Indexing the Trajectories of Moving Objects

Hung-Yi Lin, *Member, IAENG*

***Abstract*—A novel point access method based on compressed B$^+$-trees is developed to support the efficient querying of the past, current, and anticipated future position of moving objects in this paper. The contributions of the proposed approach are twofold. First, the spatial and temporal information of mobile objects are handled by the compact tree hierarchies with high space utilization and least storage requirement. Second, the resulting indexing structures provide the simple searching methods and efficient query performance. Analytical studies show that the proposed indexing scheme is more efficient than TPR-tree.**

***Index Terms*—Spatiotemporal database, indexing structures, compressed B$^+$-tree, trajectories**

I. Introduction

Moving objects change their positions/geometries over time and thus their attribute values alter continuously. Moving points are objects whose spatial sizes are zero and their trajectories change over time. Many applications that create such mobile objects, including traffic surveillance data [5], intelligent vehicle systems [4], users of wireless devices [12], wildlife distribution management, disease surveillance data, and remote sensing application. In addition, moving objects with non-zero sizes may grow/shrink their extents or areas over time. As far as object's movements are concerned, some representative points inside non-zero size objects are taken for tracking their trajectories. Spatiotemporal database systems [1] are responsible to the management of continuous movements of mobile objects.

In this paper, we develop a new indexing scheme for handling spatiotemporal data, which is designed to support applications involving continuous movement that includes the historical, the current, and the anticipated future positions of moving objects. The technique naturally extends the compressed B$^+$-tree [10]. Instead of appealing any geometric objects or data transformation to represent their continuous motions, the endpoints of objects' trajectories are handled directly in our method. Further, basic insertion and deletion operations are enough for maintaining the index. No other operation is required for data updates. Removal of the expiry movements which are valid at some past time intervals may cause partial restructuring in the index. However, no periodic index rebuilding is required in our design. The proposed indexing structure can not only utilize the economic storage requirement but also fast perform queries on a large scale of database. The issues addressed in this paper include the classification of spatial and temporal data, the insertion, deletion, and maintenance algorithms. In addition, the analytical study and query performance are also covered to have the competitive comparison between TPR-tree indexing scheme and our method.

Manuscript received December 24, 2008. This work was supported by the Nation Science Council of ROC under Grant No. 97-2221-E-025-014-.

Hung-Yi Lin is with the National Taichung Institute of Technology, 129, Sanmin Rd., Sec. 3, Taichung, TAIWAN, R.O.C (phone: 886-4-2219-6769; fax: 886-4-2219-6161; e-mail: linhy@ ntit.edu.tw).

II. Indexing Schemes

Consider the space-time plane as shown in Fig. 1, where the issue timestamps, valid time intervals, and finish timestamps of moving objects distribute on the time-axis. The start positions, end positions, and the related space involved by moving objects are lying on the value-axis. The trajectories of four moving objects O_1、O_2、O_3 and O_4 are generated in an one-dimensional space. The movement of object O_1 is composed of line segments C、E and H in the space-time plane. The object O_2 keeps still and then its trajectory (denoted by D) is the line segment parallel to the time-axis. The movements of object O_3 is composed of line segments F、J and K. The object O_4 generates the trajectories corresponding to the segments G and I.

For managing all movements in a spatiotemporal database (denoted by *STDB*), we preserve the temporal attributes of moving points in one CB^+-tree and preserve the spatial attributes relative the same moving points in another CB^+-tree(s). TCB^+-tree and SCB^+-tree are used to denote temporal and spatial CB^+-tree for short, respectively. For two-dimensional moving objects, one TCB^+-tree associated with two SCB^+-trees (horizontal & vertical directions) are built for a *STDB*. When a *n*-dimensional *STDB* is considered, one TCB^+-tree and *n* indices of SCB^+-trees are built.

In the following, only one-dimensional moving objects are considered. First of all, our method extracts the temporal and spatial coordinates of endpoints of all trajectory segments (*S*) into two datasets D^t and D^y, which contain linearly ordered indexing data. That is,

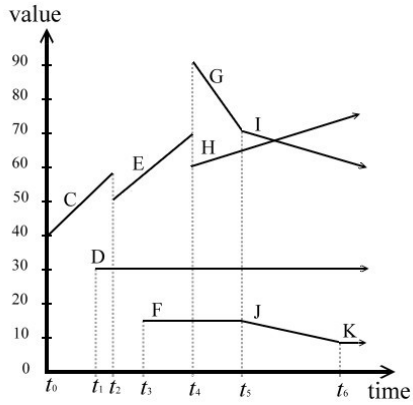

Fig. 1 Space-time representation for one-dimensional moving points.

