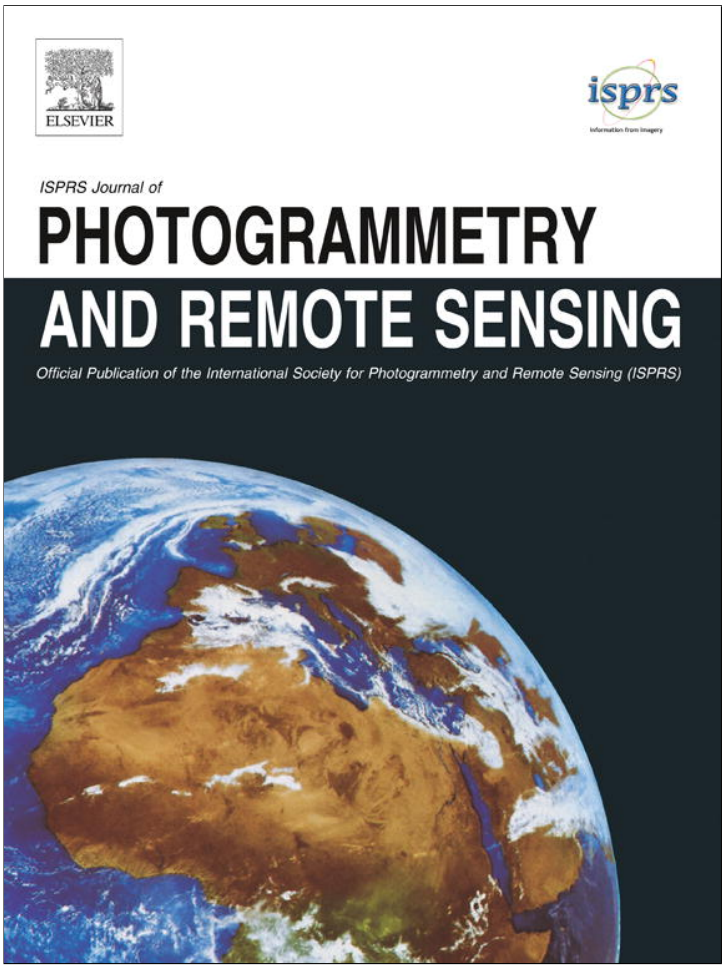

(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

ISPRS Journal of Photogrammetry and Remote Sensing 78 (2013) 116–128

Contents lists available at SciVerse ScienceDirect

ISPRS Journal of Photogrammetry and Remote Sensing

journal homepage: www.elsevier.com/locate/isprsjprs

Parallel indexing technique for spatio-temporal data

Zhenwen He [a,b,*], Menno-Jan Kraak [b], Otto Huisman [b], Xiaogang Ma [b], Jing Xiao [b]

[a] School of Computer, China University of Geosciences (Wuhan), 388 Lumo Road, Wuhan 430074, China
[b] Faculty of Geo-Information Science and Earth Observation, University of Twente, Hengelosestraat 99, 7514 AE Enschede, Netherlands

ARTICLE INFO

Article history:
Received 22 June 2012
Received in revised form 24 January 2013
Accepted 25 January 2013

Keywords:
Spatio-temporal index
Parallel index
R-Tree
Interval

ABSTRACT

The requirements for efficient access and management of massive multi-dimensional spatio-temporal data in geographical information system and its applications are well recognized and researched. The most popular spatio-temporal access method is the R-Tree and its variants. However, it is difficult to use them for parallel access to multi-dimensional spatio-temporal data because R-Trees, and variants thereof, are in hierarchical structures which have severe overlapping problems in high dimensional space. We extended a two-dimensional interval space representation of intervals to a multi-dimensional parallel space, and present a set of formulae to transform spatio-temporal queries into parallel interval set operations. This transformation reduces problems of multi-dimensional object relationships to simpler two-dimensional spatial intersection problems. Experimental results show that the new parallel approach presented in this paper has superior range query performance than R*-trees for handling multi-dimensional spatio-temporal data and multi-dimensional interval data. When the number of CPU cores is larger than that of the space dimensions, the insertion performance of this new approach is also superior to R*-trees. The proposed approach provides a potential parallel indexing solution for fast data retrieval of massive four-dimensional or higher dimensional spatio-temporal data.

1. Introduction

The world is three-dimensional and time is always going forward. Essentially, the real world is a four-dimensional spatio-temporal integrated space. Each object in the world has a presence in both time and space. The temporal dimension marks the lifecycle of an object in terms of its starting and ending times, and also establishes the validity of data. Data valid today may have had no meaning in the past, and similarly, may hold no identity in the future. Location represents the *x*, *y*, *z*, location of an object, which may change over time. This relationship with time adds a temporal identity to most data in a spatial information system. However, the time-varying data with spatial locations is difficult to manage effectively and efficiently. Most current spatial database systems represent a single state of the spatial data and this is most commonly assumed to be its current state. Any modifications normally result in the overwriting of the data with the old data being discarded. Although commercial database management systems offer some capabilities to keep track of old and discarded data, this is solely for the purpose of database recovery and not to retain the previous state of the data.

In the last decades, research on spatio-temporal databases has advanced in various aspects and reported many important results, however, many challenges still remain (Lin, 2012; Ni and Ravishankar, 2007; Stantic et al., 2010). In most previous studies, core concepts of temporal index methods and spatial index methods have been established, but it is not yet shown how they can be applied to manage spatio-temporal data efficiently. Generally, spatio-temporal databases are append-only and usually very large in size. Hence, an efficient access method for spatio-temporal databases is even more important than for conventional databases. Numerous indexing methods have been proposed, however, most of them are usually customized either specifically for temporal data or for spatial data, and most of them cannot adequately handle integrated spatio-temporal data (Ang and Tan, 1995; Berchtold et al., 1996; Ciaccia et al., 1997; Curtis and Michael, 1991; Eo et al., 2006; Ibrahim and Christos, 1993; Kim et al., 2007; Lin, 2008, 2012; Saltenis and Jensen, 2002; Stantic et al., 2010; Zhu et al., 2007). This is because indexing methods often divide the spatio-temporal index into two parts, temporal index and spatial index. This particular constraint makes these access methods unsuitable for many practical spatio-temporal applications.

Many multi-dimensional index methods have been proposed. Some of them have been recommended for handling temporal data (Betty and Vassilis, 1999; Kumar et al., 1998; Stantic et al., 2010), and some of them have been reported for handling spatial data

* Corresponding author at: School of Computer, China University of Geosciences (Wuhan), 388 Lumo Road, Wuhan 430074, China.
E-mail address: zwhe@foxmail.com (Z. He).

0924-2716/$ - see front matter © 2013 International Society for Photogrammetry and Remote Sensing, Inc. (ISPRS) Published by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.isprsjprs.2013.01.014

effectively, such as R-Tree (Guttman, 1984), R*-Tree (Norbert et al., 1990), Segment R-Tree (Curtis and Michael, 1991), packing R-Tree (Ibrahim and Christos, 1993), X-tree (Berchtold et al., 1996), M-tree (Ciaccia et al., 1997), 3D R-Tree (Zhu et al., 2007) and some other R-Tree variants (Balasubramanian and Sugumaran, 2012). Most of the spatio-temporal index methods are extensions of data partitioning spatial index methods, for instance, X-Tree and Segment R-Tree. The multi-dimensional R-Tree and its variants can be used as spatio-temporal index structures in theory. These use a spatio-temporal containment hierarchy that clusters data into bounding regions of the parent node, forming a hierarchical tree structure. These bounding regions may not represent the entire data scope and could overlap. Overlapping is a problem for data partitioning index methods because even for a simple point query it may need to examine multiple paths. Especially when they are used in now-relative temporal data and movement data, significant overlapping between nodes and dead space will cause very poor performance of the index (Lin, 2012; Saltenis and Jensen, 2002). In order to solve these problems caused by moving objects data, some spatio-temporal indexing methods based on R*-trees (Norbert et al., 1990) were proposed, such as TPR-tree (Choi et al., 2004; Lin and Su, 2004; Saltenis et al., 2000), TPR*-tree (Kim et al., 2010; Tao et al., 2003), RTR-tree and TP^{2}R-tree (Jensen et al., 2009). These indexing methods mainly improve the continuous update performance of R*-trees and R-trees via the time parameter and velocity parameter. Consequently, they only fit moving objects data, mainly the trajectory data, but not more general spatio-temporal data, such as fields. If we want to handle general spatio-temporal data, we should go back to R-trees and R*-trees, since they are basic data structures for multi-dimensional range query.

