Efficient Indexing in Trajectory Databases

Chang-Il Cha[1], Sang-Wook Kim[1], Jung-Im Won[1], Junghoon Lee[2], and Duck-Ho Bae[1]

[1] College of Information and Communications
Hanyang University
Seoul, Korea
[2] Department of Computer Science and Statistics
Jeju National University
Jeju, Korea

Abstract. This paper addresses an indexing scheme capable of efficiently processing range queries in a large-scale trajectory database. After discussing the drawbacks of previous indexing schemes, we propose a new scheme that divides the temporal dimension into multiple time intervals and then, by this interval, builds an index for the line segments. Additionally, a supplementary index is built for the line segments within each time interval. This scheme can make a dramatic improvement in the performance of insert and search operations using a main memory index, particularly for the time interval consisting of the segments taken by those objects which are currently moving or have just completed their movements, as contrast to the legacy schemes that store the index totally on the disk.

1 Introduction

By spatiotemporal data, we mean the data on the 3D space (x, y, t), which contains space coordinates, namely, (x, y), as well as time instance, (t). The path taken by an object is called a *trajectory*, and it can be expressed as the series of line segments on the 3D space[3][5].

Trajectory information has the following characteristics: (1) The cost of update operations is enormous in that an object changes its location constantly along the time axis. (2) It needs a lot of space to accumulate a large amount of changing data. (3) Search overhead inevitably gets larger due to a large quantity of stored data. Accordingly, it is necessary to develop a technique that can store, manage, and search the large-scale trajectory data.

As for an index structure for the future query, the VCI-tree[4], TPR-tree[5], and TPR*-tree[7] have been proposed. The TPR*-tree, the most outstanding scheme, stores MBR (Minimum Bound Rectangle) for the current location of an object as well as CBR (Conservative Bound Rectangle) for their moving speed[1]. Meanwhile, a lot of indexing schemes have been proposed for processing the historical query including the 3DR-tree[9], HR-tree[4], STR-tree[3], TB-tree[3], MV3R-tree[6], and SETI[2]. Among these, the 3DR-tree, HR-tree, and SETI are targeting at the range query while the others at the trajectory query. SETI has been known to be the best solution for the range query.

In this paper, we design a new indexing scheme capable of efficiently processing the range query in a large-scale trajectory database, based on the observation on the following two characteristics of a moving object database. First, each line segment in a trajectory is stored and indexed according to the time sequence, so the time field increases monotonically. Second, the range query is mainly issued for the latest time interval, not for the one long time ago.

The proposed scheme divides the time dimension onto multiple sub-intervals, which will be assigned line segments that are to be indexed. Within each time interval, another index will be used for the segments. The index that spans over the multiple recent time intervals is maintained within the main memory while other indexes are in disk. The architecture enhances the processing speed in segment insertion as the whole procedure is completed within the high-speed memory.

2 Related Work

2.1 SETI

The trajectory of a moving object, T_i, can be expressed as a tuple of (*moId*, *tId*, $\langle seg_1, seg_2, ..., seg_k\rangle$), where *moId* is the identifier of the object while *tId* is that of the trajectory. The $seg_j (1 \leq j \leq k)$ is the j^{th} line segment, belonging to a trajectory T_i, and is represented as (*sId*, (x_{start}, y_{start}, t_{start}), (x_{end}, y_{end}, t_{end})). This notation expresses that a segment named *sId*, has been moved from (x_{start}, y_{start}) to (x_{end}, y_{end}) during the time interval (t_{start}, t_{end}). This paper assumes that the line segment generated as time passes is continuously inserted to the above-mentioned index.

SETI[2] enables to efficiently retrieve moving objects within a query area using trajectory index structures. SETI is based on the critical differences in the characteristics of the temporal and spatial dimensions for storing 3D line segments using the R*-tree. That is, the values of spatial dimensions change rarely or slowly over the lifetime of the trajectory data while the values of temporal dimension are continuously increasing. Since the extent of the spatial dimensions does not change, SETI logically partitions the space extent into a number of non-overlapping spatial cells. Each cell contains only those line segments that are completely within cell. A sparse time index is built for each cell. In other words, line segments belonging to the same spatial cell are stored in a single data page, and the minimum and maximum time values that completely cover the time-spans of all the line segments in that page are indexed using a 1D R*-tree.

SETI exploits a hash table for fast insertion operations that add new line segments to the ends of existing trajectories. The hash table maintains the up-to-date locations for all moving objects. Figure 1 outlines the overall architecture of SETI.