$$D^t = \{t_k \mid t_k \text{ is the temporal - value of } S, \mid S^t \mid > 0, \forall S \in STDB\}. \quad (1)$$

$$D^y = \{y_k \mid y_k \text{ is the spatial - value of } S, \forall S \in STDB\}. \quad (2)$$

Nine trajectory segments share 11 endpoints in common in Fig. 1. The timestamp t_4 is shared by trajectory segments E, G, and H on the time-axis. Notably, a vertical segment is improper since it means that an object's location is indefinite at a fixed time. However, a horizontal segment means that an object stays still at a specific position during a time period. Our indexing scheme includes such stillness in the design. Every horizontal segment contributes one single record to D^y and two records to D^t.

A. Temporal CB^+-tree

The internal structure of a TCB^+-tree is a directory recording the critical timestamps when the objects issue, change, or terminate their movements. Object identifiers are classified by the critical timestamps and preserved at the bottom structure referred by the leaf entries of a TCB^+-tree. For a *STDB* as shown in Fig. 1, after collecting all critical timestamps to D^t, the data sequence $t_0 \to t_1 \to t_2 \to t_3 \to t_4 \to t_5 \to t_6$ is applied for the construction of TCB^+-tree. Let t_i denote an arbitrary timestamp on the time-axis, which may be an element in D^t or not. Terms t_i^- and t_i^+ are defined as the timestamps before and after t_i in D^t. Namely, we can not find t_m in D^t such that $t_i^- < t_m < t_i$ or $t_i < t_m < t_i^+$. The same argument is applied on the value-axis. For timestamp t_i, the bucket $\Gamma(t_i)$ includes the object identifiers that their movements keep moving in $[t_i, t_i^+]$. Namely, $\Gamma(t_i) = \{S \mid [t_i, t_i^+] \cap S^t \neq 0 \text{ and } \mid S^t \mid > 0, \text{where } \forall S \in STDB\}$. A modification improves the bottom structure of the resulting TCB^+-tree by eliminating the same identifier information between two consecutive buckets. The buckets referred by every leading key in leaves list the detailed information of related object identifiers. However, the minus (or plus) symbol in the consequent buckets excludes (or includes) the object from the former bucket. Fig. 2 illustrates a complete TCB^+-tree.

B. Spatial CB^+-tree

The construction of SCB^+-tree is based on organizing the critical locations of movements on the value-axis. The critical locations include the departure and arrival positions, the positions on which moving objects change their speeds and directions. The inertial and linear movement between two critical locations are not recorded but they are implied by the preservation at the bottom structure of a SCB^+-tree.

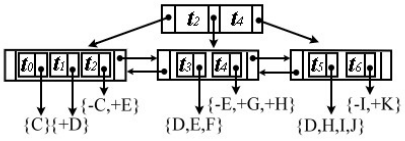

Fig. 2 TCB^+-tree.

Dissimilar to the bucket contents of a TCB^+-tree, one SCB^+-tree uses S, S', and $\overline{S}$ to distinguish the onward, backward, and still motions for a trajectory S with departure and arrival positions S_1^x and S_2^x. Following three criteria classify the object identifiers into the correct buckets at the bottom structure of a SCB^+-tree.

$$\Gamma(x_i) = \{S \mid [x_i, x_i^+] \cap S^x \neq 0 \text{ and } \mid S^t \mid > 0 \text{ and } S_1^x < S_2^x, \text{where } \forall S \in STDB\} \quad (3)$$

$$\Gamma(x_i^-) = \{S' \mid [x_i^-, x_i] \cap S^x \neq 0 \text{ and } \mid S^t \mid > 0 \text{ and } S_1^x > S_2^x, \text{where } \forall S \in STDB\} \quad (4)$$

$$\Gamma(x_i) = \{\overline{S} \mid [x_i, x_i^+] \cap S^x \neq 0 \text{ and } \mid S^t \mid > 0 \text{ and } \mid S^x \mid = 0, \text{where } \forall S \in STDB\} \quad (5)$$

At time t_0 in Fig. 1, segment C departs at position 40 and moves onward. This movement makes 40 inserted to the SCB^+-tree and creates bucket {C} referred by 40. At time t_1, segment D appears but stays still. This makes 30 inserted to the index and creates the bucket $\{\overline{\text{D}}\}$ referred by 30. Positions 50 and 60 are inserted at time t_2 and position 15 is inserted at time t_3. Particularly, at time t_4, a backward movement is issued at position 90. Position 90 becomes a critical location and then it is inserted to the index. However, as suggested by the second criterion(5), because $[90^-, 90]$ covers G's backward movement on the value-axis, the bucket referred by 90^- (i.e. 70) will include G′ in its content so that $\Gamma(70) = \{\text{G'}\}$ and $\Gamma(90) = \phi$. Further, applying the tuning mechanism similar to the TCB^+-tree, the complete SCB^+-tree is shown in Fig. 3.

