


Decomposition tree: a spatio-temporal indexing method for movement big data

Zhenwen He^{1,2}  · Chonglong Wu^{1,2} · Gang Liu^{1,2} · Zufang Zheng^{1,3} · Yiping Tian^{1,2}

Received: 27 June 2015 / Revised: 18 July 2015 / Accepted: 27 July 2015 / Published online: 11 August 2015
© Springer Science+Business Media New York 2015

Abstract Movement is a complex process that evolves through both space and time. Movement data generated by moving objects is a kind of big data, which has been a focus of research in science, technology, economics, and social studies. Movement database is also at the forefront of geographic information science research. Developing efficient access methods for movement data stored in movement databases is of critical importance. Tree-like indexing structures such as the R-tree, Quadtree, Octree are not suitable for indexing multi-dimensional movement data because they all have high space cost of their inner nodes. In addition, it is difficult to use them for parallel access to multi-dimensional movement data because they thereof, are in hierarchical structures, which have severe overlapping problems in high dimensional space. In this paper, we propose a novel access method, the Decomposition Tree (D-tree), for indexing multi-dimensional movement data. The D-tree is a virtual tree without inner nodes, instead, through an encoding method based on integer bit-shifting operation, and can efficiently answer a wide range of queries. Experimental results show that the space cost and query performance of D-tree are superior to its best known competitors.

Keywords Movement data · Big data · Spatio-temporal index · D-tree · Moving objects

✉ Gang Liu
liugang@cug.edu.cn

¹ School of Computer Science, China University of Geosciences, 388 Lumo Road, Wuhan 430074, China

² Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China

³ School of Industrial Design, Hubei University of Technology, Wuhan 430068, China

1 Introduction

Movement is a complex, continuous process that evolves through both space and time, and each object and event has both time and space. Movement data represent this continuous process of moving objects in the form of ‘sampled’ observations. Movement data generated by moving objects is a kind of big data which has been a focus of research in science, technology, economics, and social studies [1]. Moving objects, including humans, animals and vehicles, change their spatial locations over time. A wide range of applications has been created for moving objects, especially for mobile objects, including traffic surveillance, users of wireless devices, wildlife distribution, disease surveillance, etc. Because of the diversity and low cost of the tools used to capture the movement data, such as GPS devices and geo-sensor networks, these applications have generated a large amount of movement data.

Given the increasing focus upon real-time data capture, and massive data size, movement data are still hard to manage effectively and efficiently in a GIS [2]. The arrival of Cloud Computing [3,4] and Big Data [5,6] era increases the challenge of movement data. The traditional relational database and spatial database have few capabilities for handling movement data, and often only store the current status of moving objects. Otherwise, they must deal with the continuous updating workload that they cannot afford. In that case, the movement database, which is responsible for the management of movement data, appeared.

Index is the key to fast data retrieval in a movement database. In the last decade, a large number of spatio-temporal index structures have been proposed to support spatio-temporal queries for handling movement data. They include the TPR-tree [7,8], TPR*-tree [9,10], B^x-tree [11], LUGrid [12], RPPF-tree [13], B^{dual}-tree [14], B^y-tree [15],

RTR-tree and TP²R-tree [16], B^s-tree [17], STCB⁺-tree [18, 19], etc. Most of them are derived from either B-trees [20] or R-trees [21]. Those derived from R-tree employ a minimum bounding rectangle (MBR) or a time-parameterized minimum-bounding rectangle (TPMBR) to represent a moving object. The range of the bounding rectangle is always larger than the range of the moving object itself. This is the fundamental reason why they cannot resolve the overlapping problem in high dimensions. Moreover, employing a MBR or TPMBR to represent a moving point or a segment is not an adequate way because the space cost of MBR or TPMBR may be larger than a pure a moving point or segment. The indexes derived from B⁺-trees, mostly employ space filling curve transformations (SFC) or divide a moving point into multiple components. This class of indexes is suitable to index low dimensional data but not high dimensional data. Because the higher the dimension is, the more additional sub trees are needed and the computation on those result sets will increase dramatically.

Quadtrees and octrees are grid-like and tree-like. They are used in a variety of fields including computational geometry, image processing and some spatial access methods. However there are few indexes of movement data based on them. In contrast to the balanced B-trees and R-trees, they may be unbalanced due to the non-uniform distributions of movement data though they are still trees. Because the depth of an unbalanced tree is not controlled by algorithms but by the data distributions in the time and location space, the performance of the tree may be significantly reduced. Being a hierarchical structure, a tree contains many inner nodes that do not contain real index items of moving objects. The greater the depth of a tree is, the more inner nodes the tree will have. The inner nodes will incur significant memory cost and may have a severe influence on the performance of a tree. This is the main reason why the quadtree, and octree cannot be effectively used as the spatio-temporal access methods.

To resolve these problems, we propose a novel spatio-temporal indexing method, called Decomposition Tree (D-tree). The D-tree is a virtual tree without inner nodes, instead, through an encoding method based on integer bit-shifting operation. Since data is only stored in leaves, traversing the tree nodes avoids disk access, and consequently the depth of the tree does not affect the query performance. Instead of employing various kinds of geometric representation (for example, MBR or TPMBR), data transformation (for example, SFC) or component division (just like STCB⁺-tree), the actual time and spatial locations of moving objects are treated directly. The D-tree can efficiently answer a wide range of queries from the point query to the moving query. When the D-tree is disk-resident, a cached buffer in memory is used for temporarily caching some insertion and deletion results. This updating policy is similar to LUGrid [12]. The empirical

performance of D-tree is demonstrated to be superior to the STCB⁺-tree [18, 19] which is in turn reported to outperform the TPR-tree [7].

The remainder of this paper is organized as follows: in the following section we review the related work on spatio-temporal indexes for movement data. In Sect. 3, we describe the virtual structure of the D-tree. In Sect. 4, we present the algorithms of the D-tree. Section 5 contains the experimental results and analysis. Finally, Sect. 6 draws our conclusions.

2 Related work

A lot of spatio-temporal access methods for movement data have been proposed in the last decades. This kind of main index methods are listed in Table 1. They can be classified into four groups: methods for indexing the past data, the present data, the future data or the data at all points in time, according to the temporal queries which are supported [22]. Most of them are derived from B-trees [20] or R-trees [21]. According to the classification criteria of partition ways, they can be divided into two categories: data partition and space partition. R-tree [21] and R*-tree [23] were proposed only for spatial data. In theory, they can also be used in spatio-temporal data due to their general representation way for spatial objects, which is that they employ a MBR to represent a spatial object. However they are not efficient for indexing continuous moving object data because of their low updating efficiency and the serious overlapping problem in high dimension space.

