time of all the trajectories for a partition in POTSS, respectively. Here, 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], we first get a starting leaf node by searching the time-based B+-tree using t1, and then obtain records to satisfy the condition, p_end_time $\geq$ t1 AND p_start_time $\leq$ t2. The search space for processing the query with [t1, t2] ranges from Pa to Pb. Here Pa is the leaf node obtained by searching the B-tree with key = t1 and Pb is a leaf node containing the first record without holding the above condition by following leaf nodes in the sequence set from Pa. This allows for the minimum page I/O accesses required for answering the query.

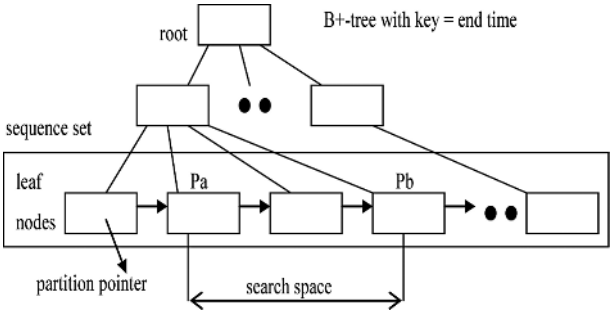

Fig. 4. Time-based B+-tree structure for partitions in POTSS

3.3 Insertion Algorithms for Trajectories of Moving Objects

The algorithms for inserting moving objects trajectories can be divided into an initial trajectory insertion algorithm and a segment insertion algorithm for its trajectory. For the initial trajectory insertion, we find the last partition in the partition table and obtain an available entry (NE) in the last partition. The initial trajectory insertion can be performed according to two cases; one with no expected future trajectories and the other with expected trajectories. First, for the insertion with no expected trajectories, we create a new expected future segment based on an edge where an object currently moves and store it into the NE entry of the trajectory information area in the last partition. Using the expected future segment created, we store start_time (StartT), current_time (CurrentT), and end_time (ExpectedET) into the NE entry of the location information area in the last partition. Here StartT and CurrentT are both assigned to the start time of the moving object and ExpectedET is assigned to NULL. Figure 5

```
Algorithm InsertFirst(MOid, TrajSegList)
/* TraSegList contains the information of a set of expected segments for
the trajectory of a moving object Moid */
1.  TrajSeg = the first segment of TrajSegList
2.  Generate a signature SigTS from TrajSeg
3.  StartT = CurrentT = ts of TrajSeg
4.    Obtain final_entry_no of the entry, in the partition table, for
      the last partition, LP
5.    NE = final_entry + 1 //NE=the next available entry in LP
6.    Obtain the location, Loc, of the entry NE in the trajectory info
      area for inserting object trajectory
7.      if(end field of TrajSeg=NULL){//no expected trajectory
8.     ExpectET = NULL
9.      Store <MOid,0,1,TrajSeg> into the entry NE, pointed by Loc, of
          the trajectory information area in LP}
10. else { // expected trajectory exists
11.    #fseg = 1
12.    while (the next segment Sn of TrajSegList ≠ NULL) {
13.        #fseg = #fseg + 1
14.        Generate a signature SSn from Sn and SigTS = SigTS | SSn }
15.       Store <MOid,0,#fseg,TrajSegList> into the entry NE, pointed by
          Loc, of the trajectory info area in LP
16.       Compute ExpectET by using ts, start, and v of the last segment
          of TrajSegList
17.   } // end of else
18.   Store SigTS into the entry NE of the signature info area in LP
19.   Store <MOid,Loc,StartT,CurrentT,ExpectET> into the entry NE of the
      location information area in the LP
20.   Store <StartT,CurrentT,ExpectET,NE> into the entry for LP in the
      partition table
End InsertFirst
```

Fig. 5. Initial trajectory insertion algorithm for moving objects

shows the initial trajectory insertion algorithm (i.e., InsertFirst). Secondly, for the initial trajectory insertion with expected ones, we insert a list of expected future segments (TrajSegList) into the NE entry of the trajectory information area in the last partition. In addition, we create a segment signature (SSn) from each of TrajSegList and generate a trajectory signature (SigTS) by using superimposing (Oring) all of the segment signatures. Using the TrajSegList, we store StartT, CurrentT, and ExpectedET into the NE entry of the location information area. ExpectedET is assigned to the expected end time of the last segment of the TrajSegList. Finally, we store the SigTS into the NE entry of the signature information area. We store <StartT, CurrentT, ExpectedET> into the last partition entry (LP) of the partition table.

