Indexing the Fully Evolvement of Spatiotemporal Objects

HUNG-YI LIN
Department of Distribution Management
National Taichung Institute of Technology
129, Sanmin Rd., Sec. 3, Taichung
TAIWAN, R.O.C.
linhy@ntit.edu.tw

Abstract: -This paper proposes a novel economic and efficient framework of indexing spatiotemporal objects based on compressed B^+-trees. Our new index technique can compress data records in leaves and in turn reduce index size to improve update and query efficiencies. The contribution of our framework is threefold. First, the proposed index structures are more succinct to resident in the main memory. Second, sufficient data with high similarity inside all indexing pages for the high retrieval quality of cache fetches are attainable. Third, in addition to process multi-dimensional spatial information, our framework is typically appropriate to model the valid- and transaction-time of a fact. We present the index efficiency analyses and experimental results which show that our technique outperforms others.

Key-Words: - Spatio-temporal databases, Compressed B^+-trees, Trajectories, Data aggregation, Storage utilization.

1 Introduction

Data management applications are rapidly developing that enables tracking of the locations of moving objects. Many applications create such mobile data, including traffic surveillance or real-time navigation systems, the positioning of users with wireless devices, and multimedia application (animated objects in movies). A wide range of sensing applications relies on the multidimensional observing variables which have ever changing over time. For example, an atmospheric or climatic observing buoys collect the changes of temperature, humidity, or pressure at various altitudes at different moments. The geographic distributions of wildlife are precisely traced for natural preservation applications. Land information system (LIS) and cadastral system are developed to trace the position and extent variations of land objects. The spatial attributes about point objects can be the information of location, position, or status of objects. The spatial attributes about region objects generally include the information of extent (i.e. the areas or volumes the objects occupy). Spatial information varies from time to time. Both point and region objects are handled in this paper.

Spatio-temporal databases [19] attempt to achieve an appropriate kind of interaction between spatial and temporal data and support an efficient and feasible solution to users' queries. Trying the integration of space and time is dealing with multi-dimensional geometries changing over time. Generally, these changes are contiguous so that the continuous movements of objects are the basic entities processed by spatio-temporal databases.

The performance of indexing structure is largely related to the techniques and strategies adopted in the indexing scheme. Indexing techniques include data model, data representation, data discrimination, and even transformation designs and methods. The requirements for these designs and methods are to precisely extract and preserve the essential information of movements. On the other hand, indexing strategies include similarity evaluation, classification, and node splitting methods which are expected to facilitate index building with ease and maintain indexing structure with efficiency. Notably, the merits of index strategies significantly affect the complexity of index hierarchy. The collaboration of functional indexing techniques with adequate

This research was supported by Nation Science Council of ROC under Grant No. 98-2221-E-025-008-.

indexing strategies is influential to the performance of an indexing structure.

Indexing of multi-dimensional data tends to be particularly troublesome since complicated spatial and temporal interaction always occur. Using dynamic data model or non-stationary function parameters in representing the movement of objects is one of the causes of frequent updates. Frequent updates of data or parameters need to traverse indexing structures so that they are costly. On the other hand, unnecessary node splits incur many costly operations. Hence, indexing techniques and indexing strategies are extraordinarily influential to the performance of multi-dimensional indexing structures.

Query performance depends on the complexities of resulted index hierarchy and triggered search paths. Compact indexing structures with short depths are expected since they bring the short search paths. Furthermore, data search may come in either single or multiple paths. Single search paths traversed in the compact index are definitely efficient and economic. The index structures proposed in this paper support data search with single and short search paths.

As the price of memory continue to become cheaper and cheaper, it is now feasible to place many of database tables and indexes in main memory. One key challenge derives from the need to accommodate the entire database index into main memory while simultaneously allowing for efficient update and query processing. The aggregative data arrangement inside the index pages and the compactness of the index hierarchy are crucial to conquer this challenge.

2 Related work

Many different policies related to data classification are developed in traditional indexing schemes. One of the common policies is to adopt pre-partitions of data space [5, 9, 16] before processing indexing scheme. The generation of data is usually one-by-one, but not of batch, so that data space pre-partitioned according to a specific design in advance of the collection of all data is difficult to conform to the practical distribution of data. Consequently, it is incapable of maintaining high data aggregation within every index page. This situation is particularly troublesome when data skewness happens. Other policies adopt real-time splits when emerging data are inserted into the full target leaves. Conventional node splits can follow unbalanced partition [8, 23], or alternatively, balanced partition [7, 15, 18, 27]. Overgrowing splits are typically derived from the situation that index hierarchy suffers from the improper data insertion order. The common outcome is that neighboring data already arranged in the same page are likely separated by force due to the subsequent data. Such scenario frequently fragments the data space. The handle of node splits is irreversible and should be invoked in the case that no alternative is suggested.

Traditional indexes resided in memory do not supply the cache fetches with sufficient retrieval quality. This is because the tremendous quantity of low space-utilization leaves dominates the index. *R**-tree [2] and extended *CIR*-tree [11] use the re-insertion techniques to reorganize parts of improper data arrangement in leaves. Although fewer splits are performed, the insertion routine with costly overhead is called more often in restructuring the indexing structures. Too many node splits still linger in them so as to cause cache quality and cache misses stay at an unsatisfactory level.

The Time-Parameterized *R*-tree (*TPR*-tree) [22] and its descendants such as *TPR**-tree [25], *PR*-tree [3], and *NSI* tree [21] represent objects' movements as motion function, so that updates are triggered by the changes of function parameters. Linear function with the minimum number of parameters and the interpolation method are employed to describe more complex movements. As revealed in [6], although the use of linear rather than constant functions reduces the need for updates by a factor of three, update performance still remain low when a number of parameters are involved.

B^x-tree [10] adopts another data model which combines object locations with their update timestamps. Using the taxonomy proposed in [1], B^x-tree only considers transaction-time of a fact. Insertions and deletions can only happen at the current time, namely, no update is allowed in the past. Basically, B^x-tree is B^+-tree which comprises several sub-partitions. The purpose of index partition is to differentiate objects based on different update timestamps. While this widens the B^+-tree and consequently increases the storage requirement for the entire structure. Spatial proximity is preserved within each partition, so quick queries are only partially facilitated on a restricted condition when objects with near update time are queried. In the case that similar objects with near proximity but are updated at different timestamps, multiple or concurrent data accesses in different index partitions are necessary. Unfortunately, the characteristic of single search path in classical B^+-trees is exacerbated by the concurrency control algorithms of B^x-tree. Another inefficiency of B^x-tree is the low space utilization of 50%-70% which is inherited from its base structure of B^+-tree.