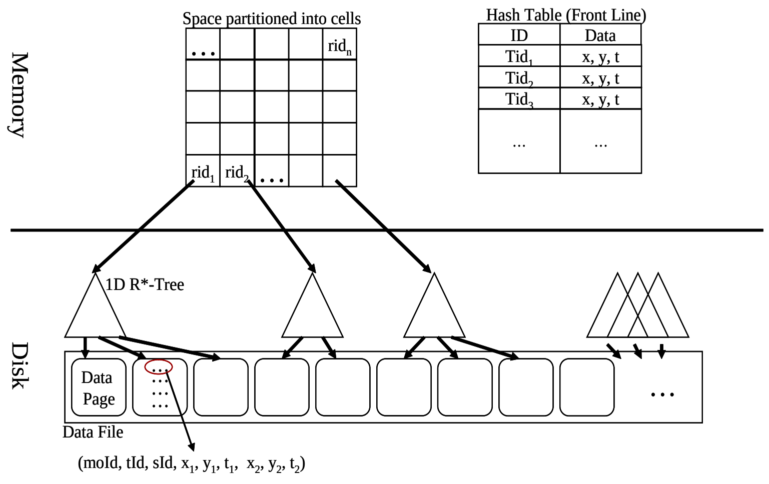

Fig. 1. Overall architecture of SETI.

2.2 Shortcomings of SETI

In this section, we point out the shortcomings of SETI as follows.

(1) **Limitation of a sparse time index**
With a sparse time index, we cannot determine whether the temporal condition of the given query is met solely by the line segments within the page pointed by the entry in the filtering step. Hence, this limitation inevitably imposes an overhead in checking whether the temporal condition is met for every candidate line segment in the data page during the refinement step.

(2) **Concentration of line segments**
The moving objects tend to concentrate at the specific small area, so many insert operations for line segments make the trajectory of objects also centered on the specific cell. Thus, the corresponding sparse time index for such a cell may be overloaded, resulting in the unpredictable performance of retrieve operations.

(3) **Missing of a spatial index**
As SETI does not exploit an additional spatial index within the partitioned cells, the number of candidate line segments gets larger. For an extreme example, when the size of the spatial condition area in a given query is much smaller than that of the cell area, a lot of line segments outside the query space are also included in the preliminary candidate set, possibly resulting in the severe waste of processing time and resources.

3 Proposed indexing scheme

3.1 Sketch of the proposed indexing method

To begin with, the proposed scheme stores the line segments constituting a trajectory of a moving object according to the time order in which they are collected. The query is more likely to be issued to the segments of moving objects which are currently moving or have just completed the movement other than those stopped or disappeared long time ago. Based on this observation, this paper proposes an indexing scheme capable of efficiently inserting and retrieving a line segment as time passes.

Contrary to SETI, the proposed scheme partitions a set of line segments into a number of non-overlapping time intervals for a temporal dimension, and in turn those line segments within each time interval are re-partitioned into spatial cells for the spatial dimension. Then, we build a spatial-temporal index for each cell. As such, if we first partition the line segments based on the temporal dimension, carried out the insert operation for the line segment of a moving object just on the last time interval. While most indexes are maintained in disk, the index of the last time interval is made to be resident on main memory. After all, our scheme can solve the problem of SETI that randomly accesses on the sparse time index in disk whenever the new line segment is inserted.

The size of an index for the insert operation is restricted to have at most k segments, considering available main memory space. That is, if the k^{th} line segment is inserted in the index, the index will be removed from main memory and moved to disk. Thereafter, only the retrieve operation can be carried out for this index in disk. This index is created by splitting the main memory index, which is built on the temporal dimension, into the sub-indexes on the spatial dimension. That is, we partition the extent of spatial dimensions of line segments within the same time interval into m non-overlapping spatial cells, and then build a sub-index to each of m cells. We call this index structure *multi-level multiple indexes.* The proposed scheme provides a non-uniform cell partition in which each cell includes $\lceil k/m \rceil$ line segments, while information on the partition of the overall spatial cell is maintained in a 2D-tree. By this, we can solve the problems of SETI, namely, the line segment concentration and missing spatial index.

3.2 Index architecture

As shown in Figure 2, the proposed scheme builds a time index table that manages information on the partition of a temporal dimension. The time index table is made up of $n+1$ entries, each of which stores the minimum and maximum time values that completely cover the time-spans of all the line segments within the time interval, and the physical location of the corresponding index to manage information on the partition of the spatial dimension. Namely, distinct $n+1$ indexes, I_0, I_1, ..., and I_n, are managed along with the index table. The size of this table is quite small enough to be loaded in main memory, while each index I_i can have at most k line segments.