C. Querying

The queries supported by the index retrieve all trajectory segments with overlaps with the specified regions. We distinguish five types of query based on the region they specify. Let t, t_a<t_b be three time values that are not less than the initial time t_0 and I, I_1, and I_2 are three spatial intervals for querying the value-axis.

Type 1: Point query: Q=(t, y) specifies a specific position y at the timestamp t.

Type 2: Interval query (slice query): Q=(t, I) and Q=(t_a, t_b, y) issue a time-slice and a location-slice query, respectively. Q=(t, I) specifies a spatial interval I at the timestamp t. Q=(t_a, t_b, y) specifies a query located at position y during the time interval between t_a and t_b. For example, in Fig. 4, Q_0 and Q_1 are the time-slice and location-slice query, respectively.

Type 3: Window query: Q=(t_a, t_b, I) specifies a spatial interval I that covers the time interval $[t_a, t_b]$. Q_2 in Fig. 4 is a window query.

Type 4: Predictive query: Q=(*now*, I) specifies a spatial interval I at the current time. Namely, this query is designed to retrieve all trajectory segments which have inertia movements passing through I from the current time to some limited future (i.e. now^+). Q_4 in Fig. 4 is a predictive query about the spatial

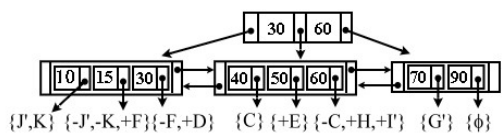

Fig. 3 SCB^+-tree.

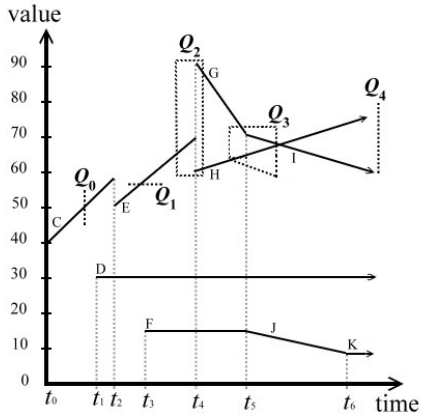

Fig. 4 Query examples for one-dimensional data.

interval [60, 80]. Further, window query Q=(t, *now*, I) retrieves all mobile objects which keep moving and pass through I during the time interval [t, *now*$^+$].

Type 5: Moving query: Q=(t_a, t_b, I_1, I_2) specifies the trapezoid obtained by connecting I_1 at time t_a to I_2 at time t_b. Q_3 in Fig. 4 illustrates this type.

The type 3 can generalize type 1, type 2, and type 4. The type 3 itself is a special case of type 5. To illustrate the data retrieving process for various query types, Q_0 to Q_4 are assigned as follows for demonstration.

Q_0=($t \in (t_0,t_1)$, [45,55])

Q_1=($t_a \in (t_2,t_3), t_b \in (t_3,t_4)$, 58)

Q_2=($t_a \in (t_3,t_4), t_b \in (t_4,t_5)$, [59,92])

Q_3=($t_a \in (t_4,t_5), t_b \in (t_5,t_6)$, [63,73], [58,73])

Q_4=(*now*, [60,80])

The implementation of query processes is threefold: 1) searching on the TCB^+-tree for the temporal output, 2) searching on the SCB^+-tree for the spatial output, 3) retrieving the common part of the temporal and spatial outputs. Since TCB^+-tree and SCB^+-tree are so compressed that they can resident at the main memory. And, the spatial queries on SCB^+-tree and temporal queries on the TCB^+-tree can execute concurrently. For query Q_0, $t \in (t_0,t_1)$ is applied on searching TCB^+-tree and the bucket {C} referred by a leading entry is retrieved. On the other hand, [45,55] is applied on searching SCB^+-tree and bucket {+E} is obtained. Bucket {+E} is referred by a non-leading entry of the leaf in the SCB^+-tree. The complete content of this bucket is reasoned as {C, E}. Segment C is the common identifier of the two retrieved buckets and then {C} is the final output for Q_0. The related outputs of Q_1 to Q_4 are listed in Table 1.