The TPR-tree [7] is a famous variant of the R*-tree supporting the queries for present and future positions of moving objects. In a TPR-tree, moving objects are enclosed by conservative bounding rectangles which never shrink. It uses a linear function of time to preserve the related position in space and associated period in time. The function parameters employed to describe every motion are the intercept position at the current time and a time-parameterized velocity vector. To achieve a better data classification, the TPR*-tree [24] improves the TPR-tree by employing a new set of insertion and deletion algorithms. It aims at minimizing the (conservative) bounding rectangle for reducing query cost. However the choice of appropriate parameter values for this improvement is sensitive; different queries need different appropriate parameters. The limitation of both types of trees is that they cannot handle historical queries because of their security policies based on the current and future positions of moving objects.

This limitation was resolved in the RPPF-tree [13] which indexes positions of moving objects at all points in time. The past positions of an object between two consecutive samples are linearly interpolated and the future positions are com-

Table 1 Main spatio-temporal access methods for movement data

Time	Type			
	Index for past data	Index for present data	Index for future data	Index for the data at all points in time
1990	RT-tree, MR-tree			
1996	3D R-tree			
1998	HR-tree		PMR-Quadtree	
1999		2 + 3 R-tree	Duality transformation	
2000	STR-tree		TPR-tree	
2001	HR ⁺ -tree, MV3R-tree Greedy-PPR-tree	Hashing	SV-Model PSI	
2002	R ST -tree	2 – 3TR-tree LUR-tree	MB-index	
2003	SETI SEB-tree FNR-tree	IMORS Bottom-up approach Q + R-tree	FT-Quadtree TPR*-tree	
2004	Chebyshev Polynomial Index	COVET	STRIPES B ^x -tree STP-tree	
2005	MTSB-tree MON-tree PA-tree SEST-Index	LGU		BB ^x -index PCFI ⁺ -index
2006		LUGrid		R ^{PPF} -tree
2007			ANR-tree	
2008	GStree CSE-tree		B ^y -tree ST ² B-tree B ^{dual} -tree B ^{dH} -tree	UTR-tree PPF1
2009	Polar tree RTR-tree TP ² R-tree	RUM-tree	MOVIES	STCB ⁺ -tree
2010			B ^s -tree	
2011			MOVIES	
2012	DR-tree			STCB ⁺ -tree
2013		RUM ⁺ -tree		

puted via a linear function using the most recent positions. The RPPF-tree applies partial persistence to the TPR-tree to capture the past positions. Leaf and non-leaf entries of the RPPF-tree include a time interval recording the insertion time and deletion time. When a node is split at time t , moving objects in this node alive at time t are copied to a new node which time interval is from time t to the future. This means that the deletion time of the new node is unidentified. A time-parameterized bounding rectangle of a TPR-tree is valid from the current time, while the time-parameterized bounding rectangle of an RPPF-tree is valid from the insertion time.

The RTR-tree and TP²R-tree [16] are R-tree based structures specially for indexing trajectories in symbolic indoor space. They are not general spatio-temporal index for movement data. TPR-tree, TPR*-tree, RPPF-tree, RTR-tree and TP²R-tree are all R-tree-like indexes and can be classified into the category of data partition.

Another category is of indexes relies upon spatial partitioning methods. An important set of these are B⁺-tree based indexes, such as B^{dual}-tree [14], B^y-tree [15], B^s-tree [17], and STCB⁺-tree [18, 19]. The B^{dual}-tree is an extension of the

B^x-tree [11] which consists of multiple B⁺-trees indexing the one-dimensional values transformed from moving objects based on a SFC. Because this space filling curve does not consider object velocities, query processing with a B^x-tree may retrieve many false hits. Unlike the B^x-tree, the B^{dual}-tree captures both d-dimensional locations and velocities in a dual two-dimensional space. The dual space is partitioned along all dimensions into cells. Then an SFC transforms the two-dimensional values in the dual space into one-dimensional values that are indexed in a B⁺-tree. Each cell in the partition space can be regarded as a d-dimensional moving rectangle that captures the locations and velocities of all objects inside it similar to the TPMBR in the TPR-tree. The query algorithm of B^{dual}-tree is similar to TPR-tree, but the insertion and deletion algorithm of B^{dual}-tree is similar to those of the B^x-tree. B^y-tree is also an extension of B^x-tree. It uses separate update frequencies for each moving object so that it works well in the environments with high variable update periods. B^s-tree is a self-tuning index which can balance the query and update performances for optimal overall performance in movement databases.

The spatio-temporal compressed B^+ -tree or STCB $^+$ -tree [18, 19] uses multiple compressed B^+ -trees or CB $^+$ -trees [25] to index trajectories of moving objects. One CB $^+$ -tree, the TCB $^+$ -tree, indexes the temporal dimension and for each spatial dimension, another CB $^+$ -tree, the SCB $^+$ -tree, indexes the spatial coordinates of moving objects in that dimension. The insertion and deletion in the CB $^+$ -trees are same as those in B^+ -trees except for the data shifting. In the TCB $^+$ -tree, each interval identified by two consecutive key values stores a bucket including the identifiers of the moving objects in that time interval. Because the generation of new temporal data is totally ordered, the insertions into TCB $^+$ -tree append new entries to the rightmost leaf node. Similar to the TCB $^+$ -tree, the SCB $^+$ -tree in each dimension keeps a bucket for each spatial interval identified by two consecutive key values in that dimension. This bucket stores the identifiers of the moving objects in this spatial interval. To answer a spatio-temporal query, the TCB $^+$ -tree and the SCB $^+$ -trees in each spatial dimension are searched for the temporal and spatial outputs. The final result is the intersection of those outputs. It is reported that both analytical and empirical studies show that the STCB $^+$ -tree [19] outperforms the TPR-tree. This is one of the reasons why we choose it for comparison to D-tree proposed in this paper.

Except for the B^+ -tree based indexes, some grid based indexes, such as LUGrid [12], GPR-tree [26] and Grid-Quadtree [27], also rely on space partition. The LUGrid is a spatio-temporal access method that answers the queries on the current status of moving objects. It adapts to arbitrary object distributions via the adaptive grid structure derived from the grid file [28]. It applies lazy insertion and deletion to handle frequent updates to the locations of continuously moving objects. Lazy insertion can reduce the update I/Os by adding an additional memory-resident layer over the disk index. The incoming updates are stored in the memory and then are lazily flushed into disk in batch. Thus, multiple updates are reduced to only a single disk update. The lazy deletion can reduce updates by avoiding deleting single absolute entry out of the index immediately. The GPR-tree is a hybrid of grid partitioning and R-tree for indexing moving objects on fixed networks. These work by dividing the network space into grids of different size and indexing trajectories in each grid. The Grid-Quadtree is a cyclic-translation-based index that queries location translations.