Parallel computing is a growing trend in handling massive spatio-temporal datasets, especially the high-dimensional data. Nevertheless, R-Trees are not suitable for applications in high dimensional spaces, because their structures are not suitable for parallel computing. Parallel computing is a form of computation in which many calculations are carried out simultaneously. At present, computer capabilities have increased due to the application of parallelism, for example in multicore processors (Asanovic et al., 2006; Knysh and Kureichik, 2010). However, most of the spatial or temporal indexing methods mentioned previously are implemented as traditional sequential algorithms. Some of these can only handle temporal data or interval data effectively, some can only handle spatial data effectively, and others can be used as general spatio-temporal indexes (such as R-Trees, R*-trees, and some of their variants). But all are difficult to implement as parallel algorithms.

We intend to propose a parallel access method for general spatio-temporal data. The fundamental idea behind this approach is to transform both temporal data and spatial data into multi-dimensional intervals, and then deploy a parallel access method for the multi-dimensional intervals. A number of access methods for interval data have been proposed, including the Interval B-Tree (Ang and Tan, 1995), RI-Tree (Brochhaus et al., 2005), TD-Tree (Stantic et al., 2010) and compressed B$^+$-Tree (Lin, 2008, 2012). The compressed B$^+$-Tree mainly focuses on movement data. Stantic et al. (2010) reported that TD-Tree is superior to RI-Tree which in turn is superior to Window-Lists (Ramaswamy et al., 1997), the Oracle tile index and IST-techniques. The TD-Tree is aimed at one- dimensional time interval data. As a result, it cannot be used in four-dimensional spatio-temporal data directly, although it is efficient for some certain temporal queries. Nevertheless, its triangular decomposition strategy is a good idea. The methods proposed in this paper represent extensions of this approach. Specifically, we will use it as one of parallel branches in our algorithm.

In this paper we present an integrated parallel index method, the "Multi-dimensional Parallel Binary Tree" (MPB-Tree) for general four-dimensional spatio-temporal data. In contrast to previously proposed access methods for intervals or R-Tree variants, this method can efficiently answer a wide range of multi-dimensional spatio-temporal query types using a uniform algorithm. The MPB-Tree is a parallel space partition access method. The basic idea is to manage four-dimensional or multi-dimensional interval using multi-dimensional parallel binary trees. This multi-dimensional index structure relies upon multiple parallel two-dimensional representations of intervals. The MPB-Tree partitions space using a parallel triangular decomposition method extending from the TD-Tree's triangular decomposition strategy. The resulting multi-dimensional parallel binary trees store a bounded number of intervals. The empirical performance of the MPB-Tree is demonstrated to be superior to the R*-tree for general four-dimensional spatio-temporal data.

The remainder of this paper is organized as follows: In the following section, we describe the representation of spatio-temporal data and interval data. In Section 3, we describe the parallel indexing algorithms for patio-temporal data. Section 4 documents the algorithm, experiment and analysis of results using the improved method. Finally we present our conclusion in Section 5.

2. Representation of spatio-temporal data and intervals

We assume a four-dimensional spatio-temporal model in the range [*TMin*, *XMin*, *YMin*, *ZMin*, *TMax*, *XMax*, *YMax*, *ZMax*], for some *TMax* > *TMin*, *XMax* > *XMin*, *YMax* > *YMin* and *ZMax* > *ZMin*. In the four-dimensional space, each point can be described as (*t, x, y, z*) and each spatio-temporal object has a minimum bounding rectangle (MBR) in the range [*TOMin*, *XOMin*, *YOMin*, *ZOMin*, *TOMax*, *XOMax*, *YOMax*, *ZOMax*]. We employ this four-dimensional MBR to represent each spatio-temporal object, like the R-Trees (Guttman, 1984; Nievergelt et al., 1984; Norbert et al., 1990). We can transform this four-dimensional MBR into four intervals: [*TOMin, TOMax*], [*XOMin, XOMax*], [*YOMin, YOMax*] and [*ZOMin, ZOMax*]. For example, there is a building whose spatial MBR is in the range [100,200] in *X* axis, the range [150,400] in *Y* axis, and the range [300,700] in *Z* axis. This building was built in 1900, and it was destroyed in 1978. Then this building may be represented by four intervals: [1900,1978], [100,200], [150,400] and [300,700]. In general, we can describe them in a uniform:

$$[Is(d), Ie(d)],\ IMin \leqslant Is(d) \leqslant Ie(d) \leqslant IMax,$$

in which *IMax* is the maximum value of the intervals, *IMin* is the minimum value of the intervals, *Is* is the starting of interval, *Ie* is the ending of interval, and $d = 0, 1, \ldots, n-1$. This representation can be extended to *n*-dimensional space, since in four-dimensional space, $n = 4$. For each dimension of this space, an interval can be described as a point in a two-dimensional plane (Stantic et al., 2010) as shown in Fig. 1. The axis *S* is the starting value of the interval, and axis *E* is the ending value of the interval. Because *Ie* is always larger than *Is*, the triangle of (*IMin, IMin*), (*IMax, IMin*), (*IMax, IMax*) is always empty in the two-dimensional space. If the dimension is four, there will be four parallel two-dimensional spaces. This offers the possibility of handling the intervals in four-dimensional space using a parallel method extended from the two-dimensional space, and the possibility for reducing query computation by skipping the empty triangular regions.

For general intervals, Allen (1983) described 13 distinct interval algebra relationships that may hold between two intervals, as shown in Fig. 2. Each of the 13 interval algebra relationships can be represented as a point, line and region (Figs. 3 and 4) in the two-dimensional space, or the plane *SE*. If the universal interval of any one dimension is [*Is, Ie*], and the input interval of a query

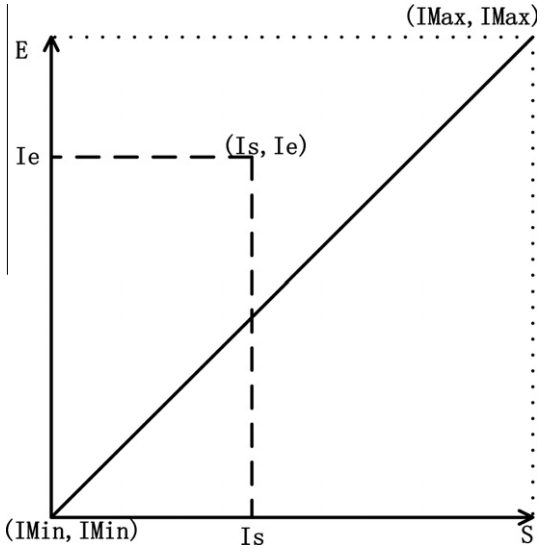

Fig. 1. Interval representation in two-dimensional space (after Stantic et al. (2010)).

is [*Qs, Qe*], we can represent the relationships of two intervals in two-dimensional space, as defined below.

(a) Equals Query: $Is = Qs$ and $Ie = Qe$.
(b) Starts Query: $Is = Qs$ and $Qs < Ie < Qe$; as shown in Fig. 3a.
(c) StartedBy Query: $Is = Qs$ and $Ie > Qe$; as shown in Fig. 3b.
(d) Meets Query: $Is < Ie = Qs < Qe$; as shown in Fig. 3c.
(e) MetBy Query: $Qs < Qe = Is < Ie$; as shown in Fig. 3d.
(f) Finishes Query: $Qs < Is < Qe$ and $Ie = Qe$; as shown in Fig. 3e.
(g) FinishedBy Query: $Is < Qs$ and $Ie = Qe$; as shown in Fig. 3f.
(h) Before Query: $Is < Ie < Qs < Qe$; as shown in Fig. 3a.
(i) After Query: $Qs < Qe < Is < Ie$; as shown in Fig. 4b.
(j) Overlaps Query: $Is < Qs$ and $Qs < Ie < Qe$; as shown in Fig. 4c.
(k) OverlappedBy Query: $Qs < Is < Qe$ and $Ie > Qe$; as shown in Fig. 4d.
(l) During Query: $Qs < Is < Ie < Qe$; as shown in Fig. 4e.
(m) Contains Query: $Is < Qs < Qe < Ie$; as shown in Fig. 4f.