Notably, two trajectories with onward movements (segment E and H) and one trajectory with backward movement are retrieved by Q_2. For the process of moving query, two sub-window queries Q_{31} and Q_{32} as shown in Fig. 5

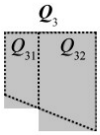

Fig. 5 The partition of a moving query.

are used to cover the trapezoid derived from Q_3. The partitioning positions for sub-windows are set at which the elements in D^t joint with Q^t. In this example, $D^t \cap Q_3^t = \{t_5\}$. Therefore, Q_3 is replaced by Q_{31}=([t_a, t_5], I') and Q_{32}=([t_5, t_b], I''), where $I' = [63 + (t_5 - t_a)/(t_b - t_a) \cdot (58 - 63), 73]$ and I''=[58,73]. Q_{31} and Q_{32} collaborate to retrieve two onward segments G and H plus the backward segment I. For query Q_4, the result predicts that the onward H and backward I are going to pass through the spatial interval [60,80] in the near future.

Without loss the generality, the spatial information of a *STDB* can also used to capture information of our surroundings by aspatial data for other applications. For instance, some environment probing need databases to handle the information about temperatures, pressures, or humidity versus temporal data. So, database users may issue the similar query Q_0 in Fig. 4 and interest in the objects whose temperatures are between 45 to 55 unit degrees at time t. In addition, other users may desire to retrieve the dynamic things related to a time interval [t_a, t_b] and a specific temperature of 58 unit degrees. Q_2 and Q_3 can meet the application of ocean exploring for searching some objects bearing some specific pressure values during a specific time interval. What is more interesting, meteorologists may expect to predict that some areas would have a specific humility value (for example 60~80 as Q_4) in the limited future.

III. INDEX MAINTENANCE

In this section, we explore the issues and complexity involved in maintaining the TCB^+-tree and SCB^+-tree. Index maintenance covers insertion and deletion of data records and/or tree nodes. Since the leaf nodes of one CB^+-tree always keep at least 75% full, the impact of an insertion (or a deletion) against the CB^+-tree is mostly local and limited. Generally, only three aspects are involved. The first is the object identifies in the retrieved buckets at the bottom structure. The second is the entry contents in the target and/or the sibling node. The third is the modification of the related key in the non-leaf node. In order to verify that less data updating overhead is required to our indexing scheme, we sketch the procedures in progressing batches of insertions and deletions. Besides data updates in the content of nodes and buckets, a batch work always involves index restructuring. The general steps for a batch work are as follows:

1) find the target leaf node for data insertions/deletions,

Table 1 The query outputs for Q_1 to Q_4.

Query	Temporal outputs	Spatial data outputs	Final outputs
Q_1	{D,E,F}	{C, E }	{E}
Q_2	{D,E,F,G,H}	{C,E,H,I′,G′}	{E,G′,H}
Q_3	{D,F,G,H}∪{D,H,I,J}	{E,H,I′,G′}∪{C,E, H,I′,G′}	{G, H}∪{H,I}={G,H,I′}
Q_4	{D,H,I,K}	{E,H,I′,G′}	{H, I′}

2) insert/delete records to/from the target node, and
3) shift/redistribute the records in the target and sibling node, and
4) insert/delete leaf and non-leaf nodes to/from the index tree, and
5) update the data pointers.

Fig. 6 shows an example in illustrating the complexity of indexing restructuring for a batch of insertions. The initial TCB^+-tree is given at step (a) in Fig. 6. The generation of new temporal data is totally ordered. Namely, the value of later temporal data is always larger than the last element in D^t. Therefore, insertions to a TCB^+-tree will append new entries to its rightmost leaf. For example, a batch of data with order $t_7 \rightarrow t_8 \rightarrow t_9 \rightarrow t_{10} \rightarrow t_{11} \rightarrow t_{12}$ is applied, t_7 and t_8 make the last leaf node overflowed. Datum t_5 is shifted to the left sibling so that the rightmost leaf can accommodate t_7 and t_8. After that, the upper entry in the parent (root) is modified (see step (b) in Fig. 6). Then t_9 makes the last leaf node overflowed again, a split of the overflowed leaf node is then carried out. There occurs a new entry to the parent node (see step (c) in Fig. 6). When t_{10} and t_{11} are inserted to the TCB^+-tree, the overloading makes t_8 being removed to the left sibling and this redistribution permits the accommodation of t_{10} and t_{11} without appealing to node splitting (see step (d) in Fig. 6). Finally, a split of the last leaf node and subsequently a further split of the non-leaf parent node are carried out by the insertion of t_{12}. Since the parent node is also the root for the initial tree, the root splitting creates the new root and grows the tree with one more level as shown at step (e) in Fig. 6. The handle that a full target appeals to its underflow sibling is the important characteristic of CB^+-tree.