Like the grid based indexes, the quadtree, octree and hex tree are also space partition access methods, which are hybrids of grid and tree. As mentioned in Sect. 1, these trees may be unbalanced due to the non-uniform distributions of spatio-temporal data. In addition, most of the movement data are updated continuously. The depth of the tree uncontrolled by algorithm may seriously reduce the performance of the tree. This means the quadtree, octree and hex tree cannot be

effectively used as spatio-temporal access methods. It is our intention to propose the decomposition tree (D-tree) to integrated binary tree, quad tree, octree, hex tree to resolve this problem via a virtual structure.

3 Decomposition and index structure of D-tree

The D-tree is a space partitioning access method for movement data. Movement data are most commonly represented as a collection of spatial points with time stored as an attribute [2]. A more formal definition of movement data as adopted in this paper is a collection $\{Mt\}$ of $t = 0, \dots, n$ ordered records, each comprising the tuple $\langle ID, T, S, A \rangle$, where ID is a unique object identifier, T is a non-duplicated sequential time stamp, S are spatial coordinates, and A are the attributes of Mt. The basic idea is to manage this kind of moving objects by a virtual index structure that relies on a multiple binary space partition method and an encoding method. Given that a space is d -dimensional; our region of interest is a d -dimensional rectangle. Our main proposal is to recursively decompose the basic rectangle into $2d$ smaller sub-rectangles. So according to this, a d -dimensional decomposition tree (D-tree) will divide the basic rectangle into $2d$ sub-rectangles in which each sub-rectangle has the same volume. If we give a unique integer to represent each sub-rectangle, then we need an integer which is composed of d binary bits at least. Assume that the integer code of the basic rectangle is 1, and if d is equal to 1, then the two sub-rectangles codes are two binary integers 10 and 11, if d is equal to 2, the four sub-rectangles codes are four binary integers 100, 101, 110, and 111, and if d is equal to 4, there will be 16 sub-rectangles, and their codes are 16 binary integers from 10000 to 11111, or from 0x10 to 0x1f in base 16. Each sub-rectangle can also be divided by this criterion, for instance, the sub-rectangle 0x10 can be divided into 16 smaller sub-rectangles whose integer codes are from 0x100 to 0x10f in base 16. If the partition goes on, a tree will be constructed and each node of the tree will have a unique integer code. Therefore we employ a unique integer code on representing a node and can easily calculate the range of the node via this integer code.

For example, if the range of the one-dimensional basic rectangle is $[0, 256]$ and there are 7 one-dimensional points, a, b, c, d, e, f, g distributing in the range $[0, 256]$ as shown in Fig. 1, according to the above-mentioned decomposition criterion, a binary tree will be formed, shown in Fig. 2. We can store this tree as a table, shown in Table 2. Because the range of each node can be calculated from the integer code in real time, there is no necessity to store it in memory or disk.

According to the decomposition criterion, it is easy to extend the D-tree from one to four or higher dimensions.

Fig. 1 An example of one-dimensional decomposition

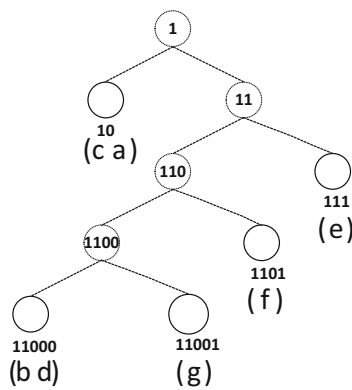
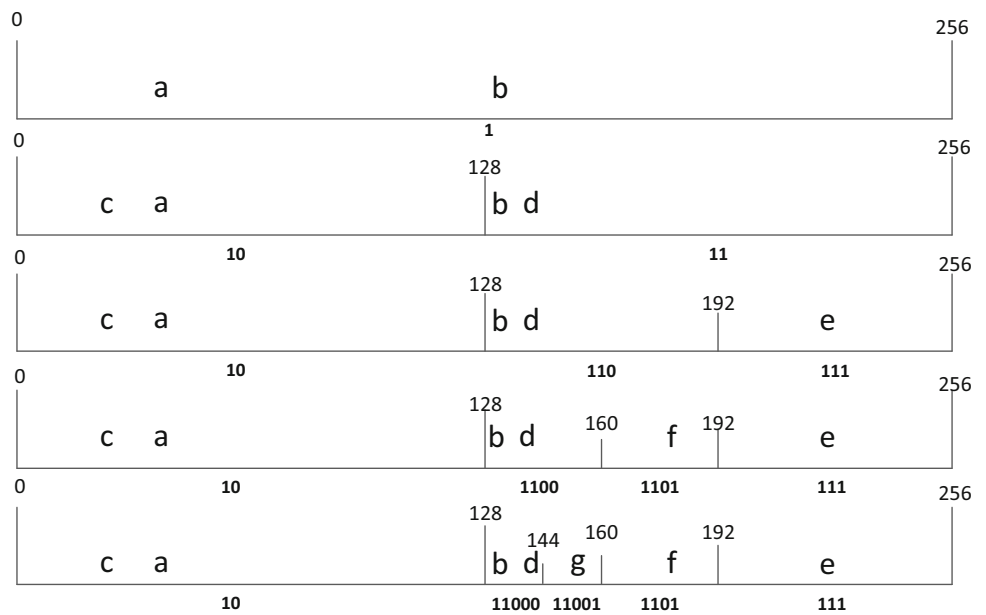


Fig. 2 An example of one-dimensional D-tree

Table 2 An example of one-dimensional D-tree

Integer codes of leaves		Range	Moving points
Binary format	Decimal format		
10	2	[0,128]	a, c
111	7	[192,256]	e
1101	13	[160,192]	f
11000	24	[128,144]	b, d
11001	25	[144,160]	g

When the dimensions is four, the D-tree is a virtual hex tree with index records in its leaf nodes containing an integer code and some pointers referring to moving objects and, hence may be unbalanced like the virtual tree shown in Fig. 3. The D-tree can be disk-resident or in-memory. If the D-tree is

disk-resident, each leaf node corresponds to a disk page; otherwise it will correspond to an array of pointers referring to moving objects in memory. Therefore the D-tree cannot be only an in-memory tree, but also a disk-resident tree. The structure is designed so that a spatio-temporal query only requires visiting the unique integer code of each leaf. The index is dynamic, and insertion and deletion can be inter-mixed. Each leaf is in the form

(LEAF-CODE, ENTRY-ARRAY)