If the intervals can be open sets, for example, the input interval of a query $\boldsymbol{Q}$ is (*Qs, Qe*) and the universal interval $\boldsymbol{U}$ may be (*Is, Ie*), the Equals Query may be extended to two other queries:

(l) Covers Query: $Is = Qs$, $Ie = Qe$, Q is close and U is open.
(n) CoveredBy Query $Is = Qs$, $Ie = Qe$, Q is open and U is close.

Figs. 3 and 4 present the transformation of a general interval query into a simpler spatial query in two-dimensional space. As shown in Fig. 3 relationships between two intervals such as 'Starts', 'StartedBy, 'Meets', 'MetBy', 'Finishes', 'FinishedBy' and 'Equals', can be queried efficiently by a one-dimensional index, for instance a B-Tree. This is because those seven types of queries can be reduced to a simple comparison of two points. Because 'Covers' and 'CoveredBy' extend from 'Equals', they can also be efficiently queried by a one-dimensional index. However the other six relationships, 'Before', 'After', 'Overlaps', 'OverlappedBy ', 'During', and 'Contains', shown in Fig. 4 cannot be handled this way. These queries need other special index methods. We can see that the results of these six queries are all regions, rectangles or triangles. Because the starting value is always smaller than the ending value, the triangles, shown in Fig. 4a, b and e, can be extended to rectangles without influence on the query results. Therefore, these six queries can be unified as a rectangle query in the two-dimensional space.

The next step is to transform the spatio-temporal query into interval queries. Generally speaking, two objects A and B, whether in three-dimensional space, four-dimensional space or multi-dimensional space, may hold 8 fundamental relationships (Egenhofer et al., 1989): either A and B are disjoint, as shown in Fig. 5a, A meets B, as shown in Fig. 5b, A overlaps B, as shown in Fig. 5c, A equals B, as shown in Fig. 5d, A contains B, as shown in Fig. 5e, A is contained by or inside B, as shown in Fig. 5f, A covers B, as shown in Fig. 5g, or A is covered by B, as shown in Fig. 5h. As mentioned above, we represent a 4D spatio-temporal object as a 4D interval (or 4 intervals in the two-dimensional space). We assume that the object B represents the input query region which can be represented as $[Qs(d), Qe(d)]$, and $d = 0, 1, 2, 3$ if the dimension is 4. In this way, we can transform the spatio-temporal relationships into interval operations, as defined below.

(1) Spatio-temporal **Disjoint** Query: find all the objects that are disjoint with B. The resulting set is the union of 'Before' operator and 'After' operator for each dimension interval.

$$\text{Disjoint}(B) = \cup_{d=0,1,2,3}(\text{Before}([Qs(d), Qe(d)])) \vee \cup_{d=0,1,2,3}(\textit{After}([Qs(d), Qe(d)]));$$

(2) Spatio-temporal **Meets** Query: find all the objects that meet object B. The resulting set is the union of 'Meets' operator and 'MetBy' operator for each dimension interval.

$$\text{Meets}(B) = \cup_{d=0,1,2,3}(\text{Meets}([Qs(d), Qe(d)])) \vee \cup_{d=0,1,2,3}(\text{MetBy}([Qs(d), Qe(d)]));$$

(3) Spatio-temporal **Overlaps** Query: find all the objects that overlap B. The resulting set is the union of two subsets, one is the intersection of 'Overlaps' operator for each dimension interval, and the other is the intersection of 'OverlappedBy ' operator for each dimension interval.

$$\text{Overlaps}(B) = \cap_{d=0,1,2,3}(\text{Overlaps}([Qs(d), Qe(d)])) \vee \cap_{d=0,1,2,3}(\text{OverlappedBy}([Qs(d), Qe(d)]));$$

(4) Spatio-temporal **Equals** Query: find all the objects that equal B. The resulting set is the intersection of 'Equals' operator for each dimension interval.

$$\text{Equals}(B) = \cap_{d=0,1,2,3}(\text{Equals}([Qs(d), Qe(d)]));$$

(5) Spatio-temporal **Contains** Query: find all the objects that contain B. The resulting set is the intersection of 'Contains' operator for each dimension interval.

$$\text{Contains}(B) = \cap_{d=0,1,2,3}(\text{Contains}([Qs(d), Qe(d)]));$$

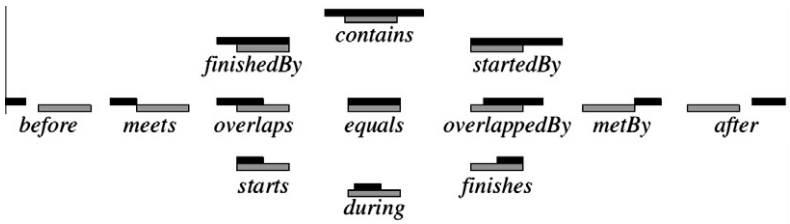

Fig. 2. Thirteen general interval relationships (Allen, 1983; Kriegel et al., 2001).

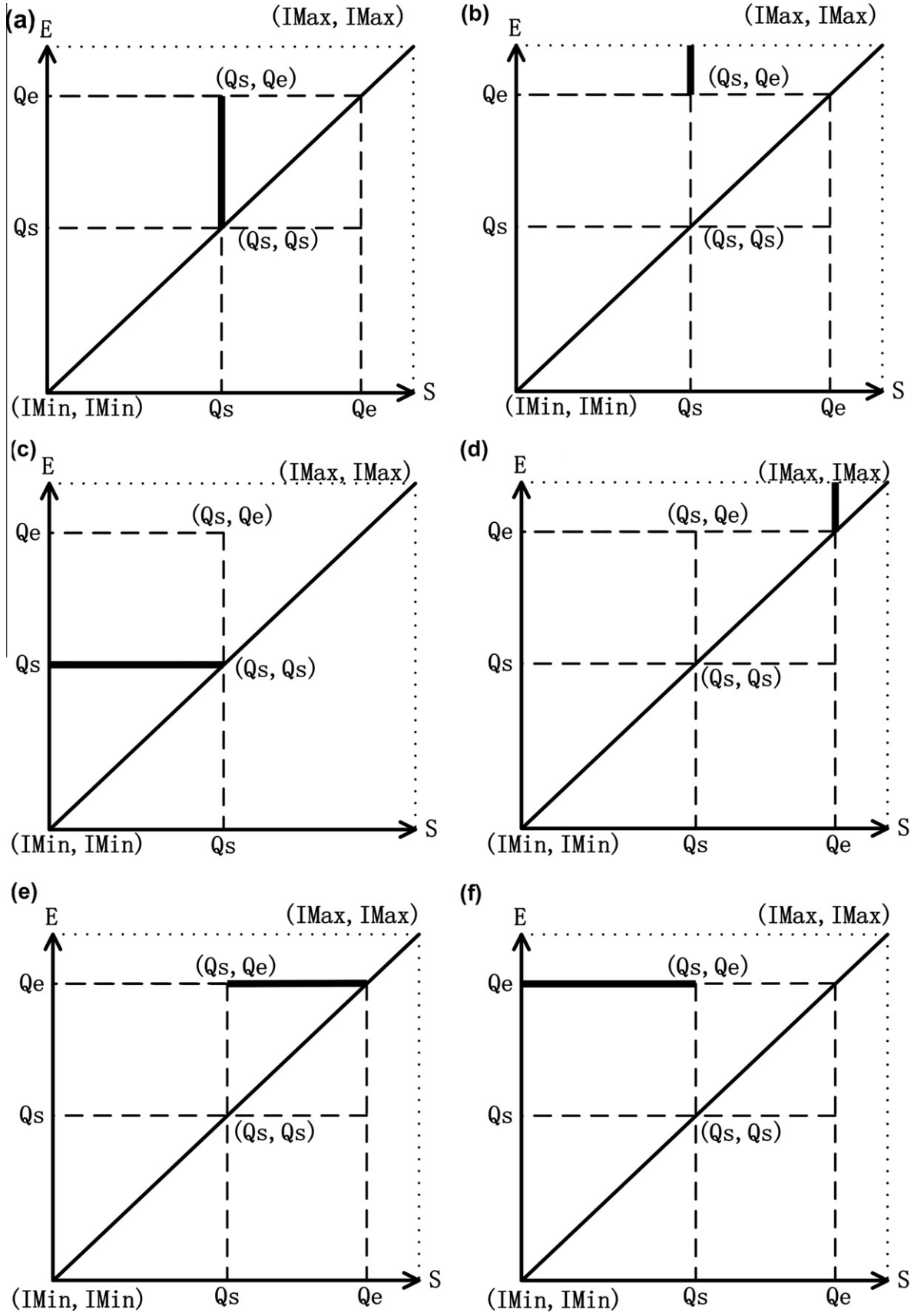

Fig. 3. Line queries in two-dimensional space. (a) Starts query. (b) StartedBy query. (c) Meets query. (d) MetBy query. (e) Finishes query. (f) FinishedBy query.