In order to trace the variation in the movement of moving objects from time to time, present spatio-temporal indexes propose many update methods. *TB*-tree [20], *SETI* [4], and *SEB*-tree [24] handle spatial dimension by using trajectory segments and all movements corresponding to the segments must be known in advance of index processing. Namely, only closed trajectories with explicit start and end positions and timestamps can be handled. Exceptional operations are required when movements last till now. The 2-3 *TR*-tree [1] keeps a two-dimensional *R*-tree for the current objects and a three-dimensional *R*-tree for the historical data. *LUR*-tree [12] is only concerned with the current positions. So, as soon as objects change their locations, old entries are deleted and new entries are inserted. Another taxonomy [17] categorizes the indexing schemes into three categories: past, current, and now plus the future. In the past two decades, *R*-tree and *B*-tree play two major roles in the evolution of spatio-temporal access methods. Especially, *R*-tree and the adoption of MBRs dominate this evolution because more than 70% of the spatio-temporal access methods [3, 21, 22, 25] are based on the relevant design issue. *TPR**-tree receives a lot of attractions as its nearly-optimal query performance is validated in the last decade. As well known, node splitting based on MBRs exhibits high concurrency overheads and hence each individual update is quite costly. On the other hand, the simple implementation and practical popularity of *B*-tree have also made it feasible to index the scalar values such as the timestamps and the end-points of trajectory segments.

This paper proposes a novel indexing framework based on compressed B^+-trees without compromising on update and query efficiencies. The advantages of using compressed B^+-trees are multiple. First, B^+-tree is widely used in commercial database systems and its variation compressed B^+-tree has been proven to be very efficient [13, 14] when executing update and query operations. Second, compressed B^+-tree relies on one-dimensional indexing and it does not exhibit the overlapping problems associated with MBR-based indexes. Third, B^+-tree is typically appropriate to model the valid- and transaction-time of a fact. Therefore, to a certain extent, modeling the trajectories of moving objects is identical to the processing of end-points of trajectory segments.

We analyze the effect of our proposed framework and make the following contributions:

(1) Bitemporal data handling: data processing with both valid- and transaction-time.
(2) Fully evolving: the cardinality of the database can change and the objects can change their locations and outlines.
(3) The storage requirement is remarkably reduced so that the database indexes can completely reside in main memory with an ease.
(4) No restructuring is required; thereby no structural adjusting overhead is requested.
(5) Only single search paths are required; thereby data access complexity is significantly reduced.

The rest of this paper is organized as follows. Section 3 demonstrates our indexing scheme and shows how this scheme collaborates with compressed B^+-trees. For the complete understanding of the difference among the mentioned approaches and our method, Section 4 analyzes the hierarchy complexity and query performance in detail. Section 5 evaluates system and query performance by means of a large volume of simulated data. Meanwhile, some factors that affect the query performance are investigated in detail. In Section 6, we summarize our key results.

3 New indexing scheme

For the simplicity in demonstrating our design, we assume objects moving and/or changing their boundaries on a two-dimensional space so that the extension to a three-dimensional data space is straightforward. An example of such a spatiotemporal evolution appears in Fig. 1. The *x* and *y* axes represent the two-dimensional space while the *time* axis corresponds to the time dimension. Time dimension in our discussion retains continuous and non-decreasing. The snapshot of data space is displayed at every marked timestamp in Fig. 1. These snapshots look like video frames in which region objects are enclosed by MBRs. As time goes by, the extents of MBRs may change so that the variations in the boundary of MBRs deliver more information than those in the area of MBRs.

At first, i.e. time=t_1, region object O_1 and point object O_2 come to the two-dimensional space. At time=t_2, point object O_3 appears, O_1 expands and moves to a new area, O_2 stays still at the same position. During the time period between t_2 and t_4, O_1 and O_3 continue their motions and change their outlines; while O_2 always keeps its stillness in this during.

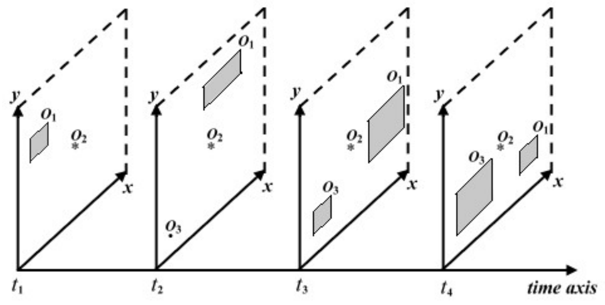


Fig.1 The snapshots of objects from t_1 to t_4

The primary information preserved in our indexing structures includes the critical positions and the critical time, where and when objects change their movements. The involved space and valid time corresponding to every movement are subsequently handled and preserved in our indexing structures. Taking object O_1 and its enclosing MBR as an example for illustration, Figure 2 outlines the movements in two-dimensional space from t_1 to t_4. The x projected segments listed below the coordinate sketch the involved space at different timestamps. The solid dots and circles represent the start and end locations of involved space, respectively. Then, the information related to the horizontal transition of O_1's MBR can be preserved by the two polygonal lines depicted in Fig. 3. The vertical transition of O_1's MBR can also be outlined by the similar process.

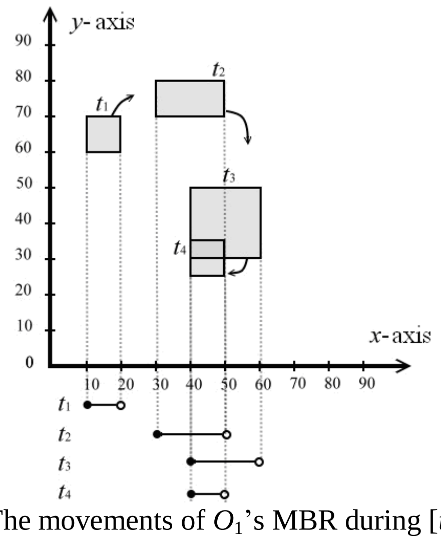


Fig.2 The movements of O_1's MBR during [t_1, t_4]

In Fig. 3, s_1, s_2, e_1, and e_2 are trajectory segments representing onward movements. Trajectory segment e_3 and s_3 represents backward and still movements, respectively. Vertical segment is unreasonable because the position can not be confirmed at a specific timestamp.