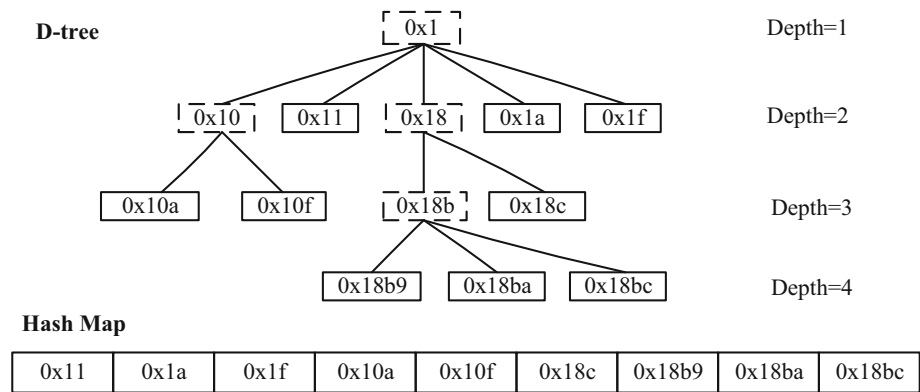
Where LEAF-CODE is an integer number representing the leaf node, ENTRY-ARRAY is array of moving objects' entries which are contained in the rectangle represented by the LEAF-CODE. Each entry in the ENTRY-ARRAY represents a moving point, and it is in the form as following:

(TRJID, PNT-POINTER)

Where TRJID is an identifier of a trajectory which contains the moving point, PNT-POINTER is a pointer referring to the moving point or an address from which the moving point begins to be stored when D-tree is a disk-resident tree. In order to find entry fast, the leaf nodes are contained in a hash map whose key is the LEAF-CODE, a unique integer.

An example of a four-dimensional tree is shown in Fig. 3, the root node 0x1 are divided into 16 sub-rectangles identified from 0x10 to 0x1f. If there are no data in those sub-rectangles except 0x10, 0x11, 0x18, 0x1a and 0x1f, we just need to mark these five sub-rectangles. Because there are too many spatio-temporal objects contained in nodes 0x10 and 0x18, these two nodes should be divided again so that the depth of the tree increases and these two nodes become two virtual nodes which should not be stored in the hash map. If there are still too many spatio-temporal objects in 0x18b, it should

Fig. 3 An example of four-dimensional D-tree



be divided, and do on until each leaf node contains a suitable number of spatio-temporal objects. The tree is virtual and do not exist in memory. The hierarchical relationships of the tree are implicated by the integer code of each leaf node. The only existing is the hash map containing leaf nodes. As we know that the node number of a full balanced hex tree whose depth is 4 is equal to 4369. However the example we present only has 9 nodes stored in the memory. That is one of the reasons why the D-tree may be efficient. The above description presents the basic data structure of the D-tree. We will present some key algorithms of D-tree in next section.

4 Algorithms of D-tree

Assume that the dimension of the basic rectangle is $_dimensions$ (because our test data set in this paper is four-dimensional, the default value is 4), and the basic rectangle is $_basicrectangle$. The maximum number of spatio-temporal objects contained by each leaf is $_maxobjects$. The current depth of the tree $_depth$, and the hash map containing leaf nodes is $_hashmap$. In addition, the current time of the D-tree is $_currenttime$ and the current entries of D-tree are $_currententries$. These two are for supporting predictive query. They are the global variables of the following algorithms we will discuss.

Because the D-tree is a virtual tree and its hierarchical relationships are implicated by the integer codes, below we describe some initial algorithms for the integer code and decomposition method. If the integer code of the current node is $_code$, its parent node's integer code should be $_code \gg _dimensions$ (shift right $_dimensions$ bits). If $_prefix$ and $_code$ are two integers codes, and $_prefix$ is equal to $_code \gg x$ (shift right x bits, x is a multiple of $_dimensions$), then the integer code $_prefix$ is one of the prefixes of $_code$. This algorithm is for how to judge whether $_prefix$ is one of the prefixes of $_codes$.

Algorithm 1: judge whether $_prefix$ is one of prefixes of $_code$

Input: $_prefix$: Integer, $_code$: Integer

Output: $_result$: Boolean

```

{
  if ( $\_prefix$  is not less than  $\_code$ ) return False;
  while ( $\_code$  is not equal to zero)
  {
    shift  $\_code$  right  $\_dimensions$  bits;
    if ( $\_code$  is equal to  $\_prefix$ ) return True;
  }
  return False;
}

```

The decomposition strategy is the basic idea of this virtual structure. If the current rectangle is $_rectangle$, it may be divided into $2_dimensions$ sub-rectangles which index is from 0 to $2_dimensions-1$. The decomposition strategy makes sure that we can get anyone of the sub-rectangles of the current rectangle according to an index value whose range is between 0 and $2_dimensions-1$.

Algorithm 2: decompose the rectangle $_rectangle$

Input: $_rectangle$: Rectangle, $_index$: Integer [0, $2_dimensions-1$]

Output: $_result$: Rectangle

```

{
  for ( $i$  is an integer from 0 to  $\_dimensions-1$ )
  {
    divide the range on the  $i$ th axis of  $\_rectangle$  into two parts,  $\_lowrange$  and  $\_highrange$ ;
    if ( $i$ th bit of  $\_index$  is equal to 1)
      the range of  $\_result$  on the  $i$ th axis equals  $\_highrange$ ;
    else
      the range of  $\_result$  on the  $i$ th axis equals  $\_lowrange$ ;
  }
}

```

The key of D-tree is that an integer code can represent a rectangle or a node of the D-tree. Hence we need algorithms for setting up the relationship between these two. Algorithm 3 shows how to calculate a rectangle according to an integer code. And Algorithm 4 is the inverse operation of Algorithm 3, which calculates the integer code of a node to which a rectangle or a point belongs at the given depth of the tree. This algorithm can handle two types of input data, rectangles and points.