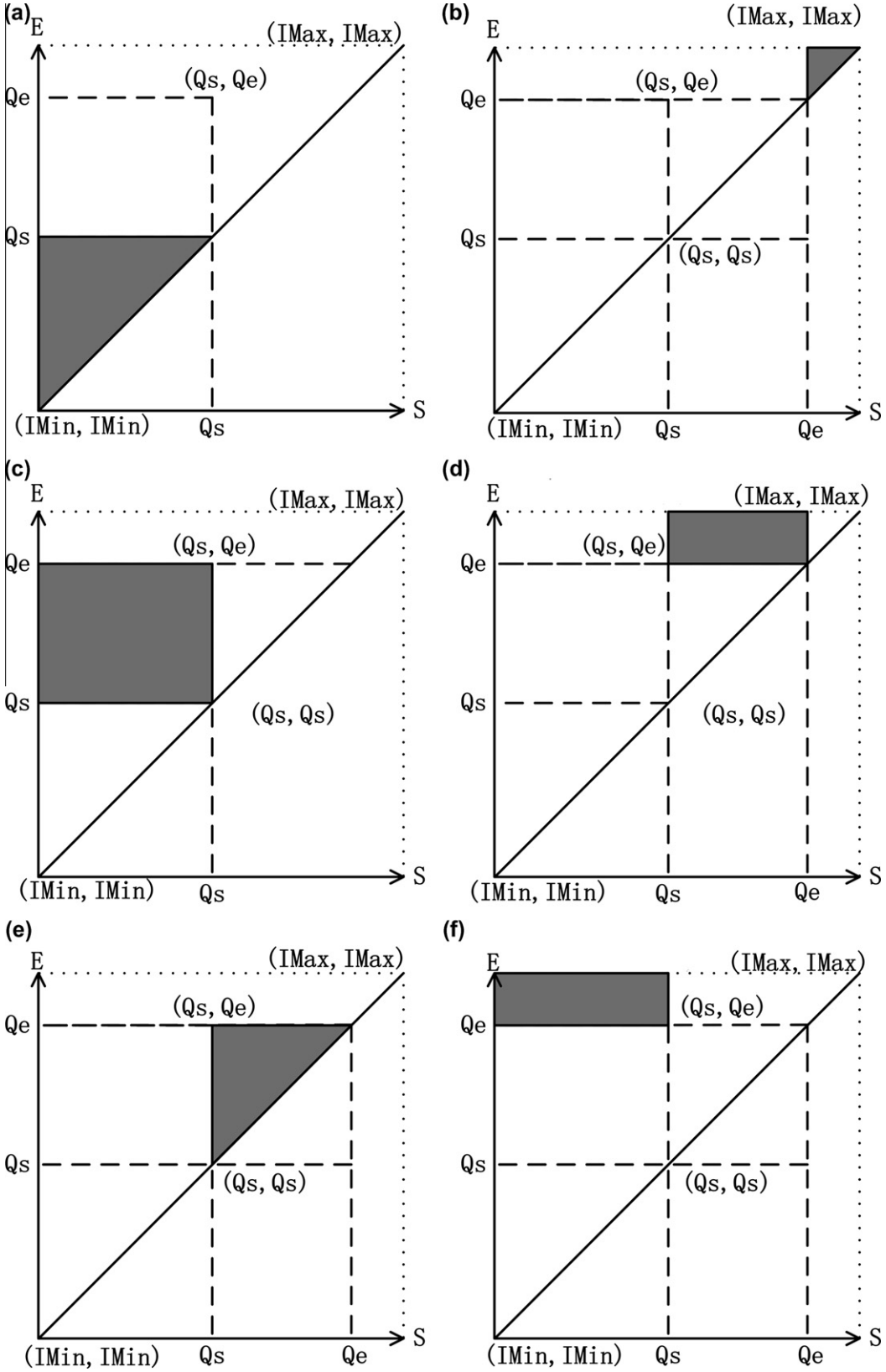

Fig. 4. Region query in two-dimensional space (after Stantic et al. (2010)). (a) Before query. (b) After query. (c) Overlaps query. (d) OverlappedBy query. (e) During query. (f) Contains query.

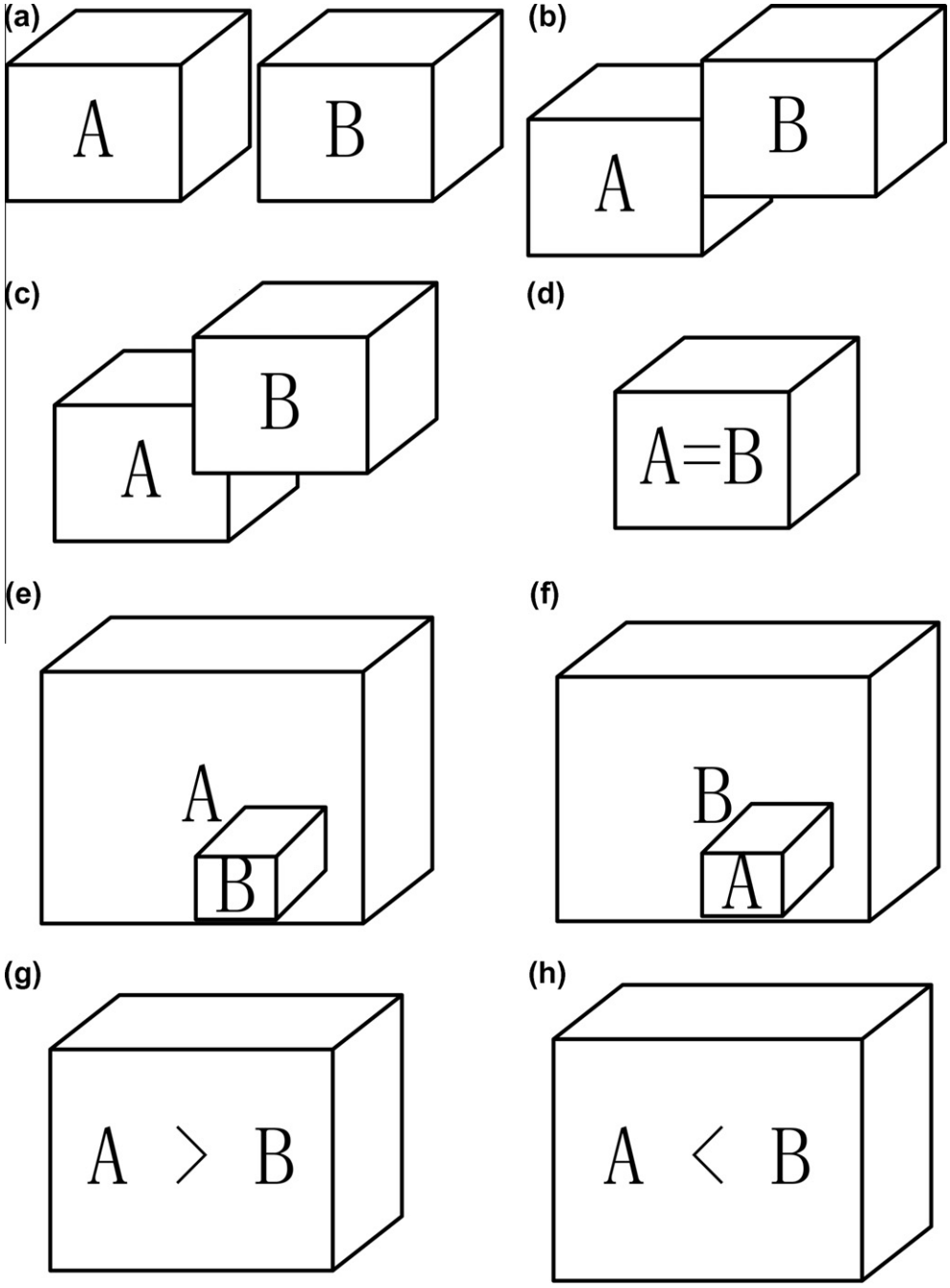

Fig. 5. Spatio-temporal object relationships. (a) A and B are disjoint. (b) A meets B. (c) A overlaps B. (d) A equals B. (e) A contains B. (f) A is contained by B. (g) A covers B. (h) A is covered by B.