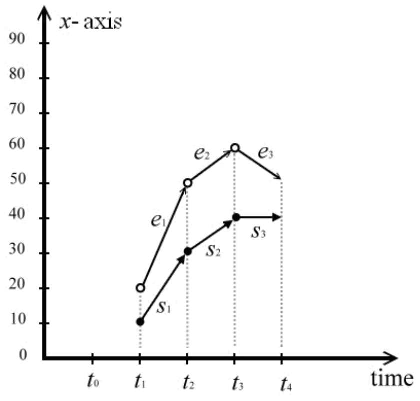


Fig. 3 The horizontal transition of O_1's MBR

3.1 The compressed B^+-tree

The base structure of compressed B^+-tree (CB^+-tree) is classical B^+-tree. The insertion algorithm of CB^+-tree takes advantage of unused space at leaf level. As shown in the step 4 of Fig.4(a), data 5 and 6 are removed from the right leaf and shifted to the left leaf. Then, datum 15 can be inserted into the right leaf. Since the variance of {1,3,4,5,6} plus the variance of {8,9,12,15} is less than that of {1,3,4,5} plus {6,8,9,12,15}, the redistribution of data 5 and 6 achieves a better data classification which is more aggregative than that only shifting datum 5. Classical B^+-tree allocates more leaves and consequently has a low space utilization as shown in Fig.4(b).

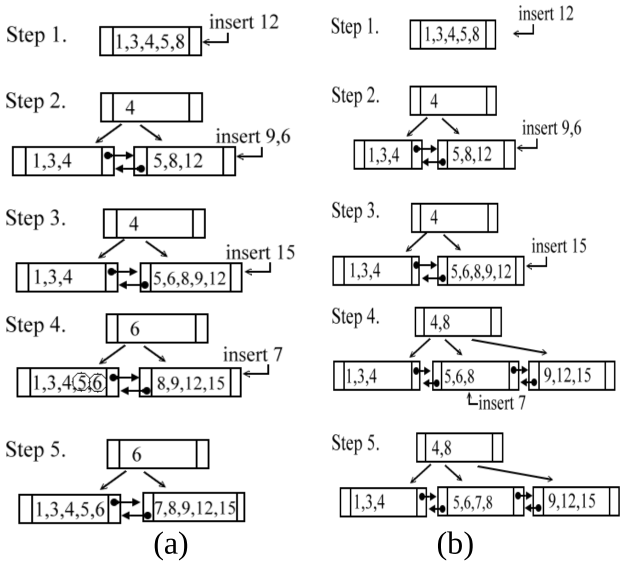


Fig. 4(a) A compressed B^+-tree and (b) a classical B^+-tree

The temporal and spatial attributes of trajectory segments corresponding to the MBRs of moving objects are processed by separate CB^+-trees. The resulting indexes are temporal and spatial CB^+-trees which are abbreviated as TCB^+-tree and SCB^+-tree, respectively. The detail processing are presented in the following subsections.

3.2 TCB^+-tree

In generally, the generation of critical timestamps for TCB^+-tree is strictly increasing. So, the insertion of new timestamp simply appends to the rightmost side of TCB^+-tree. Figure 5 shows the TCB^+-tree built by inserting the critical timestamps derived from O_1. The content of array $\Gamma(t_k)$ referred by the leaf entry t_k is fulfilled by the following criterion.

$$\Gamma(t_k)=\{s_i \text{or} e_i \mid [t_k, t_k^+] \cap (|s_i^t| \text{or} |e_i^t|) \neq \phi \text{ and} (|s_i^t| \text{or} |e_i^t|) > 0\} \tag{1}$$

, where $|s_i^t|$ or $|e_i^t|$ is the projected length on the temporal dimension. Term t_k^+ (or t_k^-) is defined as the indexed timestamp right after (or before) t_k, so that $t_k<t_k^+$ (or $t_k^-<t_k$). Namely, no indexed timestamp t_m exists such that $t_k^-<t_m<t_k$ or $t_k<t_m<t_k^+$. As shown in Fig. 3, s_1 and e_1 are valid during $[t_1, t_2]$. So, $\{e_1, s_1\}$ is referred by the leaf entry t_1 of Fig.5.

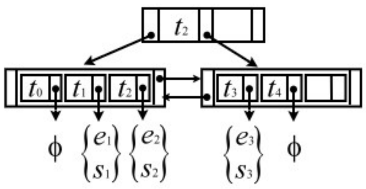


Fig. 5 TCB^+-tree

In case that continuous temporal data are generated, consecutive insertions fast extend the right side of TCB^+-tree. Since these insertion data are totally ordered, the index hierarchy can easily retain compactness after all data are organized. The compression mechanism of CB^+-tree maintains high data aggregation and high space utilization within leaf level when other operations (deletion or updating) are executed.

3.3 SCB^+-tree

Similar to TCB^+-tree, the content of array $\Gamma(x_k)$ referred by the leaf entry x_k of SCB^+-tree is fulfilled by the following criterion.

$$\Gamma(x_k)=\{s_i \text{or} e_i \mid [x_k, x_k^+] \cap (|s_i^x| \text{or} |e_i^x|) \neq \phi \text{ and} (|s_i^x| \text{or} |e^x|) > 0\} \tag{2}$$

Figure 6 shows the building procedure of SCB^+-tree when the critical spatial information of O_1's MBR are handled. Similar to TCB^+-tree, the bottom structure of SCB^+-tree is composed of a sequence of arrays referred by leaf entries. In order to label different moving directions in the generated arrays, terms s_i, s_i', and $\overline{s_i}$ (e_i, e_i', and $\overline{e_i}$) are used to denote onward, backward, and still motions, respectively. The motions happened in $[t_3, t_4]$ are taken for illustration. Object O_1's MBR holds its start position and varies its end position from a higher position to a lower in this duration. Precisely, by way of entry 40, $\overline{s_3}$ is finally referred in the bottom structure of the resulting index of Fig. 6. As to e_3', the apostrophe hints that the higher position is 50^+ (=60) and the lower position is 50.