For the segment insertion of a moving object trajectory, we find a partition storing its trajectory from the partition table by using the start time (ST) of the moving object.

```
Algorithm InsertSeg(MOid, TrajSeg, ST) /* TraSeg contains a segment for
the trajectory of a moving object Moid, to be stored with an object
trajectory's start time, ST*/
1.  Generate a signature SigTS from TrajSeg
2.    Locate a partition P covering ST in partition table
3.    Locate an entry E covering ST for the moving object with MOid in
      the location information area and get its location, Loc, in the
      trajectory information area
4.    Obtain #actual_seg, #future_seg, and #mismatch of the trajectory
      info entry E (i.e., TE) for the MOid in P
5.    if(#future_seg = 0) { // no expected trajectory
6.        Insert TrajSeg into(#actual_seg+1)-th trajectory segment of TE
7.        Store SigTS into the entry E of the signature info area in P}
8.    else { // expected trajectory exists
9.        seg_pos = find_seg(TrajSeg,Loc)
10.       #actual_seg++, #future_seg = #future_seg – seg_pos
11.       case(seg_pos = 0) { // find no segment
12.           Insert TrajSeg into segment of TE and relocate the future
              traj segments backward
13.           Store SigTS into entry E of the signature info area in P }
14.       case(seg_pos = 1) //find the first segment
15.           Insert TrajSeg into (#actual_seg)-th trajectory segment of
              TE for exchanging the old segment
16.       case(seg_pos > 1) {//find the (seg_pos)-th segment
17.           #mismatch = #mismatch + seg_pos – 1
18.        Insert TrajSeg into (#actual_seg)-th segment of TE and relo-
          cate the future traj segments forward
19.           if(#mismatch/(#future_seg+#actual_seg) >  )
              regenerate_sig(Loc,SigTS,E,P)}// end of case
20. } // end of else
21. Update #actual_seg, #future_seg, and #mismatch of TE
22. CurrentT = te of TrajSeg
23.   Store CurrentT into the current_time of the entry E of the loca-
      tion information area in the partition P and store CurrentT into
      the p_current_time of the partition P entry in the partition table
End InsertSeg
```

Fig. 6. Segment insertion algorithm for moving object trajectories

In addition, we obtain the entry storing the trajectory information in the partition. Figure 6 shows the segment insertion algorithm (i.e., InsertSeg) for moving object trajectories. Here NE is the entry in the partition covering the object identified by MOid and Loc is the location of the NE entry in the trajectory information area. The segment insertion can be performed in two cases. First, for a segment insertion for trajectories with no expected future ones, we just store a new segment (TrajSeg) into the NE entry of the trajectory information area, being addressed by Loc. In addition, we generate a trajectory signature (SigTS) from the TrajSeg and store the SigTS into the NE entry of the signature information area. Then, we store <MOid,Loc, StartT,CurrentT,ExpectET> into the NE entry of the location information area. Secondly, for a segment insertion for trajectories with expected future ones, we can store a new segment according to three types of the discrepancy between a new segment and the expected segment of a trajectory. To check if a new segment accords with an expected trajectory's segment, we call a find-seg() function to find a segment coinciding with TrajSeg from the expected trajectory of the NE entry. First, in case of no segment coinciding with TrajSeg (seg_pos = 0), we perform the same procedure as the segment insertion algorithm with no expected future segments. In addition, we move the trajectory's expected segments backward by one and store the TrajSeg into the (#_actual_seg)-th segment of the NE entry. Secondly, in case where the segment coinciding with TrajSeg is the first one (seg_pos = 1), we store only the TrajSeg into the (#_actual_seg)-th segment of the NE entry because the TrajSeg is the same as the first expected segment of the trajectory. Otherwise (seg_pos > 1), we delete the (seg_pos-1) number of segments from the expected segments of the NE entry, store the TrajSeg into the (#_actual_seg)-th segment, and move all the expected segments forward by seg_pos-2. If the ratio of mismatched segments (#_mismatch) over all the segments of the trajectory is less than a threshold (τ), we store the trajectory signature (SigTS) generated from the TrajSeg into the NE entry of the signature information area. Otherwise, we regenerate SigTS from the trajectory information by calling a signature regeneration function (regenerate_sig). Finally, we update the values of #_actual_seg, #_future_seg, and #_mismatch in the NE entry, and we update the CurrentT of the NE entry in the location information area and that of the partition P's entry in the partition table.

3.4 Retrieval Algorithm for Trajectories of Moving Objects

The retrieval algorithm for moving object trajectories finds a set of objects whose trajectories match the segments of a query trajectory. Figure 7 shows the retrieval algorithm (i.e., Retrieve) for moving object trajectories. To find a set of partitions satisfying the time interval (TimeRange) represented by <lower, upper> of a given query (Q), we call a find_partition function to generate a list of partitions (partList) by searching both the partition table of COTSS and the B+-Tree of POTSS. The search cases can be determined by comparing the TimeRange (T) with the p_end_time (PEtime) of the last partition in POTSS as well as with the p_start_time (CStime) of the first partition in COTSS as follows.