(6) Spatio-temporal **ContainedBy** Query: find all the objects that are contained by B. The resulting set is the intersection of 'ContainedBy' operator for each dimension interval.
$\mathrm{ContainedBy}(\mathrm{B}) = \cap_{d=0,1,2,3}(\mathrm{During}([Qs(d), Qe(d)]))$;

(7) Spatio-temporal **Covers** Query: find all the objects that cover B. The resulting set is the intersection of 'Covers' operator for each dimension interval.
$\mathrm{Covers}(\mathrm{B}) = \cap_{d=0,1,2,3}(\mathrm{Covers}([Qs(d), Qe(d)]))$;

(8) Spatio-temporal **CoveredBy** Query: find all the objects that are covered by B. The resulting set is the intersection of 'CoveredBy' operator for each dimension interval.
$\mathrm{CoveredBy}(\mathrm{B}) = \cap_{d=0,1,2,3}(\mathrm{CoveredBy}([Qs(d), Qe(d)]))$;

As mentioned previously, the 'Meets', 'Equals', 'Covers' and 'CoveredBy' queries can be handled efficiently by a one-dimensional index, so we will not discuss these further in this paper. For the convenience of discussion, we combine 'Spatio-temporal Contains Query' and 'Spatio-temporal ContainedBy Query' with 'Spatio-temporal Overlaps Query' to form one query: the 'Intersection Query'. Like this, the absolute complement set of intersection query is the result of disjoint query.

3. Parallel indexing algorithm for spatio-temporal data

The Multi-dimensional Parallel Binary Tree (MPB-Tree) is an integrated parallel index method for four-dimensional spatio-temporal data, which is composed of four binary trees (each binary

tree is a Triangular Binary Tree, called TB-Tree) and a shared memory pool which stores all the four-dimensional intervals, as shown in Fig. 6. Each TB-Tree represents a dimension of the spatio-temporal space. As shown in Fig. 1, all data and query intervals represented in two-dimensional space (the SE plane) lie in the isosceles right-angle triangle with three vertices at (*IMin, IMin*), (*IMin, IMax*) and (*IMax, IMax*), which lies above the line *Is = Ie*. This is because *Is*, the starting value, is always smaller than the ending value *Ie*. We call this triangle the basic triangle of one dimension of the four-dimensional space. Obviously in four-dimensional space there are four basic triangles in the MPB-Tree. Each TB-Tree of the MPB-Tree is based upon the decomposition of this basic triangle. The triangle decomposition strategy is same as that of the TD-Tree (Stantic et al., 2010). The main idea is to recursively decompose the basic triangle into two smaller triangles.

Because TB-Tree and TD-Tree share the same decomposition strategy, they have the same hierarchical structures. The triangle decomposition strategy of the TD-Tree decides that the TD-Tree as well as the TB-Tree is essentially a binary tree. The only difference is their data structures representing the hierarchical structures. That is, the TD-Tree is a virtual binary tree using a linear representation while the TB-Tree uses the original representation of binary tree. In theory, MPB-Tree is a template whose sub tree in each dimension can be any index structure supporting interval data, such as a TD-Tree and a TB-Tree. So we can theoretically use the TD-Tree as the sub tree of an MPB-Tree. There are several reasons why we still use a TB-Tree instead of a TD-Tree though the TD-Tree can offer more efficient storage than the TB-Tree used in this paper.

The first reason is about the identifier of a TD-Tree's node. When the data set is very large, the length of a binary identifier may be larger than 32 bits or 64 bits. If one of the identifiers exceeds 32 bits or 64 bits, all the identifier's extensions will exceed 32 bits or 64 bits. It is no longer easy to use an integer to represent a node's identifier. One of the alternatives is replacing the integer with a string. However, all the benefits of using an integer as an identifier will disappear. Moreover, the memory cost of a string containing 64 characteristics is at least 64 bytes, which is larger than the memory cost of a triangle in two-dimensional space (if the data type of the point is 64 bits double, the memory cost is 48 bytes; if the data type of the point is 32 bits float, the memory cost is 24 bytes).

The second reason is that each triangle region of a TD-Tree node needs to be computed in real time. As a result, the longer the identifier of a TD-Tree's node is, the more time cost the computation will incur. This real time strategy is not suitable for continuous queries and large region queries. The pointer operation is much faster than this real time computation. The last reason is that the memory limitation is no longer a main obstacle for some algorithms with the fast development of computer's hardware. Gener-

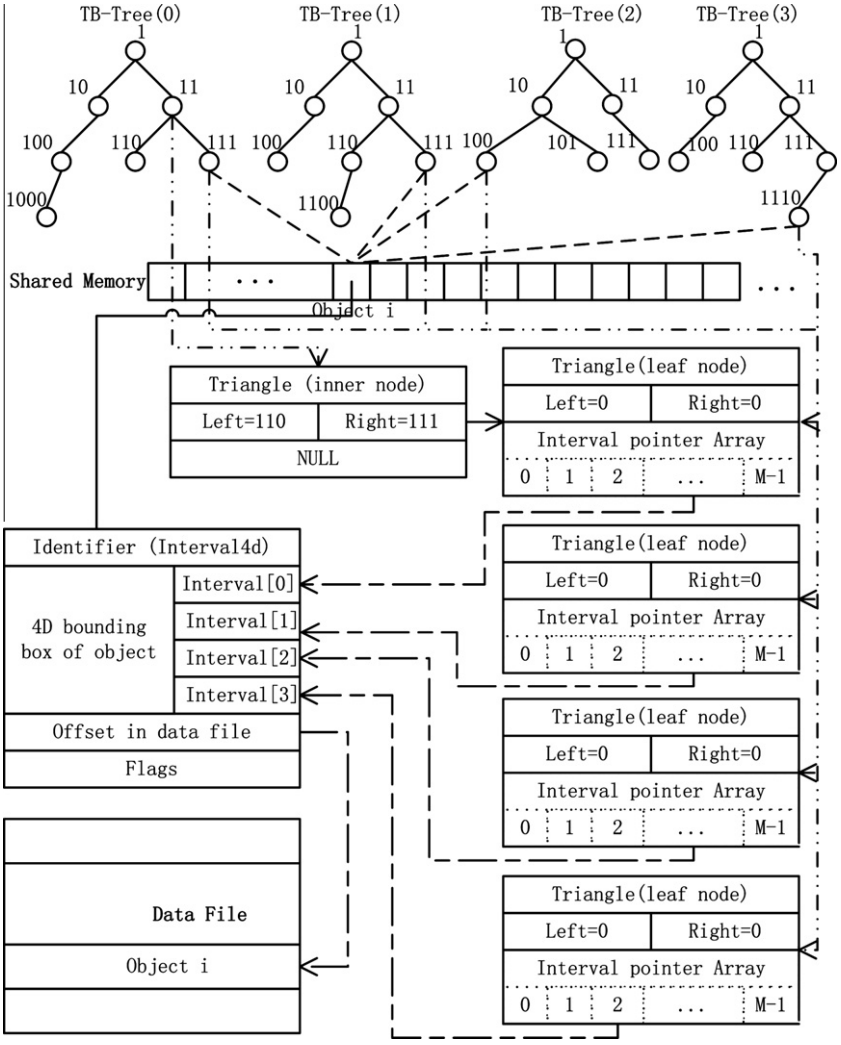

Fig. 6. MPB-Tree structure.

ally, the TB-Tree is a variant of the TD-tree which uses more memory space to improve the query performance of MPB-tree. The TB-Tree's node can be defined as:

```
struct TreeNode {
IsoscelesRightTriangle _triangle;
TreeNode*              _left;
TreeNode*              _right;
IntervalPointerArray*  _intervals;
};
```

The member '*_triangle*' stores the parameters of the isosceles right-angle triangle which is covered by this node. If the node is a leaf, the left pointer and right pointer both are null, and the interval pointer array '*_intervals*' may have pointers pointing to intervals in the shared memory pool. The maximum number of the intervals contained by the array is MAX_NUMBER_PERNODE which is same as the block factor of the TD-Tree (Stantic et al., 2010) and the parameter M of R-trees (Guttman, 1984; Nievergelt et al., 1984; Norbert et al., 1990). If the node is an inner node, the left pointer points to the left sub-tree and the right pointer points to the right sub-tree. Meanwhile the interval pointer array is empty and the pointer '*_intervals*' is null. The inner node structure and the leaf node structure of TB-Tree were shown in Fig. 6.