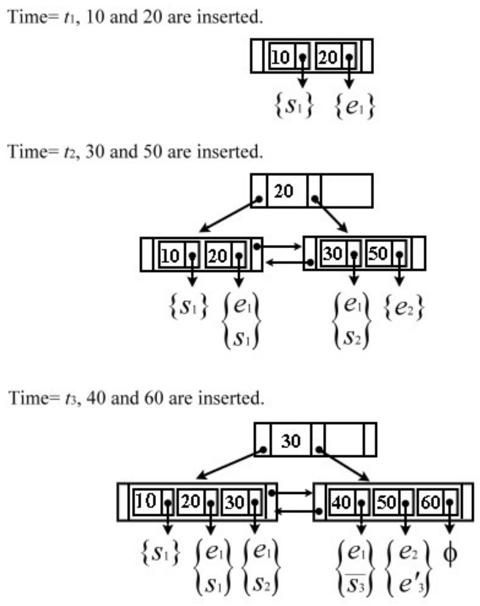


Fig. 6 SCB^+-tree

Query procedures over TCB^+-tree or SCB^+-tree are identical to conventional B^+-tree. Due to the succinct index hierarchies with TCB^+-tree and SCB^+-tree, they can easily reside at the main memory so that the concurrent execution can be simply fulfilled as well. Point and interval queries are two typical operations that users often specify. One point query over TCB^+-tree or SCB^+-tree only triggers one single searching path. One interval query needs two single searching paths to embrace the related information. Users who are attracted by specific timestamps and locations will launch point queries over TCB^+-tree and SCB^+-tree. Time periods and spatial areas when and where are concerned by users will trigger interval queries.

As a result, all query types are initialized as the form of $Q(T, S_1, S_2, ..., S_n)$, where T is a specific timestamp or period and S_i is a specific location or interval on the i-th dimension, $i \in [1,2,...,n]$. For example, concentrating our attention on x-dimensional space, $Q(t, x_a, x_b)$ and $Q(t_a, t_b, x_a)$ respectively issue time-slice and location-slice queries. Window query $Q(t_a, t_b, x_a, x_b)$ specifies a spatial interval that is valid for a time interval. $Q(now, x_a, x_b)$ specifies a spatial interval on the instant. $Q=(t, now, x_a, x_b)$ is a window query retrieving those movements that keep moving and pass through $[x_a, x_b]$ during $[t, now^+]$. In case timestamp *now* is involved in the query, the movements with lasting inertia motions to the near future are retrieved. Such case is predictive queryies for the anticipated future. In addition, users can issue moving query $Q(t_a, t_b, x_a, x_b, x_c, x_d)$ which specifies (x_a, x_b) at time t_a and (x_c, x_d) at time t_b.

4 Index efficiency analyses

*TPR**-tree and B^x-tree are taken as the competitor schemes against our method. *TPR**-trees deal with time parameterized boundaries of MBRs (minimum bounding rectangles for spatial data) and VBRs (velocity bounding rectangles). B^x-trees deal with movement vectors together with update time. Different data models lead to the different designs of data classification and data comparison methods. For the completeness of this comparison, hierarchy complexities including tree order, index depth, and total amount of tree nodes are carefully investigated. For the precise grasp of query execution efficiency of these spatio-temporal access methods, the amount of generated data entries, the storage requirement of whole index, the number of nodes accessed, and the CPU time are investigated as well.

4.1 Efficiency analysis

The notations adopted in our analysis are listed in Table 1. We start our analysis by first giving a definition of the tree order for CB^+-tree, *TPR**-tree, and B^x-tree.

Table 1 Notations for analysis

Symbol	Description
B	block size for one page
N	the total data amount generated from moving objects
N_L	number of leaf nodes
C_L	maximum capacity of leaf nodes
h	the depth (height) of tree.
m	the tree order (maximal fan-out)
t	byte number for one integer
p	byte number for one linking pointer

LEMMA 1. *The tree order m of a* CB^+*-tree is* $\left\lfloor \frac{B-2p}{t+p} \right\rfloor$.

Proof: Every entry in the leaf or non-leaf node of a CB^+-tree comprises one index key and one pointer. Except the root, every node of CB^+-tree retains two pointers to link its fore and next sibling. Generally, one memory block corresponds to one index page (node) in the computer system. The order m of a CB^+-tree is evaluated as following:

$$m \cdot (t+p) + 2p = B \ \Rightarrow m = \left\lfloor \frac{B-2p}{t+p} \right\rfloor \qquad (3)$$

In a general implementation, 4 Kbytes is a proper size for one memory block. One integer number usually occupies 16 bytes and one pointer occupies 32 bytes. Therefore, the order of a CB^+-tree is approximated as 84. In case of using *TPR**-tree, every entry contains one pointer linking to a sub-tree and one rectangle that bounds the positions of all moving points or other bounding rectangles in that sub-tree. One n-dimensional rectangle needs $2n$ values to position its location in the space. So, in case n=2, *TPR**-tree's order is approximated as following:

$$m \cdot (4t+p) + p = B \ \Rightarrow m = \left\lfloor \frac{B-p}{4t+p} \right\rfloor \qquad (4)$$

Furthermore, applying the same conditions as previous, the order of a *TPR**-tree is deduced as only half of that in a CB^+-tree. In case of using B^x-tree, besides the pointer to the next sibling, a leaf entry of B^x-tree corresponding to one movement updated at certain timestamp has the representation composed by a position vector ($\vec{x}$), a velocity vector ($\vec{v}$), and an update time (t_u). Five integers and one pointer are preserved in each leaf entry of B^x-tree when two-dimensional motions are handled. The B^x-tree's order is formulated as $\left\lfloor \frac{B-P}{5t+p} \right\rfloor$ which is evaluated as 36 under the same conditions as previously mentioned. In summary, the order ratio between these three indexes is 1 :0.5 :0.43.

LEMMA 2. The ratio of index depth among CB^+-tree, *TPR**-tree, and B^x-tree is approximated as

$$h_1 : h_2 : h_3 = \left\lceil \log_{(m\ln 2)} \frac{N}{m} \right\rceil + 1 : \left\lceil \log_{(0.5m\ln 2)} \frac{N}{0.5m\ln 2} \right\rceil + 1 : \left\lceil \log_{(0.43m\ln 2)} \frac{N}{0.43m\ln 2} \right\rceil + 1 \cdot$$

Proof: The number of leaf nodes (N_L) of CB^+-tree can be estimated as $\left\lceil \frac{N}{m} \right\rceil$ because the high accommodation rate near 100% is ensured in its external structure. However, in the internal structure, the average space utilization of non-leaf nodes is about $\ln 2 \times 100\% = 69.3\%$ [26]. So, the depth of *CB*$^+$-tree (h_1) is approximated as $\left\lceil \log_{(m\ln 2)} \frac{N}{m} \right\rceil + 1$.