Algorithm 3: calculate rectangle according to an integer code

Input: *_code*: Integer
Output: *_result*: Rectangle

```

{
  if ( _code is equal to 1) _result equals the basic rectangle and return;
  let _stack be a stack;
  let _oldcode be _code;
  while ( _code is equal to zero)
  {
    push _code - (_code>>_dimensions)<<_dimensions into _stack;
    let _code be _code>>_dimensions;
  }
  restore _oldcode to _code;
  pop the top number of the stack, let _index be the top number, use Algorithm 2 to
  decompose the basic rectangle recursively until the stack is empty;
  let _result be the last sub-rectangle and return;
}

```

Algorithm 4: calculate the integer code of a node to which a rectangle or a point belongs at the maximum depth of the tree

Input: *_rect*: Rectangle, *_depth*: Integer
Output: *_code*: Integer

```

{
  let _temprectangle be _basicrectangle;
  let _code be 1;
  let i be 1;
  while ( i is less than _depth)
  {
    for ( j is an index from 0 to 2 _dimensions-1)
    {
      call Algorithm 2 to decompose _temprectangle and get the jth sub-rectangle,
      if the sub-rectangle encloses _rect, let _temprectangle be the sub-rectangle,
      shift _code left _dimensions bits and plus j;
    }
    i increase 1;
  }
}

```

4.1 Insertion algorithm

Inserting a moving point into a D-tree is a procedure according to Algorithm 5. We first descend the tree from the root to the virtual leaf at maximum depth containing the moving point, and then retrieve the integer code of the virtual leaf. This is done arithmetically by Algorithm 4 without disk access. The leaf found is called ‘virtual’ because that branch of the tree may have depth less than the maximum depth, or it may not exist in the hash map. According to the decomposition criteria, the moving point for insertion must belong to one of the nodes whose integer code is in the collection containing the integer code of the virtual leaf and all the prefixes of the integer code. If the hash map contains anyone of the integer codes in the collection, we find the leaf node represented by this integer code and push the moving point into the leaf. If not, we add a leaf node to the hash map. The integer code of this leaf node is the prefix with longest digits of the virtual leaf code. At last we push the moving point for insertion into this leaf. The Algorithm 5 is the insertion algorithm. In this algorithm, when the D-tree is a disk-resident index, all the insertions needing no splitting will be handled in memory via a cached buffer, and the insertions will store to disk only when the cached buffer is overflowing or there are one or more node splits.

Algorithm 5: insert

Input: *_object*: Moving point; *_trjid*: The ID of a Trajectory
Output:

```

{
  let _mbr be the minimum bounding rectangle of _object;
  call Algorithm 4, get the integer code of a node which _mbr belongs to at the
  maximum depth of the tree, and let _code and _old_code be the code;
  while ( _code is not zero)
  {
    find the code _code in the hash map _hashmap;
    if ( find a node which integer code is _code)
    {
      judge whether the number of spatio-temporal objects contained by this
      node is less than _maxobjects, if it true, push _object into this node, else
      push _object into this node and split this node;
    }
    else
      shift _code right _dimensions bits;
  }
  let _code be _old_code;
  calculate the depth of node represented by _code, let _tempdepth be the depth
  value;
  add a leaf node which integer code is the prefix with longest digits of _code, and
  push _object into this node;
  if _currenttime is older than the time of this moving point, let _currenttime be the
  time of this moving point;
  if the time of _object is newer than the time of the second last point of the trajectory
  _trjid, replace the last two entry pointers of the trajectory _trjid in _currententries;
}

```

Algorithm 6: split node

Input: *_node*: Leaf node of D-tree
Output:

```

{
  let _mbr be the minimum bounding rectangle of _node;
  let _code be the integer code of _node;
  call Algorithm 2, decompose _mbr into 2 _dimensions sub-rectangles;
  let i be an integer from 0 to 2 _dimensions-1;
  traverse all the spatio-temporal objects in the nodes, if an object is enclosed by ith
  sub-rectangle, then push it into the node which integer code is equal to
  _code<<( _dimensions+i), and push this node into the hash map _hashmap;
  if all of the objects are been pushed into a same sub-rectangle, call split algorithm
  recursively;
}

```

In order to support the efficient predictive query for the future data, we need to store some additional information in the process of insertion. The first is the current time of the D-tree, *_currenttime*. When *_currenttime* is older than the time of the moving point for insertion, we let *_currenttime* be the time of the moving point for insertion. The second is the last two moving point of each trajectory. We use two entry pointers referring to them. In the predictive query, we use these two to calculate the current velocity and direction of the last point in a trajectory. Following this, we can calculate some future positions of this trajectory.

When the number of the moving points contained by a leaf is larger than the maximum number permitted, we should call the split algorithm to split this leaf. At first, we decompose the rectangle of the node for splitting, test all the moving points contained in the node which sub-rectangle covers them and push them into each sub nodes. There is an exception that all the moving points are pushed into a same sub node. In this case, we need to recall the split algorithm recursively. Algorithm 6 presents the splitting algorithm.

These two algorithms build up the insertion operation. For some other spatio-temporal indexes, such as TPR-tree and STCB⁺-tree, the data inserted should be ordered by time. While for the D-tree, the data inserted are not necessary to be ordered.

4.2 Query algorithm

There are several query types for moving objects. In this paper, they are classified into five types summarized from the literature reviewed here [7, 19, 22].

Point query: $Q = (t, p)$ specifies a spatial position p at a particular time t . The query retrieves all the trajectories are near to the spatial position p at the time t .

Timeslice query: $Q = (t, r)$ specifies a multi-dimensional spatial rectangle r at a particular time t . The query retrieves all the trajectories that are overlapped by the spatial rectangle r at the time t .

Spatialslice query: $Q = (ts, te, p)$ specifies a spatial position p during a particular time interval $[ts, te]$. The query retrieves all the trajectories that are near to the spatial position p during a particular time interval $[ts, te]$.

Window query: $Q = (ts, te, r)$ specifies a multi-dimensional spatial rectangle r that is valid during a particular time interval $[ts, te]$. The query retrieves all the trajectories that are overlapped by the spatial rectangle r during a particular time interval $[ts, te]$.

Moving query: $Q = (ts, rs, te, re)$ specifies the multi-dimensional trapezoid obtained by connecting the multi-dimensional spatial rectangle rs at time ts to the multi-dimensional spatial rectangle re at the time te . The query retrieves all the trajectories are overlapped by the trapezoid.

According to the time arguments provided, each of the above queries can be classified into the queries for past, present, and future data, or all data points. When the time argument provided is newer than the current time of the D-tree, the query is a predictive query. While the time argument inputted is older than the current time of the D-tree, the query is a historical query. So there are 10 types of queries in total. The D-tree is designed for indexing the data at all points in time.

According to the classification mentioned above, from Point Query to Moving Query, the constraints on the queries become less than their previous types. The Window Query generalizes the Point Query, Timeslice Query and Spatial-slice Query, and is itself a special case of the Moving Query. Because multi-dimensional trapezoid is not convenient for overlapping computing, we let $r = rs + re$ and apply a Window Query to approximately simulate the Moving Query in this paper. So we only discuss the Window Query algorithm of D-tree here.