Each item in the shared memory pool is defined as below:

```
struct Interval4d {
  Identifier      _identifier;
  Interval        _data[4];
  unsigned long   _offset;
  unsigned char   _flags[4];
};
```

The member '*_identifier*' is a 64 bits integer number that can identify a spatio-temporal object uniquely. The member '*_data*' stores 4 intervals representing the 4D MBR of the spatio-temporal object in four-dimensional space. Each of the 4 intervals will be pointed by a pointer from a TB-Tree's leaf node as shown in Fig. 6 after the MPB-Tree was constructed. The member '*_flags*' is a status array corresponding to the 4 intervals in the '*_data*' array. If the 0 dimension interval has been traveled, the '*_flags[0]*' should be set to 1, otherwise it is 0. So if all of the four intervals have been traveled, the '*_flags*' is a 32 bits integer number which equals 0x01010101. The member '*_offset*' is an offset from which the spatio-temporal object stores in a data file.

3.1. Insertion algorithm for the MPB-Tree

Insertion is the key step of MPB-Tree generation, in which new spatio-temporal objects are inserted into the MPB-Tree, and the rational structure of MPB-trees is formed. Insertion includes two important sub-operations: node choice and node split. Because for four-dimensional spatio-temporal objects the MPB-Tree is composed of four TB-Trees, the insertion operation is through four parallel TB-Tree insertions. Therefore, the key step of MPB-Tree insertion is how to insert an interval into a TB-Tree. To insert one interval into the TB-Tree, one starts from the root to recursively choose the left sub-tree and the right sub-tree until the leaf node is reached. If the new insertion of an interval into the leaf node leads to the overflow of a node, the overflowing node has to be split and divided into two small nodes.

3.1.1. Interval insertion algorithm of TB-Tree

As mentioned in Section 2, an interval can be represented as a 2D point. At first we find the leaf node which the interval point [*Is, Ie*] belongs to. If the number of intervals in the leaf is smaller than *MAX_NUMBER_PERNODE*, we push back the interval pointer into this leaf node. If the number of intervals in the leaf is equal to MAX_NUMBER_PERNODE, we should split the leaf node and add the interval pointer to a split leaf node which contains the interval point. The pseudocode of this algorithm is as following:

Algorithm 1: insert an interval pointer into a TB-Tree

Input: *pi:* an interval [*Is, Ie*]

Output: boolean

```
{
  call Algorithm 2 to find the leaf node which the interval
   point [Is, Ie] belongs to, let lfn be the leaf node;
  if (lfn-> intervalsNumber < MAX_NUMBER_PERNODE)
  {
    push back pi into this leaf node lfn;
  }
  else
  {
    call Algorithm 3 to split the leaf node lfn and add pi to a
   split leaf node which contains pi;
  }
}
```

Algorithm 2: find a leaf node which the interval belongs to

Input: *pi*: an interval [*Is, Ie*]; *root*: root node of a TB-Tree
Output: *lfn*: a leaf node of TB-Tree

```
{
  if (the triangle of the root node does not contain pi) return
   null;
  let left = root->left;
  let right = root->right;
  let parent = root;
  if (left is null and right is null) return root;
  while (both left and right are not null)
  {
    If (left contains pi)
    {
      let parent = left;
      let left = parent->left;
      let right = parent->right;
    }
    else
    {
      let parent = right;
      let right = parent->right;
      let left = parent->left;
    }
  }
  return parent;
}
```

3.1.2. Split algorithm of TB-Tree

Because every leaf node of a TB-Tree has a certain capacity MAX_NUMBER_PERNODE, a leaf node need to be split when the number of the intervals contained in it is larger than the maximum capacity. The main ideal of this algorithm is to recursively decompose the current leaf node's triangle into two sub triangle according to the TD-Tree's decomposition strategy until each leaf node's interval number is smaller than MAX_NUMBER_PERNODE. The pseudocode of this algorithm is as following:

Algorithm 3: split a leaf node and insert an interval into a TB-Tree

Input: *pi:* an interval [*Is, Ie*]; *parent*: a leaf node to be split

Output: void

```
{
  let left and right be two empty leaf nodes;
  decompose parent->triangle into two sub triangles (left-
  >triangle and right->triangle) according to the TD-Tree's
  decomposition strategy;
  for (i = 0; i < MAX_NUMBER_PERNODE; i++)
  {
  if (left contains parent->intervals[i])
    push parent->intervals[i] into left;
  else
    push parent->intervals[i] into right;
  }
while (left->intervalsNumber equals
  MAX_NUMBER_PERNODE or
  right->intervalsNumber equals MAX_NUMBER_PERNODE)
{
  if (left->intervalsNumber equals MAX_NUMBER_PERNODE)
    let parent be left;
  else
    let parent be right;
  let left and right be two empty leaf nodes;
  decompose parent->triangle into two sub triangles (left-
  >triangle and right->triangle) according to the TD-Tree's
  decomposition strategy;
  for (i = 0; i < MAX_NUMBER_PERNODE;i++)
  {
    if (left contains parent->intervals[i])
      push parent->intervals[i] into left;
    else
      push parent->intervals[i] into right;
  }
}
if (left contains pi)
  push pi into left;
else
  push pi into right;
}
```

3.1.3. MPB-Tree's insertion algorithm

After the implementation of TB-Tree's insertion algorithms, what the MPB-Tree's insertion algorithm does is to parallel call the TB-Tree's insertion algorithms. Given a certain four-dimensional interval 'interval4d' (see the definition of Interval4d structure in Section 3) objects, we push it into the shared memory pool. At last, we call the **Algorithm 1** given by Section 3.1.1 in parallel for each dimension interval of the interval4d.

3.2. Query algorithm

The key step of the MPB-Tree's query is deciding how to divide the spatio-temporal query into four parallel interval queries on each TB-Tree. Following the analysis of Section 2, every interval query corresponds to a 2D rectangular region in the two-dimensional interval space, as shown in Fig. 4. The task of the query is to find all the intervals that occur within this query rectangle. The particular region chosen depends on what type of the query we are performing. The research work (Stantic et al., 2010) resolved this problem. So if we implement the TB-Tree's 2D rectangle query, we can implement all the type queries listed in Fig. 4 by providing different 2D rectangle parameter.

3.2.1. TB-Tree's query algorithm

The TB-Tree's query algorithm is the base of MPB-Tree's query algorithm. Because TB-Tree is binary tree, we can implement the query algorithm by a recursive binary search algorithm. The pseudocode of this algorithm is as following:

Algorithm 4: find all the intervals contained by a 2D rectangle

Input: *root:* the root node of a TB-Tree; *rect2d*: a 2D rectangle
Output: *results:* identifier array of query results

```
{
  if (root is empty) return;
  if (root->triangle does not intersect with rect2d) return;
  let left be root->left and let right be root->right;
  if (root is a leaf node and root->intervalsNumber>0)
  {
    for (i = 0; i < root->intervalsNumber; i++)
    {
      if (rect2d contains root->intervals[i])
      {
        set the flag byte of the Interval4d object in shared
  memory
        corresponding to root->intervals[i];
        if (the flags equal to 0x01010101)
          push the identifier of the Interval4d object into
  results;
      }
    }
  }
  else
  {
    if (left->triangle intersects with rect2d)
      recursively call this algorithm with parameters left,
  rect2d and results;
    else
      recursively call this algorithm with parameters right,
  rect2d and results;
  }
}
```

3.2.2. Query algorithm of MPB-Tree

Given a four-dimensional minimum bounding rectangle (MBR) of a spatio-temporal object, it can be transformed to four two-dimensional rectangles. Each of them is one input parameter of **Algorithm 4**. So we call the **Algorithm 4** given by Section 3.2.1 in parallel for each dimension, then we can get the query results.