The node splitting strategy employed in *R**-tree and consequently adopted in *TPR**-tree lacks the ability to compactly aggregate data entries in their external structures. Their leaf nodes still have the low space utilization less than $\ln 2 \times 100\%$ on average. This leads to the difficulty to organize the succinct data arrangement in the internal structure of *TPR**-tree. With the same page size as that used in building *CB*$^+$-tree, the N_L of *TPR**-tree is approximated as $\left\lceil \frac{N}{0.5m\ln 2} \right\rceil$, where 0.5*m* is the order of *TPR**-tree. The depth of *TPR**-tree (h_2) is then approximated as $\left\lceil \log_{(0.5m\ln 2)} \frac{N}{0.5m\ln 2} \right\rceil + 1$. Regarding *B*^{x}-tree, the total data amount *N* depends on the number of timestamps that objects are inserted or updated. The update time dominate the entry values and decide the exact portion of index partition. The depth of *B*^{x}-tree (h_3) is approximated as $\left\lceil \log_{(0.43m\ln 2)} \frac{N}{0.43m\ln 2} \right\rceil + 1$, where the order of *B*^{x}-tree is 0.43*m*.

COROLLARY 1. Based on the same data size *N*, the difference between the depth of *TPR**-tree and that of *CB*$^+$-tree is $\left\lceil \log_{(m\ln 2)} m \cdot 2^k \right\rceil - 1$, where $k = \log_{(0.5m\ln 2)} N$.

Proof: Deduced from Lemma 2, the depth difference ($\triangle h$) between *TPR**-tree and *CB*$^+$-tree is

$$h_2 - h_1 = \left(\left\lceil \log_{(0.5m\ln 2)} \frac{N}{0.5m\ln 2} \right\rceil + 1\right) - \left(\left\lceil \log_{(m\ln 2)} \frac{N}{m} \right\rceil + 1\right) \quad (5)$$

$\approx \log_A(\frac{A^k}{A}) - \left\lceil \log_{2A}(\frac{A^k}{m}) \right\rceil$, where $A = 0.5m\ln 2$ and $N = A^k$.

$$= (k-1) - \left\lceil \log_{(2A)}[\frac{(2A)^k}{m \cdot 2^k}] \right\rceil$$
$$= (k-1) - [k - \left\lceil \log_{(2A)} m \cdot 2^k \right\rceil]$$
$$= \left\lceil \log_{(2A)} m \cdot 2^k \right\rceil - 1$$
$$= \left\lceil \log_{(m\ln 2)} m \cdot 2^k \right\rceil - 1$$

As *k* varies from 10 to 50 data sizes vary from $(\frac{84 \times \ln 2}{2})^{10}$ to $(\frac{84 \times \ln 2}{2})^{50}$ when *m* equals 84. Table 2 lists some examples and reveals *SCB*$^+$- and *TCB*$^+$-trees significantly improve index size when data sizes grow.

Table 2 Depth difference

k	10	20	30	40	50
$\triangle h$	2	4	6	7	9

COROLLARY 2. Based on the same data size, the total number of nodes generated by *SCB*$^+$-tree (or *TCB*$^+$-tree), *TPR**-tree, and *B*^{x}-tree are approximated as the ratio of $\frac{1}{m\ln 2 - 1} : \frac{0.5}{0.5\ln 2 \cdot (0.5m\ln 2 - 1)} : \frac{0.43}{0.43\ln 2 \cdot (0.43m\ln 2 - 1)}$.

Proof: Deduced from Lemma 2, the total number of nodes generated by *SCB*$^+$- or *TCB*$^+$-tree is

$$\left\lceil \frac{N}{m} \right\rceil \cdot (1 + \frac{1}{m\ln 2} + \cdots) \approx \left\lceil \frac{N}{m} \right\rceil \cdot (\frac{m\ln 2}{m\ln 2 - 1}) \quad (6)$$

The total number of nodes generated by *TPR** is

$$\left\lceil \frac{N}{0.5m\ln 2} \right\rceil \cdot (1 + \frac{1}{0.5m\ln 2} + \cdots) \approx \left\lceil \frac{N}{0.5m\ln 2} \right\rceil \cdot (\frac{0.5m\ln 2}{0.5m\ln 2 - 1}) \quad (7)$$

The total number of nodes generated by *B*^{x}-tree is

$$\left\lceil \frac{N}{0.43m\ln 2} \right\rceil \cdot (1 + \frac{1}{0.43m\ln 2} + \cdots) \approx \left\lceil \frac{N}{0.43m\ln 2} \right\rceil \cdot (\frac{0.43m\ln 2}{0.43m\ln 2 - 1}) \quad (8)$$

The ratio among them can be simplified as $\frac{1}{m\ln 2 - 1} : \frac{0.5}{0.5\ln 2 \cdot (0.5m\ln 2 - 1)} : \frac{0.43}{0.43\ln 2 \cdot (0.43m\ln 2 - 1)}$ after reducing the common parts. We obtain an approximation of 1: 2.94: 3.43 when replacing *m* by 84.

4.2 Query performance analysis

Query performance is deeply affected by the complexity of index hierarchy. Applying queries over the higher index usually accesses more data pages (nodes) from the index. Data model with complex design always consumes much CPU time to advance data comparisons. In addition, data retrieval with insufficient quality is another cause deteriorates query performance. Single searching paths with short length, simple data comparison mechanism, and sufficient data provision in every retrieved page are our achievements in *SCB*$^+$-tree and *TCB*$^+$-tree. Following corollaries investigate the performance of point and interval queries.

COROLLARY 3. The point query performance among *CB*$^+$-tree, *TPR**-tree, and *B*^{x}-tree is a ratio of $h_1 : ch_2 : h_3$, where $c > 1$.

Proof: Based on Lemma 2, the processing of every point query over *CB*$^+$-tree and *B*^{x}-tree respectively access h_1 and h_3 nodes. The experiments conducted in *R**-tree [2] had pointed out that the average number of disc accesses (nodes) per point query is about 1.5~2 times of the index depth. So, the point query performance of *TPR**-tree is approximated as the complexity of multiple h_2's.

As two-dimensional moving objects are concerned, one TCB^+-tree and two SCB^+-trees with similar index hierarchies are built. Even thought the case that temporal and spatial queries can not execute concurrently, at most $3h_1$ nodes are accessed sequentially in TCB^+- and SCB^+-trees for every point query.