The basic idea of the query algorithm is replacing rectangle intersection computation with the integer shifting computation on the integer codes of leaves because each integer code is corresponding to each rectangle covered by a virtual node or a leaf. If the input parameter is not a rectangle but a point, they are similar due to the Algorithm 4. At first, we call Algorithm 4 to calculate the integer code of a node to which a rectangle belongs at the maximum depth of the tree. Then we add three types of leaf nodes to a list. The first type is composed of all the leaf nodes whose integer codes are the prefixes of the integer code. The second is the leaf node whose integer code is equal to the integer code. The last is composed of all the leaf nodes in which the integer code is always a prefix of their integer codes. Adding these three types of leaf nodes to a list changes the tree structure to a linear structure without relationships between every two elements of the list. It is the reason why the query can be a parallel algorithm easily. At last, we parallel traverse all leaf nodes in the list and compare each leaf's integer code to the integer code of the node to which the query rectangle belongs, if the two integer codes are equal or the former is a prefix of the latter, add all the moving points of the leaf which overlap the query rectangle to the result, else we calculate the rectangle of the leaf node and add each moving point of the leaf node which overlaps the query rectangle to the result. The query algorithm is shown in Algorithm 7.

Algorithm 7: window query

Input: `_queryrectangle`: Rectangle

Output: `_result`: Moving points sorted by their TRJID and times.

```

{
  call Algorithm 4 to calculate the integer code of a node which _queryrectangle
  belongs to at the maximum depth of the tree, and let _code be this integer code;
  add all leaf nodes in the _hashmap whose integer codes are prefixes of _code to a
  list _list;
  if the leaf node whose integer code is _code exists in _hashmap, add it to _list;
  add all leaf nodes in the _hashmap whose integer codes have a prefix which is equal
  to _code to the _list;
  parallel traverse all leaves in _list, assume the current leaf node is _node;
  {
    let _nodecode be the integer code of _node;
    if (_nodecode is equal to _code or is a prefix of _code)
    {
      traverse all moving points in this node, each moving point which overlaps
      _queryrectangle should be added to _result;
      sort the _result by TRJID and time;
      return;
    }
  }
  if (_code is a prefix of _nodecode)
  {
    call Algorithm 3, let _rect be the rectangle of _node;
    if (_queryrectangle encloses _rect)
    {
      add all the objects of this leaf node to _result;
    }
    else if (_queryrectangle overlaps _rect)
    {
      traverse all the objects in this node, if the rectangle of an object overlaps
      _queryrectangle, push this object into _result;
    }
  }
}
sort the _result by TRJID and time;
}

```


Algorithm 7 is designed for querying the past data and the present data. When time arguments is newer than the current time of the D-tree, the entries of the last two points of all the trajectories are found in `_currententries`, and then judge whether the future points will appear in the specific future time interval and be overlapped by the rectangle argument input. Because the number of trajectories is always much less than the number of points on the trajectories, we employ a small buffer, `_currententries`, to store all the pointers of those entries of all the last two points can improve the performance of predicative queries.

4.3 Deletion algorithm

A D-tree is a dynamic index structure, so it supports deleting anyone of moving objects in the D-tree. At first, we calculate the integer code of a node to which the moving object for deletion belongs at the maximum depth of the tree. An integer code collection is made up of this integer code and all of its prefixes. The integer code of the leaf node containing the moving object for deletion must be in the collection. Algorithm 8 is for the deletion.

```

Algorithm 8: delete
Input:  _object: Moving Object for Deletion
Output:
{
  let _mbr be the minimum bounding rectangle of _object.
  call Algorithm 4, get the integer code of a node to which _mbr belongs at the
  maximum depth of the tree, and let _code be the code;
  find _code in the hash map _hashmap;
  while ( _code does not exist in _hashmap)
  {
    shift _code right _dimensions bits;
    find _code in the hash map _hashmap;
  }
  get the node which integer code is _code, and remove _object in this node;
  if the node is empty, remove it from _hashmap;
  if _object is in _currententries, remove it from _currententries;
}

```

4.4 Extend algorithm

The insertion algorithm mentioned in 3.1 has a hypothesis that all the moving objects for insertion are contained in the basic rectangle. As time moves on, it is possible that the moving object for insertion may be out of the range of the basic rectangle. In this case, we should extend the basic rectangle of the virtual tree. Because there are not real rectangles storing in the virtual tree, the only thing we need to do is to change the integer code of each leaf node. Algorithm 9 shows the process of the extending operation.

```

Algorithm 9: extend
Input:  _object: Moving Object for Insertion
Output:
{
  calculate the minimum parent rectangle of the virtual root node, in which the
  minimum bounding rectangle of _object is contained;
  let the _basicrectangle be the minimum parent rectangle;
  calculate the new integer code of the old virtual root node in the new virtual tree;
  change the first digital '1' of the integer code of all the leaves into the new integer
  code of the old virtual root node;
}

```

5 Experimental evaluation

In order to evaluate the D-tree, the four-dimensional moving point data are extracted for studies, and the spatio-temporal index structures STCB⁺-tree [19], which is superior to the TPR-tree [7] in both space cost and query performance, is chosen for comparison studies.

5.1 Space cost evaluation

The following parameters are used in our space cost evaluation.

- M: memory size;
- S: the total number of the moving objects;
- N: the number of the non-leaf nodes in a tree index;
- L: the number of the leaf nodes in a tree index;
- C: the maximum capacity of each node;
- D: the depth of the tree;
- d: the size of a double float number;
- p: the size of a pointer;
- i: the size of an integer;
- K: the dimension of movement data

To preserve the four-dimensional moving point data, the STCB⁺-tree requires four CB⁺-trees, one TCB⁺-tree and three SCB⁺-trees. Every entry preserved in the nodes of a CB⁺-tree includes a pointer referring to a sub-tree and a key whose type is double float. Except that, each leaf in a CB⁺-tree needs two links to its previous and next sibling leaf nodes. Thus, the total space cost can be calculated by the following formula:

$$M = N * (d + p) * C + L * (d + 3 * p) * C + 4 * S * (d + p) \quad (1)$$

Different to the STCB⁺-tree, the D-tree proposed in this paper has no inner node and all the nodes in the tree are leaf nodes. Therefore the parameter N is equal to zero. Each entry of a leaf node stores a pointer referring to a moving point and an integer identifying a trajectory. Each leaf node is composed of an integer code and some entry pointers for

which the maximum number is C . Because the bounding rectangle of each leaf can be calculated by the integer code, it is not necessary for the leaf node. Thus, the total space cost can be calculated by the following formula:

$$M = L * (i + p * C) + S * (p + i) \tag{2}$$

In the C/C++ language, the size of an integer is often equal to the size of a pointer, so the formula (2) can be written as following:

$$M = L * p * (C + 1) + 2 * S * p \tag{3}$$

If the moving objects are not moving point data or trajectories but complex spatio-temporal objects, such as moving regions or moving solids, the D-tree will employ a bounding rectangle to represent the objects. In that case, we call the D-tree for the Complex D-tree (CD-tree). The total space cost can be calculated by the following formula:

$$M = L * p * (C + 1) + S * (p + 8 * d) \tag{4}$$

If we let the minimum object number in a node be $M/2$, because each CB^+ -tree of the $STCB^+$ -tree is a balanced tree and there are K CB^+ -trees, L is at least $\lfloor \frac{S * K}{C} \rfloor$ and we can approximately evaluate the parameters D and N via the following formulae:

$$\lceil \log_{M/2} S \rceil - 1 \geq D \geq \lfloor \log_M S \rfloor - 1 \tag{5}$$

$$N \approx \sum_{j=0}^D C^j \tag{6}$$

To evaluate the space utilization of $STCB^+$ -tree and D-tree in the experiment, we count the node number in each index and then use these formulae to calculate the space cost of each index. A test environment was established using a computer with the configuration of a 2.53 GHZ Intel(R) Core(TM) Duo CPU, 2 GB RAM. The test data is a simulated four-dimensional dataset. The simulated three-dimensional spatial range is from $[0, 0, 0]$ to $[10000, 10000, 10000]$, and the velocity range is from 0 to 1 m/s in each axis. We randomly generated from 100,000 to 1,000,000 objects and tested the space utilization and query performance of these three index methods.

In order to keep the performance comparison manageable, the maximum number of the objects permitted in one node (the parameter C), is 128, and the minimum is 64. The space cost comparison results among D-tree, CD-tree, and $STCB^+$ -tree are shown in Fig. 4. Because the D-tree is a virtual tree without inner nodes, it means that the space cost of the D-tree is not related to the parameter N and the parameter D . Contrary to a virtual D-tree, the larger the number of the

moving objects is, the greater the depth of a real tree is. According to formula (6), the number of the inner nodes of a real tree, the parameter N , will increase dramatically with the growth of the parameter D , the depth of the tree. That is why the D-tree economizes on space utilization comparing to $STCB^+$ -tree.

5.2 Query performance evaluation

Because the window query can stimulate the point query, time slice query, spatial slice query, and the moving query as mentioned in Sect. 4.2, we mainly study and the window query. The experimental data is the test dataset mentioned in Sect. 5.1.

The whole range of the test data is a four-dimensional rectangle, which we called it the basic rectangle BR. The query rectangle QR is the input parameter of a window query. The Fig. 5 shows the relationship between query time cost and the size of query rectangle. The axis X represents the ratio of each side length of QR and BR in each dimension. The

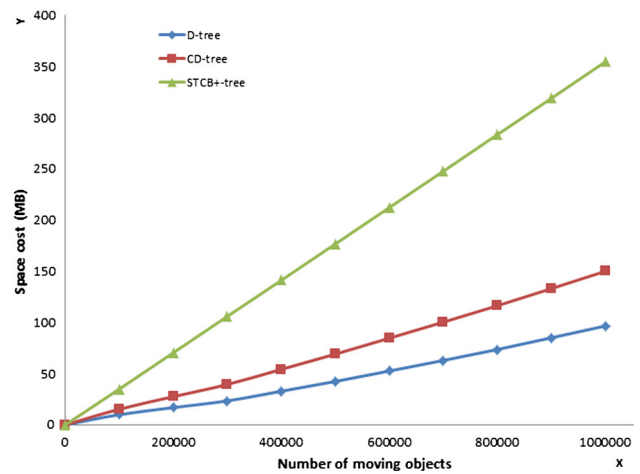


Fig. 4 Space cost comparison

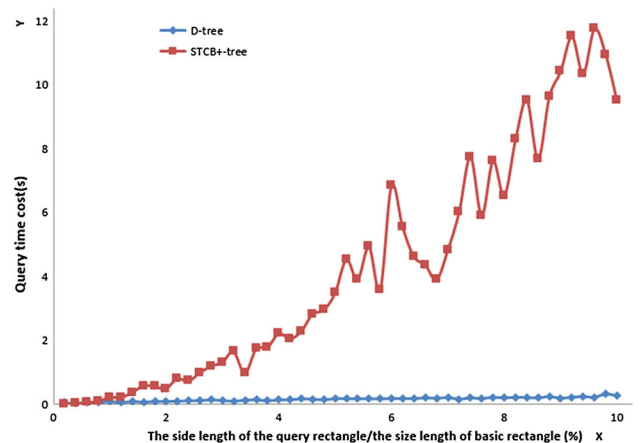


Fig. 5 Query performance comparison over query range

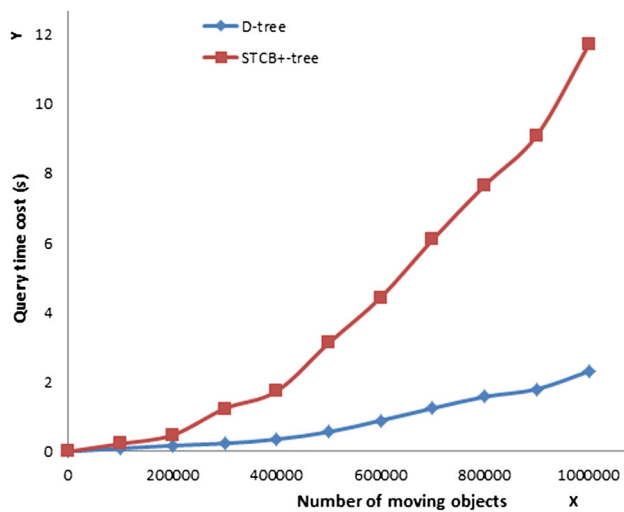


Fig. 6 Query performance comparison over data size

ratio, noted as QR/BR, is from 0 to 10 % and its step is 0.2 %. At each point on the axis X, there are 100 query rectangles (QRs) which distribute randomly in the basic rectangle (BR). Each point on the axis Y represents the total query time cost of the 100 queries based on the related 100 query rectangles. The results in Fig. 5 show that the window query performance of the D-tree exceeds the query performance of the STCB⁺-tree. Generally, the query time cost will increase while the range of query rectangle becomes larger. This trend on the STCB⁺-tree is very obvious, however this trend on D-tree is not obvious. It implicates that D-tree adapts to not only the small range queries but also the large range queries. The Fig. 6 shows the relationship between the average query time cost and the data size. The axis X represents the number of the moving points and the axis Y represents the average query time cost. It is obvious that average query performance of D-tree is superior to STCB⁺-tree's while the data size or the number of moving objects becomes larger and larger.

The Figs. 5 and 6 show that the D-tree is not much sensitive to the increment of the data size and the range of query rectangle comparing to the STCB⁺-tree. All the test results demonstrate that the D-tree is superior to the STCB⁺-tree not only in the space cost but also in the query performance.