We should pay attention to the fact that the query algorithm mixed spatial and temporal dimensions, because it may cause some ambiguous query results. For example the meets query: two spatio-temporal objects meet if they meet in one dimension. So for instance, two buildings meet, if they have a different spatial location but one was demolished at the same time when the other was constructed. In this case, the results of this query algorithm does not show clearly whether these two buildings meet in time or in spatial locations. Hence, the flags of the return objects in the query results should be returned together according to some certain real applications. For example, if the flag of an object is 0x01000000, it indicates that these two buildings just meet in temporal dimension. If the flag is 0x01000001, it indicates that these two buildings meet in both temporal dimension and *Z* dimension. Additionally, if we want to separate the spatial query from the

temporal query, we can provide some advanced query functions based on this index because the four TB-Trees of MPB-Tree is independent. Another we should state clearly about the index is that all the results are not for the objects' geometry but for the 4D minimum bounding rectangles of the objects.

3.3. Deletion algorithm

Deletion and insertion are the bases of a dynamic index structure. Deletion algorithm removes an object in a leaf node, and if empty nodes appear, they should be removed too. The key of the deletion algorithm of MPB-Tree is the TB-Tree's deletion algorithm whose pseudocode is as following:

Algorithm 5: delete an interval in a TB-Tree

```
Input: root: the root node of a TB-Tree; pi: an interval to be
   removed from the TB-Tree
Output: void
{
  call the Algorithm 2 to find the leaf node containing the
   object to be deleted;
  let root be the leaf node;
  if (root is empty)
  {
    get root's parent node and remove root;
    let root be the root's parent node;
  }
  while (root does not have two children)
  {
    if (both root->left and root->right are null)
    {
      get root's parent node and let root be the root's parent
   node;
      if (root has only one child) remove this child node;
    }
  }
  remove the root's child node which is empty;
}
```

What the deletion algorithm of MPB-Tree requires to do is to call **Algorithm 5** in parallel and then remove the Interval4d object in shared memory.

3.4. Complexity of the algorithm

The query performance is the most important factor to evaluate an index. The MPB-Tree consists of four TB-Trees which are the basic unit of parallelism in the MPB-Tree. So the complexity of TB-Tree decides the performance of the MPB-Tree. Given that S is the total number of the spatio-temporal objects, N is the number of nodes in a TB-Tree, b is the maximum capacity of each leaf node of TB-Tree and h is the depth of the TB-Tree. Because the TB-Tree is essentially a binary tree, we know that $L \leqslant 2^h$, $N \leqslant 2^{h+1} - 1$, the average query complexity is $O(\log(N))$ and the worst query complexity is $O(N)$. According to the decomposition strategy, the N is related to the interval points' distribution in the two-dimensional plane shown in Fig. 1. If the interval points' distribution in the basic triangle is uniform, the TB-Tree will be a fully balanced binary tree. In that case, N is appropriate equal to $(2S/b - 1)$, so the best average query complexity is $O(\log(S/b))$. But the worst distribution may cause that N is equal to $S - 1$ in theory so that the worst query complexity is $O(S)$. Fortunately, when the data set is very large, this theoretical extreme case is impossible in fact. Because the MPB-Tree is a parallel method, the parallelism capability of the CPU may also influence its performance. We present an experimental evaluation in the following section.

4. Experimental evaluation

In order to verify the algorithm proposed in this paper, we performed an experimental evaluation of the MPB-Tree and compared it to the R*-Tree. The R*-Tree was chosen because it provides the same properties as our approach that can index general four-dimensional or multi-dimensional (>4) spatio-temporal data without redundant parameters, such as velocity and direction. The family of R-Trees is still the most popular index structure used for virtual geographic environments (Zhu et al., 2007), and used for multi-dimensional ($\geqslant 4$) spatio-temporal data retrieval (Balasubramanian and Sugumaran, 2012). Popular spatial database extensions provided by platforms such as Oracle, MySQL, IBM DB2, PostgreSQL, Microsoft SQL Server and so on, also use the R-Trees as their basic index structure for spatial data. Furthermore, many variants of R-Trees have evolved, each performing better in specific aspects, such as query retrieval, intersection cost, specific application, and so on. The R*-Tree was chosen here because it was found to outperform the R-Tree for all queries (Balasubramanian and Sugumaran, 2012; Norbert et al., 1990).

To compare the performance of the two index methods, two different types of 4D models were selected. One is a city model (DS-1) which is a part of real model data of a city of central China, and the other is a simulated random dataset (DS-2). The city model dataset is loaded dynamically, and is shown in part in Fig. 7. The experimental model includes 10,610 buildings, 60 roads, 1000 geological solids underground and some rivers, lakes and trees, as shown in Fig. 7a. Every feature object, for instance a building or a geological solid is a big group of geometry which is composed of triangles. The total number of the triangles of these models is 2,378,158. Because the original data is three-dimensional and does not include temporal information, the temporal interval of every feature object is generated randomly in order to create a four-dimensional test dataset. This is shown in Fig. 7b. The time interval points are distributed in the upper triangle of the two-dimensional space.

In order to keep the performance comparison manageable, the maximum number of the objects permitted in a tree node is 32 for R*-Tree, and the minimum is 16. The maximum capacity of a TB-Tree's leaf node is 32. Without special instructions, all the experimental results were computed on a notebook computer with the configuration of a 2.53 GHz Intel(R) Core(TM) Duo CPU, 2 GB RAM. This computer has two cores. All parallel algorithms are implemented using openMP 2.0 for C++ and compiled using the Microsoft visual C++ 2010 compiler.

For the city model dataset (DS-1), the area involved in the region queries varies from 0.1%, 1% to 10% of the whole model space. The experimental results of query and insertion are listed in Table 1. Each query time cost is an average of 1000 queries with random time interval, random spatial locations in the dataset ranges and certain ratio, for example 0.1%. The insertion time is the total time cost for inserting all the spatio-temporal objects in the city model dataset into an index, MPB-Tree or R*-Tree. The results in Table 1 show that MPB-Tree is superior to R*-Tree not in insertion performance but in query performance.

For the simulated models (DS-2), we randomly generated from 1000 to 30,000 objects (4D rectangles, each side of length of a rectangle varies from 1 to 10,000) with the step 1000. Therefore we got 30 group datasets and used them for the insertion and query performance test of these two index methods. The test results are shown in Fig. 8. Each point in the Fig. 8a represents an average query time cost for a certain group dataset. The average query time cost is the average of 1000 random queries' time costs. Each query

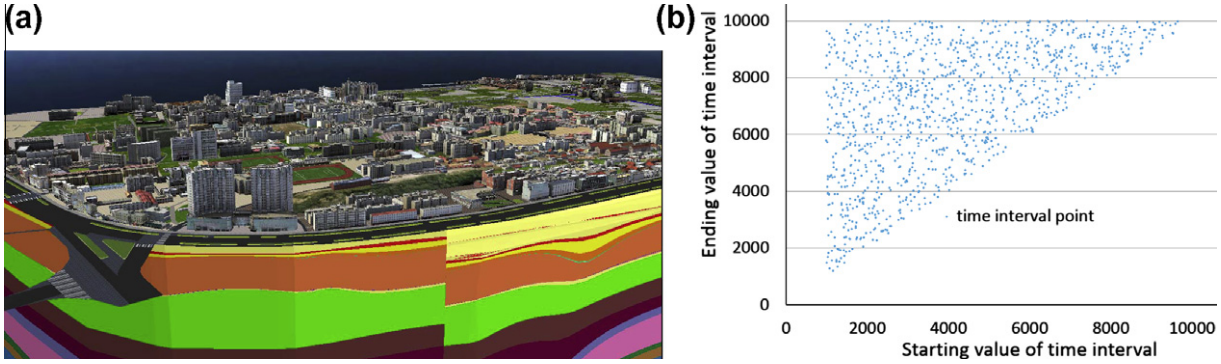

Fig. 7. City model data (geographical model and geological model). (a) Spatio-temporal objects in 3D view. (b) Time intervals in two-dimensional space.