Next we present the complexity of index restructuring for a batch of deletions. We also present another example as shown in Fig. 7. The initial tree is given at step (a) in Fig. 7. The expiry data are always those temporal data generated earliest. The deletions of expiry data must happen to the leftmost side of the leftmost leaf node in the TCB^+-tree. Suppose the temporal data less and equal t_5 are going to become overdue in this illustration. We expect to recovery the similar data arrangement and the same tree hierarchy as shown previously before data t_7 to t_{12} are inserted. At first, data (t_0, t_1, t_2) in the first leaf are removed. As shown at step (b) in Fig. 7, the first leaf node is returned to the system and the pointer corresponding this leaf node becomes void. Note that the entry at the parent is replaced by the maximum of all entries in the next leaf node. Due to the underflow, a merging with sibling nodes is carried out at the second level. Because the non-leaf node (t_5) has the same entry with its parent node (root node (t_5)), these two non-leaf nodes are collapsed together. Consequently, a new non-leaf node (t_5, t_8, t_{10}) is created (see step (c) in Fig. 7). Finally, data $t_3 \sim t_5$ also become overdue and must be removed from the index. Again, the leftmost leaf node is removed from the index and this operation causes the upper related entry and the corresponding pointer are removed from the parent node. The resulting index returns to the same manner as the initial tree in Fig. 6. As regards the SCB^+-tree, insertion and deletion may happen to anywhere at leaf level. The processes of data shifting and data merging can follow either right pointer or left pointer to find the suitable sibling for index restructuring. We omit to illustrate the process of data inserting and deleting in a SCB^+-tree.

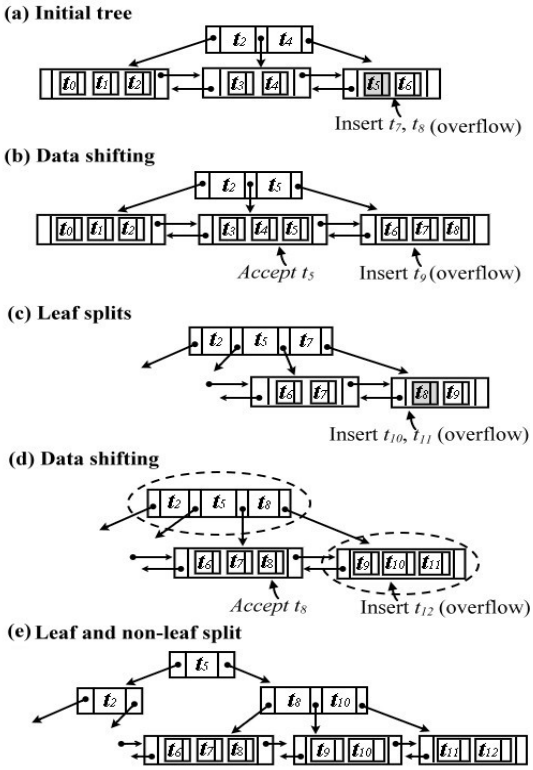

Fig. 6 The index maintenance for insertion operations.

IV. Analytical Evaluation

In this section, the spatiotemporal data moving on a two-dimensional space is utilized for analytical studies. The node capacity (order), bucket size, data size, and index depth are first investigated for facilitating the analysis of internal and external structures of the index. The storage requirement

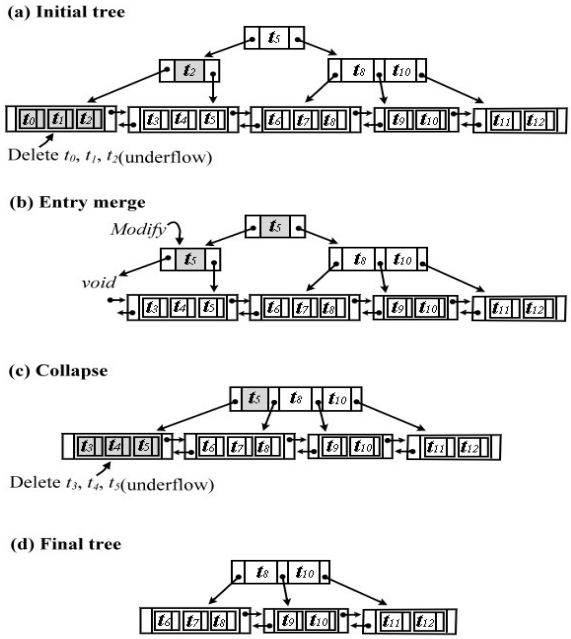

Fig. 7 The index maintenance for deletion operations.

of an index is analyzed via collecting the number of non-leaf and leaf nodes generated in the index. The query performance of an index is evaluated via counting the number of involved nodes and the number of triggered data comparisons in the searching process. The following system parameters are used in our analysis.