COROLLARY 4. The time complexity of interval query among CB^+-tree, *TPR**-tree, and B^x-tree is approximated as a ratio of $O(h_1):O(\varpi^{h_2}):O(h_3)$, where $\varpi > 1$.

Proof: In addition to the nodes accessed on two single searching paths, some consecutive leaf nodes are traversed as well for the coverage of whole query range. So, $2h_1 + c$ nodes are involved and thus a time complexity of $O(h_1)$ is deduced for CB^+-tree. As far as B^x-tree is concerned, the query complexity is approximated as $2h_3 + c \approx O(h_3)$. However, as the experimental results presented in *R**-tree, the wider range one query covers the more time cost the query operation spends. The number of nodes accessed is several times or even exceeds 10 times of the index depth. In regard to *TPR**-tree, a factor ϖ indicating the overlapping degree among MBRs is used to predict the branch number involved by the multiple searching paths for every range query. This factor is obtained by dividing the summation of all MBRs' ranges generated at all levels with the union area of these MBRs. Obviously, the query with wider range causes the more branches for search. The design of ϖ is based on a simplified consideration. Therefore, the time complexity of range queries over *TPR**-tree is estimated as $1 + \varpi + \varpi^2 + \cdots + \varpi^{h_2 - 1} \approx O(\varpi^{h_2})$.

5 Performance evaluation

5.1 Experiment setup

Datasets with object sizes varying from 500 to 5000 are generated randomly. The underlying space is based on a two-dimensional image with a resolution of $10^4 \times 10^4$ pixels. The time period considered in our experiments is *T*[0,4999]. The time unit is second. Not only point objects but also regional objects are collected into a dataset for the generation of movements. Initially, the starting position for each movement is uniformly selected from the underlying space and the starting timestamp is uniformly selected from *T*[0,4999]. All motions could have a short appearance or last their movements for a while. Motion directions might be forward, backward, or still. In this time period, point objects can grow into regional objects, regional objects can shrink into point objects, existent objects can vanish, or new objects can be generated. In our experiments, the life time of motions follow certain types of exponential distribution. One object can produce so many continuous movements that the number of trajectory segments for indexing is far more than the number of objects. The moving speed for every piece of movement and the changing speed for every MBR's outline follow the normal distribution with parameters $\mu = 0$ and $\sigma \in \{2,6,10\}$, where speed unit is pixel/second.

5.2 The amount of generated data entries

We first count the number of data entries generated for completing the whole index. *TPR**- and B^x-trees transform every linear movement into a single data entry. Based on their data models, data corresponding different movements are handled individually. But, in SCB^+- and TCB^+-trees, the temporal and spatial data are extracted based on their variations. No matter what objects or what motions are considered, several variations happen at the same location or on the same time only appeal to one spatial data or one temporal data for indexing. This is the reason why CB^+-trees generate fewer data than *TPR**- and B^x-trees as shown in Fig. 8. The generated spatial data have higher repetition than the generated temporal data so that SCB^+-trees create fewer data than TCB^+-trees. This difference is also revealed in Fig. 8.

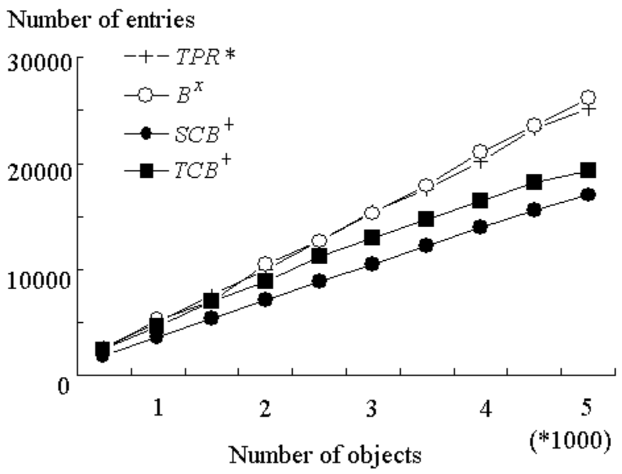
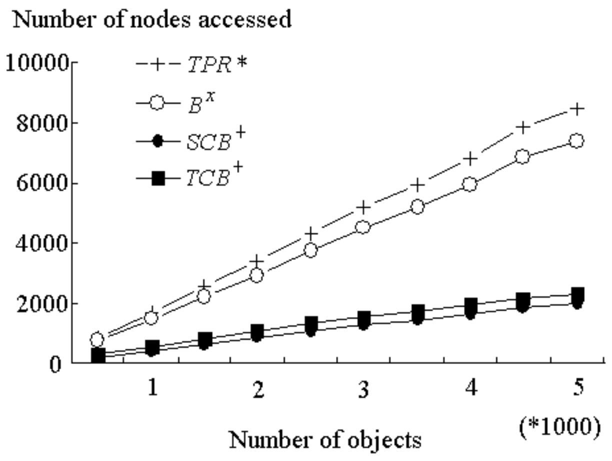


Fig. 8 The number of entries generated for different indexes.

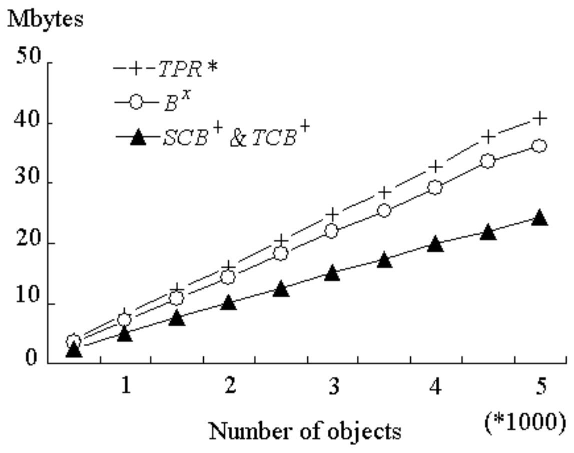
5.3 Storage requirement

In order to promote system performance, more and more databases in modern applications request to cache all indexing structures in main memory. Thus, storage requirement becomes a bottleneck when developing new indexing structures. Figure 9(a) shows the amount of leaves, in which *SCB*$^+$-trees generate the least number of all and *TCB*$^+$-trees generate a slight higher amount. There are three factors that cause *TPR**- and *B*^{x}-trees to assign a large number of leaves. The first factor is explained in Subsection 5.2. Data models adopted in *TPR**- and *B*^{x}-trees allocate more memory bytes in representing their data entries. Data accommodations in *TPR**- and *B*^{x}-trees are so low that results in the second factor. The third factor derives from the low space utilization in the base structures of *R**- and *B*$^+$-trees. These factors cause *TPR**- and *B*^{x}-trees frequently allocate new leaves and in turn inflate the index sizes.