Notice that the corresponding indexes, I_0, I_1, ..., and I_{n-2} for the time intervals, t_0, t_1, ..., and t_{n-2}, can be referred for the retrieve operation, as this index deals with just those trajectories whose movement has completed. Index $I_i (0 \leq i \leq n-2)$ is created by partitioning line segments within time interval t_i into m small cells based on the 2D space dimensions, so each of the partitioned

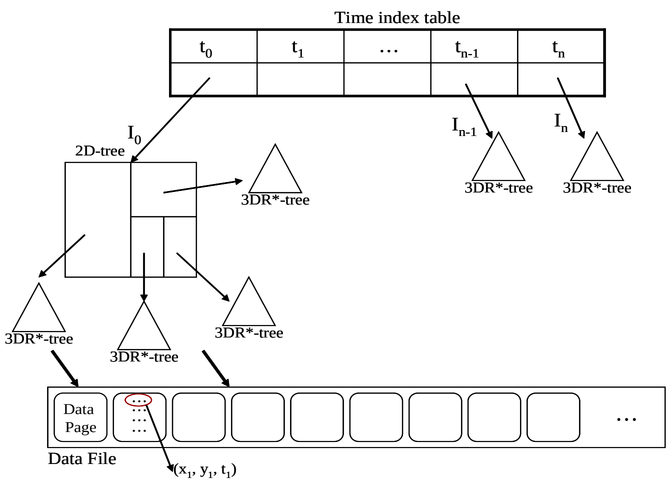

Fig. 2. Overall architecture of the proposed index.

cells has a variable size storage that can include up to $\lceil k/m \rceil$ line segments. Index I_i can be implemented using a hash table or a 2D-tree. Each of them stores the physical location of the corresponding 3D R*-tree index built for the line segments on 3D space (x, y, t) within the same space cell.

In the mean time, indexes I_{n-1} and I_n, mapped from t_{n-1} and t_n are responsible for the insert and retrieve operations for the latest segments of a currently moving object, being created with an R*-tree for the spatio-temporal data on the 3D (x, y, t). Our scheme makes indexes I_{n-1} and I_n contain at most predefined k line segments, and be loaded on the high-speed main memory for the sake of efficient insert and retrieve operations.

4 Performance evaluation

4.1 Experiment setup

Our experiment takes the user-defined parameters such as an object type and a distribution type in addition to 1,000 trajectory data uniformly distributed over [0, 1] interval created by GSTD (Generate SpatioTemporal Data) algorithm's scenario 4[8]. For the experiment according to the number of segments, five data sets are employed having 1,000,000 (1M), 2,000,000 (2M), 4,000,000 (4M), 8,000,000 (8M), and 16,000,000 (16M) segments which are chosen from the trajectory data, respectively. For each point randomly selected from the trajectory data, and also for each area tolerant of 0.01%, 0.1%, and 1%, 1,000 query areas are generated. Finally, for a pair of a each data set and a query area, 1,000 range queries are submitted and their average processing times are measured.

The performance of our scheme is compared with that of SETI. The number of cells for space dimensions is set to 400 ($= 20 \times 20$). In both schemes, a multidimension index is built by using the R*-tree provided by GIST[10] and the size of a data page is set to 2KB. Finally, experiments are performed on the platform employing Windows 2003 operating system upon the hardware stuffed with 3GB main memory, and Intel Xeon 3.0 no-coda 3.0 GHz CPU.

4.2 Results and analyses

Experiment 1: Comparison of index sizes

Experiment 1 compares the performance of two schemes according to the entire index size, which considers both the index file and the data page file. Figure 3 plots the index sizes for the proposed scheme and SETI according to the data size. This figure tells that the index size increases according to the increase of a data size. However, from the viewpoint of the absolute size, the proposed scheme can reduce the storage overhead by $3 \sim 4$ times compared with SETI. This is due to the fact that the proposed scheme stores (x, y, t) tuples and tId in the data page header just once, as the segments of a trajectory are already stored in the time order while SETI uses a record structure containing $(tId, sId, x_1, y_1, t_1, x_2, y_2, t_2)$.

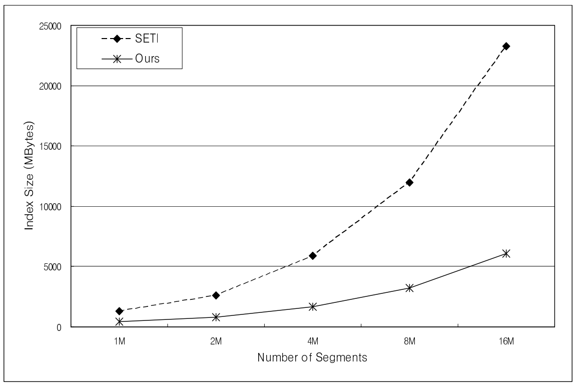

Fig. 3. Index sizes with various data sizes.