- B: Memory block size;
- t: Integer size;
- p: Pointer size;
- N: The data amount of moving objects in a database;
- M_α, M_β: The orders of CB^+-tree and *TPR*-tree;

For handling the movements of 2D moving objects, every entry in a *TPR*-tree's leaf nodes contains a pair of the position parameter and a pointer to the moving point. There is also a pointer links to the next leaf. Similarly, every entry in a *TPR*-tree's non-leaf nodes is a pair of a pointer to a sub-tree and a rectangle that bounds the positions of all moving points or other bounding rectangles in that sub-tree. Then, the order M_β of *TPR*-tree follows the formula:

$$M_\beta(4t+p)+p=B \Rightarrow M_\beta=\lceil (B-p)/(4t+p)\rceil \quad (6)$$

On the other hand, although each leaf node of a CB^+-tree needs two links to its fore and next sibling, the order of a CB^+-tree is larger than that of a *TPR*-tree when the same spatiotemporal data are handled. This is because every data entry preserved in a CB^+-tree has merely one indexing value and one pointer. Then, together with two pointers link to the siblings, the order M_α of CB^+-tree is given as following:

$$M_\alpha(t+p)+2p=B \Rightarrow M_\alpha=\lceil (B-2p)/(t+p)\rceil \quad (7)$$

In the general implementation, 4 Kbytes is a reasonable size for a memory block. One integer number and one pointer employ 16 bytes and 32 bytes, respectively. As a result, M_α and M_β are approximated as 84 and 43. This means that each node of CB^+-tree and *TPR* -tree can respectively accommodate at most 84 and 43 entries. The data capacity of one node in a CB^+-tree is twice of that in a *TPR* -tree.

A. The evaluation of storage requirement

The average number of data preserved in a bucket is approximately determined as $2N/|D^t|$ and $2N/|D^y|$ for TCB^+-tree and SCB^+-tree, where N is the total amount of data handled by the index. The magnitude of buckets is bounded by the size of $|D^t|$ (or $|D^y|$) as given in [9]. For convenience, we assume that the arrival of mobile objects is a Poisson process. Hence, their inter-arrival time is exponentially distributed with a mean $1/\lambda$. Let X be a Poisson random variable which represents the number of arrivals in unit time. Then $|D^t|$ and $|D^y|$ can be approximated by

$$2N \times \text{Prob}(X>0)=2N[1-(1-e^{-\lambda})]=2Ne^{-\lambda} \quad (8)$$

The average number of data preserved in a bucket is thus given by $2N/2Ne^{-\lambda}=e^{\lambda}$.

Our CB^+-tree has the near-full accommodation at leaf level, then the number of leaf nodes can approximate as $\lfloor (2Ne^{-\lambda})/M_\alpha \rfloor$. Since CB^+-tree does not organize the compactness in its internal structure, the accommodation of non-leaf nodes in a CB^+-tree is supposed to possess the similar utilization as that of a traditional B^+-tree. That is $\ln 2 \times 100\% = 69.3\%$ full [15]. We account for the leaf and non-leaf nodes together; the total amount of nodes generated by a SCB^+-tree or a TCB^+-tree is approximated by

$$\begin{aligned} &\lceil (2Ne^{-\lambda})/M_\alpha \rceil \cdot [1+1/(M_\alpha \ln 2)+\cdots] \\ &\approx \lceil (2Ne^{-\lambda})/M_\alpha \rceil \cdot [(M_\alpha \ln 2)/(M_\alpha \ln 2-1)] \end{aligned} \quad (9)$$

On the other hand, the total amount of nodes generated by a *TPR*-tree is approximated by

$$\begin{aligned} &\lceil (2Ne^{-\lambda})/(M_\beta \ln 2) \rceil \cdot [1+1/(M_\beta \ln 2)+\cdots] \\ &\approx \lceil N/(M_\beta \ln 2) \rceil \cdot [(M_\beta \ln 2)/(M_\beta \ln 2-1)] \end{aligned} \quad (10)$$