Furthermore, the internal structure together with the external structure of resulting *TCB*$^+$- and *SCB*$^+$-trees are taken into account. Temporal and spatial structures based on the same dataset are taken as the whole one for storage requirement analysis. The experimental results as shown in Fig. 9(b) validate that *SCB*$^+$- and *TCB*$^+$-trees employ the storage requirement with better efficiencies.



(a)



(b)

Fig. 9. (a) The number of leaves and (b) the scale of storage requirement.

5.4 Range query

In this subsection, we study the performance of range query when varying the number of uniformly distributed moving objects from 500 to 5000. This paper assumed that the length and width of query size have the uniform distribution $U(10^2, 10^3)$. We first generate 100 different sizes. For each size, 20 locations are selected from the underlying space as the left-lower corners to locate 20 range queries. The number of nodes accessed for each located range query is counted and 20 results from the same query range are averaged for performance analysis. As shown in Fig. 10, *B*^{x}- and *CB*$^+$-trees have a stable performance and the amount of involved nodes always stay at the low level. The performance of *TPR**-trees not only suffer from the high amount of involved nodes, but this situation is inflamed by the increase of data size. When *SCB*$^+$- and *TCB*$^+$-trees are queried concurrently (the index system is abbreviated as concurrent *CB*$^+$), the amount of nodes accessed in this system is evaluated as the higher number between the results involved in *SCB*$^+$- and *TCB*$^+$-trees.

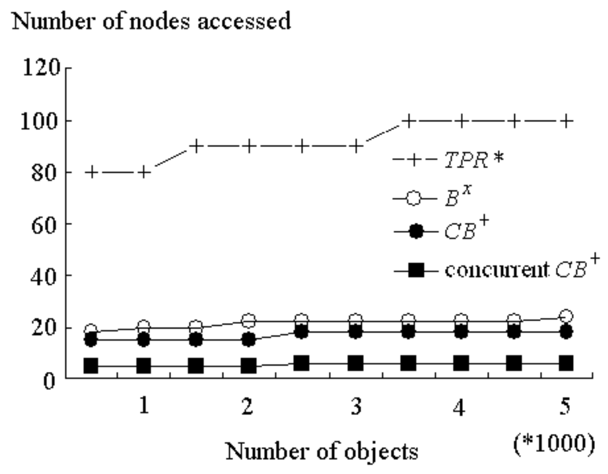


Fig. 10 The range query performance

5.4.1 Effect of speed rates

In order to investigate how the factor of speed rate affects range query performance, we simulate to change the moving and shrink-expand speeds of objects. This paper assumed that these two types of speed have the normal distribution ranging from 2^1 to 2^{10} pixels/second. Figure 11 shows that *TPR**-trees continue to grow the amount of accessed nodes when increasing speed rates. This is because the high speed rates tend to expand the cover areas of MBRs. And this consequently leads to that the overlaps between MBRs become more and more serious. Finally, query ranges are so easily sunk into dead space that a large portion of indexed pages is involved in the processing.

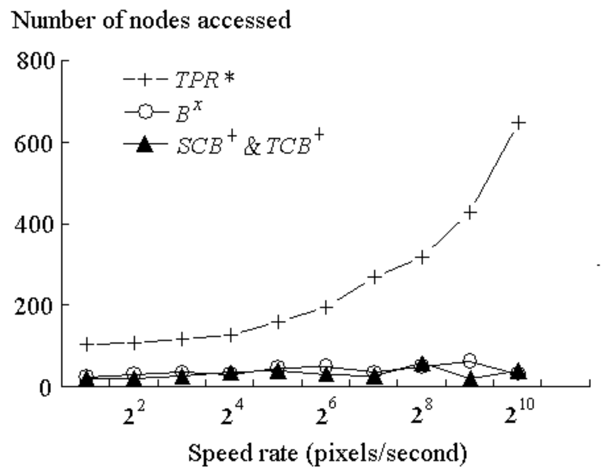


Fig. 11 Effect of speed rate

5.4.2 Effect of lasting time

In order to study the effect of movement's lasting time on query performance, it is assumed that objects move and/or change with a speed rate of 10 pixels/second. This experiment assumed lasting time has the exponential distribution with parameter λ varying from 5 to 55. Figure 12 shows that the number of nodes accessed from *TPR**-trees still stays at the highest level of all. In contrast, the performance of B^x- and CB^+-trees are not affected by the variations of lasting time. This experimental result once again verifies that our proposed method can work with a stable and efficient performance.

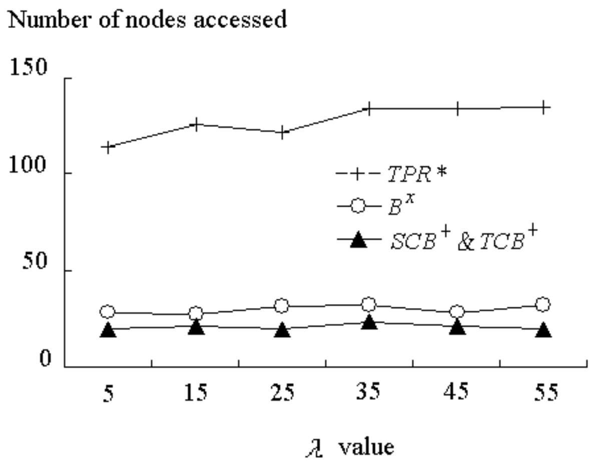


Fig. 12 Effect of lasting time

6 Conclusions and future work

In this paper we have proposed and evaluated a new spatio-temporal indexing mechanism. Temporal and every single spatial dimension are processed separately by compressed B^+-trees. Our TCB^+- and SCB^+-trees benefit from the aggregative data arrangement inside the indexed pages and achieve the high update and query efficiencies. Logically, our designs are easily implemented by modifying the existing B^+- or B*-tree-based database systems.