1. If T.lower > PEtime, both T.lower and T.upper are ranged in COTSS
2. If T.upper $\leq$ PEtime AND T.upper < CStime, both T.lower and T.upper are ranged in POTSS

3. If T.upper≤PEtime AND T.upper ≥ CStime, both T.lower and T.upper are ranged in POTSS and T.upper is at least within in COTSS simultaneously
4. If T.lower≤PEtime AND T.upper>PEtime, T.lower is within POTSS while T.upper is in COTSS

For the first case, we perform the sequential search of the partition table in COTSS and find a list of partitions (partList) to satisfy the condition that if end_time≠NULL, end_time ≥ T.lower AND start_time ≤ T.upper and otherwise, current_time ≥ T.lower AND start_time ≤ T.upper. Because the partition table of COTSS is resident in a main memory, the cost for searching partition table is low. For the second case, we get a starting leaf node by searching the B+-tree of POTSS with key = lower and obtain the partList to satisfy the above condition by searching the next leaf nodes from the starting leaf node in the sequence set. For the third case, we get two lists of partitions to satisfy the TimeRange in both COTSS and POTSS, respectively. We obtain the partList by merging the two lists of partitions acquired from both POTSS and COTSS. For the last case, we get a starting leaf node by searching the B+-tree of POTSS with key = lower and obtain a list of partitions to satisfy the TimeRange and obtain a list of partitions to satisfy a condition p_start_time ≤ T.upper by searching

```
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 = ∅
2.  t1 = TimeRange.lower, t2 = TimeRange.upper
3.  for each segment QSj of QsegList {
4.       Generate a signature QSSi from Qsj
5.       QSig = QSig | QSSj, #qseg = #qseg + 1 }
6.  find_partition(TimeRange, partList)
7.  for each partition Pn of partList {
8.       Obtain a set of candidate entries, CanList, examining the sig-
          natures of signature info area in Pn
9.       for each candidate entry Ek of CanList {
10.      Let s,e,c be start_time, end_time, current_time ofthe entry Ek
          of location information area
11.      if((s ≤ t2) AND (e ≥ t1 OR c ≥ t1)){
12.       #matches = 0
13.       Obtain the first segment ESi of the entry Ek of the trajectory
          info area, TEk and obtain the first segment QSj of QsegList
14.       while(ESi ≠ NULL and QSj ≠ NULL) {
15.       if(match(Esi, QSj)=FALSE)
               Obtain the next segment ESi of TEk
16.       else { #matches = #matches + 1
17.            Obtain the first segment ESi of Tek }
18.       if(#matches=#qseg)MOidList=MOidList ∪ {TEk's MOid}
19.       } } } //end of while //end of if //end of for- CanList
20. } // end of for - partList
End Retrieve
```

Fig. 7. Retrieval algorithm for moving object trajectories

the partition table of COTSS. We obtain the partList by merging the partitions acquired from POTSS and those from COTSS. Next, we generate a query signature (QSig) from a query trajectory's segments. For each partition of the partList, we search the signatures in the signature information area and acquire a list of candidates (CanList). For the entries corresponding to the candidates, we determine if their start_time, end_time, and current_time satisfy the condition. Finally, we determine if the query trajectory matches the object trajectories corresponding to the entries. If it matches object trajectories, we insert the object 's ID into a result list (MoidList).

4 Performance Analysis

We implement our trajectory indexing scheme under Pentium-IV 2.0GHz CPU with 1GB main memory, running Window 2003. For our experiment, we use a road network consisting of 170,000 nodes and 220,000 edges [WMA]. For simplicity, we consider bidirectional edges; however, this does not affect our performance results. We also generate 50,000 moving objects randomly on the road network by using Brinkhoff's algorithm [B2]. For performance analysis, we compare our indexing scheme with the TB-tree and the FNR tree in terms of insertion time and retrieval time for moving object trajectories. Table 1 shows the insertion performance to store one moving object trajectory. It is shown from the result that our indexing scheme preserves nearly the same insertion performance as TB-tree, but the FNR tree provides about two orders of magnitude worse insertion performance than TB-tree. This is because the FNR-tree constructs a tremendously great number of R-trees, i.e., each per a segment in the road network.