The depth of a SCB^+-tree (or a TCB^+-tree) and a *TPR*-tree are approximated by $\lfloor \log_{\lceil M_\alpha \ln 2 \rceil}(2Ne^{-\lambda})/M_\alpha \rfloor + 1$ and $\lfloor \log_{\lceil M_\beta \ln 2 \rceil} N/(M_\beta \ln 2) \rfloor + 1$, respectively. The ratio between their major parts is

$$\begin{aligned} &(\lfloor \log_{\lceil M_\alpha \ln 2 \rceil}(2Ne^{-\lambda})/M_\alpha \rfloor)/(\lfloor \log_{\lceil M_\beta \ln 2 \rceil} N/(M_\beta \ln 2) \rfloor) \\ &\approx [\log(M_\beta \ln 2)]/[\log(M_\alpha \ln 2)] \\ &\quad \times [\log(2Ne^{-\lambda})/M_\alpha]/[\log N/(M_\beta \ln 2)] \\ &< [\log(2Ne^{-\lambda})/M_\alpha]/[\log N/(M_\beta \ln 2)] \\ &\approx (\log N - \log e^{\lambda} - \log M_\alpha/2)/(\log N - \log \ln 2 - \log M_\beta) \end{aligned} \quad (11)$$

Since $e^{\lambda} > \ln 2$ and $M_\alpha/2 \approx M_\beta$, the ratio value is less than 1. It reveals that the depth of a SCB^+-tree (or TCB^+-tree) is always shorter than that of a *TPR*-tree.

For the completeness of analysis, we apply the data amount $N=10^5$ and $\lambda=1$ to the simulation and the related results are listed in Table 2. Notably, the average number of entries accommodated in a *TPR*-tree's leaf is about $43 \times 69.3\% \approx 30$. Two SCB^+-trees and one TCB^+-tree cooperate to complete the indexing work. The ratio of total nodes generated between our method and *TPR*-tree is about $(3 \times 892)/3473 \approx 0.77$. This reveals that the more economic storage requirement is entailed on our method.

B. The evaluation of query performance

Since the compactness is ensured by the proposed indexing techniques we suppose the whole index trees can resident in the memory. The time complexity of data comparison executed between two values is assumed to be $O(1)$ and the time complexity of data search starting from the roots of SCB^+-trees and *TPR*-tree is investigated in this subsection. Since one round of point query in a CB^+-tree only activates one single search path. It involves $H=\lfloor \log_{\lceil M_\alpha \ln 2 \rceil}(2Ne^{-\lambda})/M_\alpha \rfloor + 1$ nodes in the search path. In average, every non-leaf node is $\ln 2 \times 100\%$ full and every leaf

Table 2. The simulated results of a SCB^+-tree (or TCB^+-tree) and a *TPR*-tree.

Indexing scheme	Order	# leaves	# nodes	Depth
CB^+-tree	84	876	892	3
TPR-tree	43	3356	3473	4

node is near 100% full, where we suppose half of the entries in the involved non-leaf and leaf nodes are taken for data comparison. The average time complexity for completing one round of point query on the SCB^+-trees (or *TPR*-tree) is approximately estimated as

$$O(\text{Type 1}) = (\left| \log_{\lceil M_\alpha \ln 2 \rceil}(2Ne^{-\lambda}) / M_\alpha \right| \cdot (M_\alpha \ln 2)/2 + M_\alpha / 2) \cdot O(1) \quad (12)$$

One round of interval query in a CB^+-tree launches two single search paths. These two paths traverse the index tree and arrive at leaf level for retrieving two target data covering the query interval. Hence, (2*H*-2) non-leaf nodes and two target leaf nodes on the paths are involved. The time complexity of an interval query on the SCB^+-trees (or *TPR*-tree) is approximately estimated as

$$\begin{aligned} O(\text{Type 2}) &= [(2H-2)\cdot(M_\alpha \ln 2)/2 + 2 \cdot M_\alpha / 2] \cdot O(1) \\ &= (\left\lceil \log_{\lceil M_\alpha \ln 2 \rceil}(2Ne^{-\lambda}) / M_\alpha \right\rceil \cdot \ln 2 + 1) \cdot M_\alpha \cdot O(1) \end{aligned} \quad (13)$$

As regards window query, the total time complexity is approximated as

$$\begin{aligned} O(\text{Type 3}) = [&O(\text{Type 2 query on the } x\text{-axis}) \\ &+ O(\text{Type 2 query on the } y\text{-axis}) \\ &+ O(\text{Type 1 or Type2 query on the time-axis})] \cdot O(1) \end{aligned} \quad (14)$$