6 Conclusion

Although computer memory capacity continues to grow and become cheaper, the actual main memory of a computer for managing increasingly large movement data sets is still limited. Indexing strategies for reducing and/or optimizing memory utilization for spatio-temporal data are therefore highly relevant. We presented a new access approach, named the Decomposition Tree (D-tree), for handling multi-dimensional movement data. Comparing with STCB⁺-tree,

the smaller memory utilization and more efficient query performance of the D-tree are the major contributions of our study. Because D-tree is a discreet linear structure, it is easy to be implemented in parallel. Therefore, it is suitable for using in cluster computing for movement big data access. Further research will include optimizing the D-tree for complex moving objects, called Complex Decomposition Tree (CD-tree), as well as the integration of the D-tree for trajectory and the CD-tree.

Acknowledgments The work described in this paper was supported by National Natural Science Foundation of China (41101368, 41172300) and National High Technology Research and Development Program of China (2012AA121401).

References

- Guo, H., et al.: Scientific big data and digital Earth. *Chin. Sci. Bull.* **59**(35), 5066–5073 (2014)
- Long, J.A., Nelson, T.A.: A review of quantitative methods for movement data. *Int. J. Geogr. Inf. Sci.* **45**, 1–27 (2012)
- Wang, L., et al.: Cloud computing: a perspective study. *New Gener. Comput.* **28**(2), 137–146 (2010)
- Wang, L., et al.: Towards enabling cyberinfrastructure as a service in clouds. *Comput. Electr. Eng.* **39**(1), 3–14 (2013)
- Chen, D., et al.: Fast and scalable multi-way analysis of massive neural data. *IEEE Trans. Comput.* **64**(3), 707–719 (2015)
- Wang, L., et al.: Particle swarm optimization based dictionary learning for remote sensing big data. *Knowl. Based Syst.* **79**, 43–50 (2015)
- Saltenis, S., et al.: Indexing the positions of continuously moving objects. *SIGMOD Rec.* **29**(2), 331–342 (2000)
- Lin, B., Su, J.W.: On bulk loading TPR-tree. In: 2004 IEEE International Conference on Mobile Data Management. pp. 114–124 (2004)
- Tao, Y., et al.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In: Proceedings 2003 VLDB Conference, pp. 790–801. Morgan Kaufmann, San Francisco (2003)
- Kim, S.W., Jang, M.H., Lim, S.: Active adjustment: an effective method for keeping the TPR*-tree compact. *J. Inf. Sci. Eng.* **26**(5), 1583–1600 (2010)
- Christian, S.J., Dan, L., Beng Chin, O.: Query and update efficient B⁺-tree based indexing of moving objects. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30. VLDB Endowment, Toronto (2004)
- Xiaopeng, X., Mohamed, F.M., Walid, G.A.: LUGrid: update-tolerant grid-based indexing for moving objects. In: Proceedings of the 7th International Conference on Mobile Data Management. IEEE Computer Society (2006)
- Mindaugas, P., et al.: Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.* **31**(1), 255–298 (2006)
- Yiu, M.L., Tao, Y.F., Mamoulis, N.: The B-dual-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.* **17**(3), 379–400 (2008)
- Chen, N., et al.: Adaptive indexing of moving objects with highly variable update frequencies. *J. Comput. Sci. Technol.* **23**(6), 998–1014 (2008)
- Jensen, C., et al.: Indexing the Trajectories of Moving Objects in Symbolic Indoor Space Advances in Spatial and Temporal Databases, pp. 208–227. Springer, Berlin (2009)

17. Chen, N., et al.: B(S)-tree: a self-tuning index of moving objects, part II. In: Proceedings of Database Systems for Advanced Applications, pp. 1–16 (2010)
18. Lin, H.Y.: Indexing the trajectories of moving objects. In: Iccs 2009: International Multi-Conference of Engineers and Computer Scientists, vol. I, II, pp. 732–737. International Association Engineers-IAENG, Hong Kong (2009)
19. Lin, H.Y.: Using compressed index structures for processing moving objects in large spatio-temporal databases. *J. Syst. Softw.* **85**(1), 167–177 (2012)
20. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. *Acta Inf.* **1**(3), 173–189 (1972)
21. Guttman, A.: R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.* **14**(2), 47–57 (1984)
22. Nguyen-Dinh, L.-V., Aref, W.G., Mokbel, M.F.: Spatio-temporal access methods: part 2 (2003–2010). *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2010)
23. Norbert, B., et al.: The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.* **19**(2), 322–331 (1990)
24. Tao, Y., Papadias, D., Sun, J.: The TPR*-tree: an optimized spatio-temporal access method for predictive queries (2003)
25. Lin, H.-Y.: Using B+-trees for processing of line segments in large spatial databases. *J. Intell. Inf. Syst.* **31**(1), 35–52 (2008)
26. Huang, M.L., Hu, P., Xia, L.F.: A grid based trajectory indexing method for moving objects on fixed network. In: 2010 18th International Conference on Geoinformatics (2010)
27. Chen, H., et al.: Recent Trends in Wireless and Mobile Networks. A cyclic-translation-based grid-quadtrees index for continuous range queries over moving objects, pp. 95–109. Springer, Berlin (2011)
28. Nievergelt, J., Hans, H., Kenneth, C.S.: The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.* **9**(1), 38–71 (1984)



Gang Liu is a professor of School of Computer, China University of Geosciences at Wuhan, China. He received his PhD in Geodetection and Information Technology from China University of Geosciences in 2004. From 2006 to 2007 he stayed at University of Ottawa, Canada as a Post-doctorate Fellow. His research interests include geoscience information system engineering, 3D geographical information system and spatio-temporal geoscience data analysis.



Zufang Zheng is a lecturer of School of Industrial Design at Hubei University of Technology, Wuhan, China. She received her PhD in computer science from School of Computer, China University of Geosciences, China, in 2014. Her research interests are Spatial Database and Industrial Design.



Zhenwen He is an associate professor of computer science at China University of Geosciences, Wuhan, China. He received his PhD from Huazhong University of Science and Technology, China, in 2008, and completed his post-doctorate research at ITC's Geo-Information Processing Department of the University of Twente in the Netherlands, in 2013. His research interests include spatial-temporal database and 3D geosciences information system.



Yiping Tian is a professor of School of Computer, China University of Geosciences at Wuhan, China. He received his PhD from China University of Geosciences in 2001. His research interests include geoscience information system engineering, 3D geological information system and petroleum pool-forming dynamics simulation.



Chonglong Wu is a senior professor of School of Computer, China University of Geosciences at Wuhan, China. His research interests include geological information science and technology, geoscience information system engineering and petroleum pool-forming dynamics simulation.