Table 1. Ttrajectory insertion performance

	TB-tree	FNR-tree	Our indexing scheme
Trajectory insertion time(sec)	1.232	401	1.606

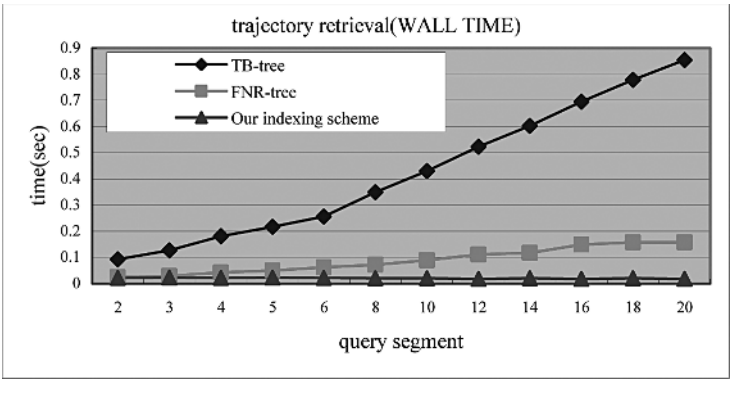

Fig. 8. Trajectory retrieval performance

We measure retrieval time for answering queries whose trajectory contains 2 to 20 segments. Figure 8 shows the trajectory retrieval performance. It is shown from the result that our indexing scheme requires about 20 ms while the FNR-tee and the TB-tree needs 25ms and 93ms, respectively, when the number of segments in a query is 2. It is shown that our indexing scheme outperforms the existing schemes when the number of segments in a query trajectory is small. On the contrary, the TB-tree achieves bad retrieval performance due to a large extent of overlap in its internal nodes even when the number of segments in a query trajectory is small. As the number of segments in queries increase, the retrieval time is increased in both the FNR-tree and the TB-tree; however, our indexing scheme requires constant retrieval time. 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 once 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 requires about 20 ms while the FNR-tree and the TB-tree needs 150ms and 850ms, respectively. Thus our indexing scheme achieves about one order of magnitude better retrieval performance than the existing schemes. This is because our indexing scheme constructs an efficient signature-based indexing structure by using a superimposed coding technique. On the contrary, the TB-tree builds a MBR for each segment in a query and performs a range search for each MBR. Because the number of range searches increases in proportion to the number of segments, the TB-tree dramatically degrades on trajectory retrieval performance when the number of segments is great. Similarly, the FNR-tree should search for an R-tree for each segment in a query. Because it gains accesses to as the large number of R-trees as the number of segments in the query, the FNR-tree degrades on trajectory retrieval performance as the number of segments is increased.

5 Conclusions

Even though moving objects usually moves on spatial networks, there has been little research on trajectory indexing schemes for spatial networks, like road networks. Therefore, we proposed an efficient indexing scheme for moving objects' trajectories on road networks. For this, we designed a signature-based indexing scheme for efficiently dealing with the current trajectories of moving objects as well as for maintaining their past trajectories. In addition, we provided both insertion and retrieval algorithms for their current and past trajectories. Finally, we show that our indexing scheme achieves, to a large extent, about one order of magnitude better retrieval performance than the existing schemes, such as the FNR-tree and TB-tree. As future work, it is needed to study on a parallel indexing scheme for moving objects' trajectories, due to the simple structure of signature files [ZMR98].

References

[B02] T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," GeoInformatica, Vol. 6, No. 2, pp 153-180, 2002.

[F03] R. Frentzos, "Indexing Moving Objects on Fixed Networks," Proc. of Int'l Conf on Spatial and Temporal Databases (SSTD), pp 289-305, 2003.

[FC84] C. Faloutsos and S. Christodoulakis, "Signature Files: An Access Method for Documents and Its Analytical performance Evaluation," ACM Tran. on Office Information Systems, Vol. 2, No. 4, pp 267-288, 1984.

[PJT00] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approach to the Indexing of Moving Object Trajectories," Proc. of VLDB, pp 395-406, 2000.

[PZM03] S. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," Proc. of VLDB, pp, 802-813, 2003.

[Se99] S. Shekhar et al., "Spatial Databases - Accomplishments and Research Needs," IEEE Tran. on Knowledge and Data Engineering, Vol. 11, No. 1, pp 45-55, 1999.

[SKS03] C. Shahabi, M.R. Kolahdouzan, M. Sharifzadeh, "A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases," GeoInformatica, Vol. 7, No. 3,, pp 255-273, 2003.

[SJK03] L. Speicys, C.S. Jensen, and A. Kligys, "Computational Data Modeling for Network-Constrained Moving Objects," Proc. of ACM GIS, pp 118-125, 2003.

[WMA] http://www.maproom.psu.edu/dcw/

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted Files Versus Signature Files for Text Indexing," ACM Tran. on Database Systems, Vol. 23, No. 4, pp 453-490, 1998.