For future work, we plan to experiment with a number of extensions to high dimensional applications. Currently, based on our proposed model, *n*-dimensional moving objects impose *n*+1 CB^+-trees. Although succinct index hierarchies are achieved, system implementation with too many indexes is still problematic. One possible solution is to filter dimensions before processing index scheme. The decision factors can be so complex that they include practical application need, query accuracy, system efficiency, system effectiveness, and others. Another possible research direction is to develop the data mining technologies for spatio-temporal data. It is highly expected to discover the useful techniques suitable for the handle of interaction between spatial and temporal data.

References:

[1] M. Abdelguerfi, J. Givaudan, K. Shaw, R. Ladner,

The 2-3 TR-tree, A Trajectory-Oriented Index Structure for Fully Evolving Valid-time Spatio-temporal Datasets, *In: Proc. of the ACM workshop on Advances in Geographic Information Systems*, ACM GIS, 2002, pp. 29-34.

[2] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An Efficient and Robust Access Method for Points and Rectangles, *In: Proc. of ACM SIGMOD International Conf. on Management of Data*, 1990, pp. 322-331.

[3] M. Cai, P. Revesz, Parametric R-Tree: An Index Structure for Moving Objects, *In: Proc. of the International Conference on Management of Data*, COMAD, 2000.

[4] V.P. Chakka, A. Everspaugh, J.M. Patel, Indexing Large Trajectory Data Sets with SETI, *In: Proc. of the Conf. on Innovative Data Systems Research*, CIDR, 2003.

[5] E. Chávez, G. Navarro, A Compact Space Decomposition for Effective Metric Indexing, *Pattern Recognition Letters*, Vol.26, No.9, 2005, pp. 1363-1376.

[6] A. Civilis, C.S. Jensen, J. Nenortaite, S. Pakalnis, Efficient Tracking of Moving Objects with Precision Guarantees, *In: Proc. MobiQuitous 2004, The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004, pp. 164-173.

[7] C.H. Goh, A. Lim, B.C. Ooi, K.L. Tan, Efficient Indexing of High-dimensional Data through Dimensionality Reduction, *Data and Knowledge Engineering*, Vol.32, 2000, pp. 115-130.

[8] A. Henrich, H.S. Six, P. Widmayer, The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects, *In: Proc. of 15th VLDB Conf.*, 1989, pp. 45-53.

[9] H. Jagadish, B.C. Ooi, K.L. Tan, C. Yu, R. Zhang, iDistance: An Adaptive B^+-tree Based Indexing Method for Nearest Neighbor Search. *ACM, Transactions on Database Systems* (*TODS*), Vol.30, No.2, 2005, pp. 364-397.

[10] C.S. Jensen, D. Lin, B.C. Ooi, Query and Update Efficient B^+-tree Based Indexing of Moving Objects, *In: Proc. of the 30th VLDB Conf.* Toronto, Canada, 2004.

[11] J. Kim, D.Y. Hur, J.S. Lee, J.S. Yoo, S.H. Lee, Insertion Method in A High-dimensional Index Structure for Content-based Image Retrieval, US Patent 6,389,424 (2002).

[12] D. Kwon, S. Lee, S. Lee, Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree, *In: Proc. of Third International Conf. on Mobile Data Management*, 2002, pp. 113-120.

[13] H.Y. Lin, Efficient and compact indexing structure for processing of spatial queries in line-based databases, *Data Knowl. Eng*, Vol.64, No.1, 2008, pp. 365-380.

[14] H.Y. Lin, Using B^+-trees for processing of line segments in large spatial databases, *J. Intell. Inf. Syst*, Vol.31, No.1, 2008, pp. 35-52.

[15] K. Lin, H. Jagadish, C. Faloutsos, The TV-tree: An Indexing Structure for High-dimensional Data, *VLDB Journal*, Vol.3, 1995, pp. 517-542.

[16] J. Lukaszuk, R. Orlandic, On Accessing Data in High-Dimensional Spaces: A Comparative Study of Three Space Partitioning Strategies, *Journal of Systems and Software*, Vol.73, No.1, 2004, pp. 147-157.

[17] M.F. Mokbel, T.M. Ghanem, W.G. Aref, Spatio-Temporal Access Methods, *IEEE Data Eng*, Bull, Vol.26, No.2, 2003, pp. 40-49.

[18] R. Orlandic, B. Yu, A Retrieval Technique for High-dimensional Data and Partially Specified Queries, *Data and Knowledge Engineering*, Vol.42, No.1, 2002, pp. 1-21.

[19] N. Pelekis, B. Theodoulidis, I. Kopanakis, Y. Theodoridis, Literature Review of Spatiotemporal Database Models. *The Knowledge Engineering Review*, 2005-Cambridge Univ. Press.

[20] D. Pfoser, C.S. Jensen, Y. Theodoridis, Novel Approaches in Query Processing for Moving Object Trajectories, *In: Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, 2000, pp. 395-406.

[21] K. Porkaew, I. Lazaridis, S. Mehrotra, Querying Mobile Objects in Spatio-Temporal Databases, *In: Proc. of the Intl. Symp. on Advances in Spatial and Temporal Databases, SSTD*, Redondo Beach, CA, 2001, pp. 59-78.

[22] S. Saltenis, C.S. Jensen, S.T. Leutenegger, M.A. Lopez, Indexing the positions of continuously moving objects, *In: Proc. of the ACM International Conf. on Management of Data. Dallas*, USA: ACM Press, 2000.

[23] B. Seeger, H.P. Kriegel, The Buddy-tree: An Efficient and Robust Access Method for Spatial Database Systems, *In: Proc. of 16th VLDB Conf*, 1990, pp. 590-601.

[24] Z. Song, N. Roussopoulos, SEB-tree: An Approach to Index Continuously Moving Objects, *In: Mobile Data Management, MDM*, 2003, pp. 340-344.

[25] Y. Tao, D. Papadias, J. Sun, The TPR*-tree: An Optimized Spatio-temporal Access Method for Predictive Queries, *In: Proc. of the 29th International Conf. on Very Large Data Bases*, Berlin, Germany, 2003, pp. 790-801.

[26] A. Yao, Random 2-3 trees, *Acta Informatica*, Vol.2, No.9, 1978, pp. 159-170.

[27] B. Yu, T. Bailey, R. Orlandic, J. Somavaram, KDB_{KD}-Tree: A Compact KDB-tree Structure for Indexing Multidimensional Data, *In: Proc. of International Conf. on Information Technology, ITCC*, Las Vegas, 2003, pp. 676-680.