Experiment 2: Comparison of query processing times

Experiment 2 compares the performance of both schemes in terms of the query processing time, and Figure 4 exhibits the measured result according to the data size. The tolerance range of a query area is set to 0.1%. The result indicates that, for both schemes, the query processing time increases with the increase of the data size, but the proposed scheme outperforms SETI due to the reduced index search time caused by the smaller index. Figure 4 also shows that the performance of the proposed scheme can be improved by 30% $\sim$ 100%, compared with SETI.

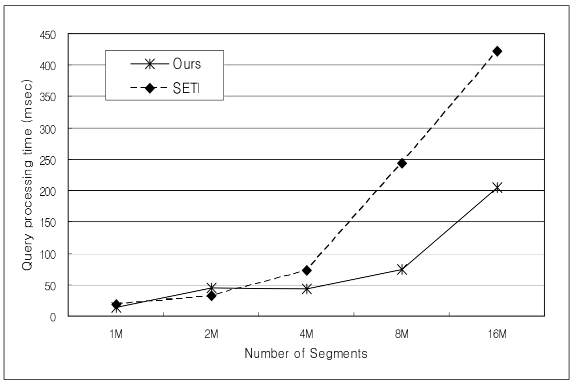

Fig. 4. Query processing times with various data sizes.

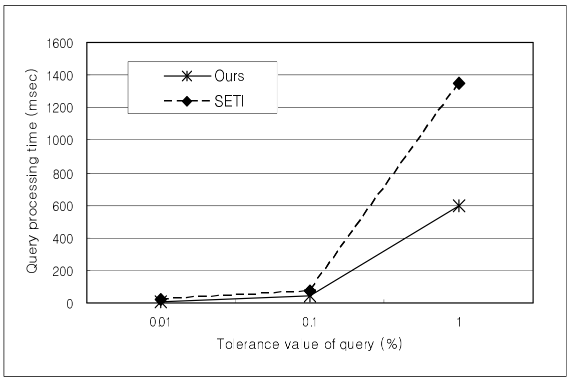

Fig. 5. Query processing times with various tolerances.

Figure 5 plots the query processing time according to the query area size, with the number of segments fixed to 4M. The increase of this range leads to the enlarged processing time as the number of indexes to be searched also increases. However, the proposed scheme shows much better performance because, in SETI, which uses a sparse time index, the expansion of a query area generally imposes a significant overhead in checking whether the temporal condition is satisfied for all the segments inside a data page. This overhead is quite high when the size of a query area is large. After all, the proposed scheme can improve the processing time by 50% $\sim$ 120%, compared with SETI.

5 Conclusions

This paper has proposed an efficient index structure for a large-scale trajectory database of moving objects, and also has devised a range query processing algorithm. The proposed scheme makes maximizing the efficiency of insert and retrieve operations, based on the observation that each line segment constituting a trajectory is inserted as time increases and queries are mainly issued to the newly inserted segment rather than the old ones. To this end, current and old segments are discriminated by classifying segments into multiple categories according to time intervals. The index for those segments frequently accessed is maintained in main memory while the indexes for others are maintained in disk. The experiment result reveals that the proposed indexing scheme enhances the speed of retrieve operations by up to 3 $\sim$ 10 times with a much smaller storage overhead.

Acknowledgments: This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2007-313-D00651 and KRF-2007-314-D00221), and was partly supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2008-C1090-0801-0040).

References

1. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger.: The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. Int'l. Conf. on Management of Data*, (1990) 322–331
2. V. P. Chakka, A. C. Everspaugh, and J. M. Patel.: Indexing large trajectory data sets with seti. In *Proc. Int'l. Conf. on Biennial Conference on Innovative Data Systems Research*, (2003)
3. D.Pfoser, C. S. Jensen, and Y. Theodoridis.: Novel approaches in query processing for moving object trajectories. In *Proc. Int'l. Conf. on Very Large Data Bases*, (2000) 395–406
4. M. A. Nascimento and J. R. O. Silva.: Towards historical r-trees. In *ACM Symp. on Applied Computing*, (1998) 235–240
5. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. Lopez.: Indexing the positions of continuously moving objects. In *Proc. Int'l. Conf. on Management of Data*, (2000) 331–342
6. Y. Tao and D. Papadias.: Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, (2001) 431–440
7. Y. Tao, D. Papadias, and J. Sun.: The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, (2003) 790–801
8. Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento.: On the generation of spatiotemporal datasets. In *Proc. Int'l. Symp. on Large Spatial Databases*, (1999) 147–164
9. Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis.: Spatio-temporal indexing for large multimedia applications. In *Proc. Int'l. Conf. on Multimedia Computing and Systems*, (1996) 441–448
10. B. University.: The gist indexing project. *http://gist.cs.berkely.edu*, (2007)