As far as *TPR*-tree is concerned, one round of rectangle comparison needs eight times of data comparison. This is because, for the *x*-value, the lower and upper *x*-coordinates of the issued query MBR execute data comparisons with the lower and upper *x*-coordinates of the candidate MBR. It consumes the time complexity of $4 \cdot O(1)$. The same augments are applied on the *y*-value. So, one round of rectangle comparison spends the higher overhead that is eight times of one round of data comparison. Namely, in a *TPR*-tree's non-leaf or leaf node, the average computing time of $8 \cdot O(1) \cdot (M_\beta \ln 2)/2$ is required to decide the next appropriate node for further searching. Unfortunately, the worst matter is the multiple paths. The average time complexity is approximated as $cH \cdot 8O(1) \cdot (M_\beta \ln 2)/2$, where *c* is the multiple factor satisfying $1 \le c \le M_\beta \ln 2$ and *H* is the depth of *TPR*-tree. Index depth, data comparison complexity, and search path are three major causes that make all types of query on a CB^+-tree more efficient than on a *TPR*-tree.

V. Conclusion

The compactness and stable query performance of indexing structures are two major contributions in this study. The using of compressed indexing structures makes data queries processed in a condensed memory bound so that the retrieval costs are limited within a very satisfactory range. Space and time efficiencies are simultaneously achieved by our indexing scheme.

The future direction of research is to preserve the spatial and temporal relations among mobile objects. Techniques for answering complex queries, such as reporting the areas with high density of mobile objects, predicting the time moment when a specific amount of mobile object will emerge or vanish, are of high practical interest. Furthermore, it would be also worth considering the problem in reporting the average velocity of a stream of mobile objects with similar movements.

References

[1] T. Abraham and J.F. Roddick, "Survey of Spatio-Temporal Databases," *GeoInformatica*, 1999, 3(1), pp. 61-99.

[2] N. Beckmann, H. Kriegel , R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *Proceeding of ACM SIGMOD*, 1990.

[3] H. Berliner, "The B*-tree search algorithm: a best-first proof procedure," *Tech. Rep. CUM-CA-78-112*, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, 1978.

[4] R. Bishop, "Intelligent Vehicle Technology and Trends," *Baker & Tayl*, 2005.

[5] M. Bramberger, J. Brunner, B. Rinner and H. Schwabach, "Real-time video analysis on an embedded smart camera for traffic surveillance," *10th IEEE Real-Time and Embedded Technology and Applications Symposium* (RTAS'04), 2004.

[6] C.S. Jensen, D. Lin and B.C. Ooi, "Query and Update Efficient B^+-tree Based Indexing of Moving Objects," *Proceedings of the 30th VLDB Conference 2004*, Toronto, Canada.

[7] G. Kollios, D. Papadopoulos, D. Gunopulos and V.J. Tsotras, "Indexing mobile objects using dual transformations," *The VLDB Journal*, 2005, 14(2), pp. 238-256.

[8] Y. Kuroda and Y. Amitani, "TRITON: New ocean and atmosphere observing buoy network for monitoring ENSO," *Umi no Kenkyu*, 2001, pp. 157-172.

[9] H.Y. Lin, "Efficient and Compact Indexing Structure for Processing of Spatial Queries in Line-based Databases," *Data & Knowledge Engineering*, 2008, 64(1), pp. 365-380.

[10] H.Y. Lin, R.C. Chen and S.Y. Chen, "Enhancement of Data Aggregation Using A Novel Point Access Method," *WSEAS Transactions on Computers*, issue 12, vol. 7, December 2008, pp 2001-2010.

[11] D. Papadopoulos, G. Kollios, D. Gunopulos and V. J. Tsotras, "Indexing mobile objects on the plane," *Proceedings of 13th International Workshop on Database and Expert Systems Applications*, 2002.

[12] T. Pering, V. Raghunathan and R. Want, "Exploiting radio hierarchies for power-efficient wireless device discovery and connection setup," *18th International Conference on VLSI Design*, 2005.

[13] S. Saltenis, C. S. Jensen, S. T. Leutenegger and M. A. Lopez, "Indexing the positions of continuously moving objects," *Proceedings of the ACM International Conference on Management of Data*, 2000, Dallas, USA: ACM Press.

[14] Y. Tao, D. Papadias and J. Sun, "The TPR*-tree:an optimized spatio-temporal access method for predictive queries," *Proceedings of the 29th International Conference on Very Large Data Bases 2003*: Berlin, Germany, pp. 790-801.

[15] A. Yao, "Random 2-3 trees," *Acta Informatica*, 1978, 2(9), pp. 159-170.