Table 1
Query and insertion time for city model data (DS-1).

Index type	Query (ms)			Insertion(s)
	0.1%	1%	10%	
MPB-Tree	0.015	0.655	1.919	1.210
R*-Tree	0.234	0.967	2.754	0.420

rectangle's area involved in the region queries varies from 0.1% to 10% randomly. Each point in Fig. 8b represents a total insertion time cost for a certain group dataset. The results in Fig. 8 show that MPB-Tree's query performance is superior to the R*-Tree's, however its insertion performance is inferior to the R*-Tree's.

In order to test whether the number of cores affects the query performance, we ran the same program with same datasets on a workstation computer with the configuration of two 2.80 GHz Intel(R) Xeon(R) CPUs, and 24 GB RAM. This computer effectively has eight cores. The experimental results are shown in Fig. 9. The points in Fig. 9 have the same meanings of the points in Fig. 8.

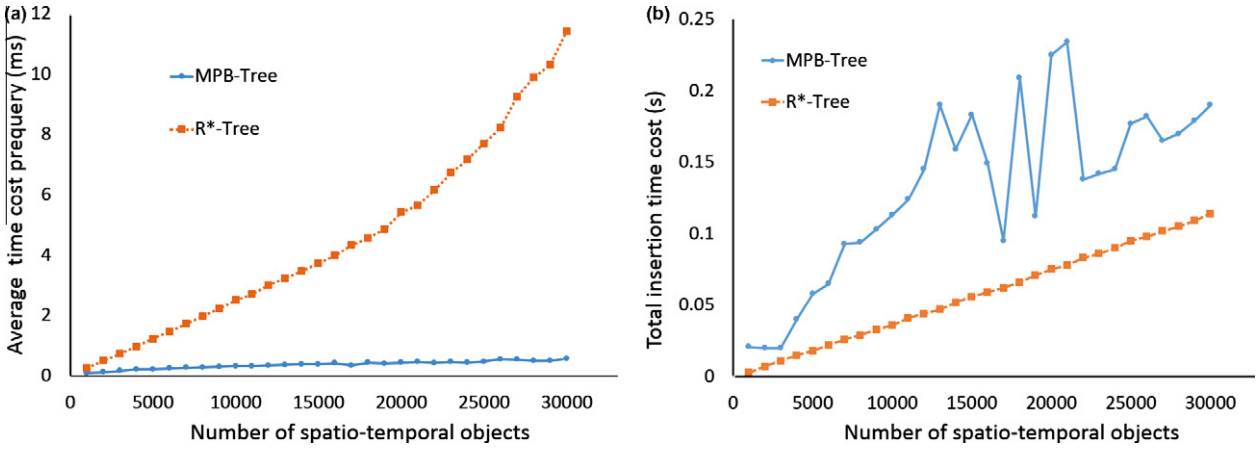

Fig. 8. Performance test computed with two cores. (a) Query. (b) Insertion.

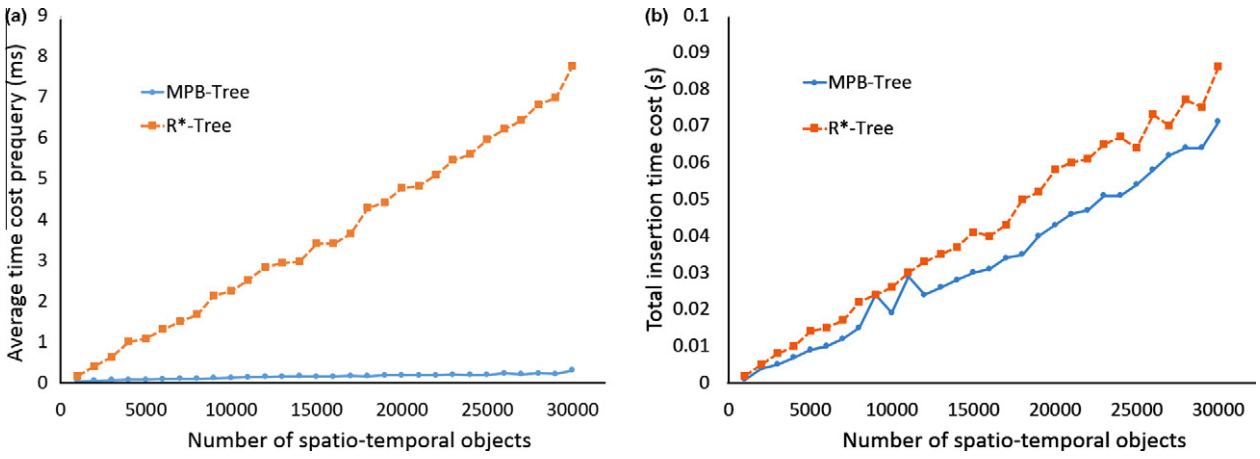

Fig. 9. Performance test computed with eight cores. (a) Query. (b) Insertion.

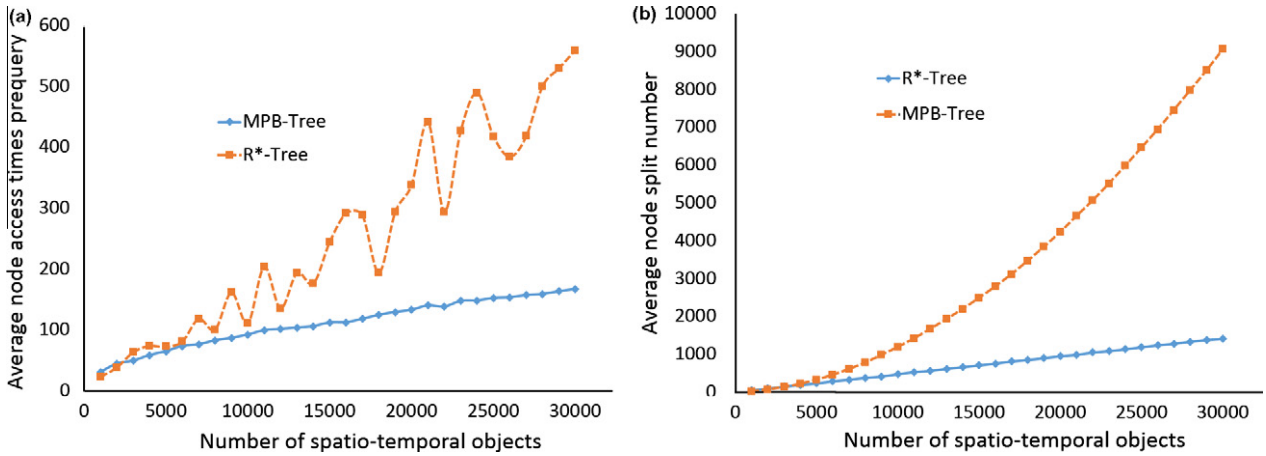

Fig. 10. Performance test measured by the account of the accesses. (a) Query. (b) Insertion.

The results shown in Fig. 9 indicate that the more parallel units a computer has, the more efficient the MPB-Tree running on the computer is. When the cores of computer are more than 4, MPB-Tree may exceed R*-Tree in the insertion performance.

Except for the absolute time information, we also carried out the test results (Fig. 10) about the account of the accesses, because such numbers are independent of the implementation and the used system or computer. Fig. 10a shows the relationship between the average node involved number of pre-query and the number of spatio-temporal objects. Each average value is calculated by 3000 queries (1000 for %0.1, 1000 for 1%, and 1000 for 10%; DS-2). It shows that the query performance of MPB-Tree is superior to R*-Tree no matter whether MPB-Tree runs on parallel computer or not. Fig. 10b shows the relationship between the node split number and the number of spatio-temporal objects in the process of insertion. It indicates that the MPB-Tree's insertion performance will be inferior to the R*-Tree's without parallel computation. But the insertion performance of MPB-Tree can be obviously improved by parallel computation. This can explain why the different results in Figs. 8b and 9b appeared.

Generally, the experimental results suggest that our proposed index structure, MPB-Tree, is superior to the R*-Tree in the query performance, shown in Figs. 8a and 9a. However, when the number of cores is less than 4, the insertion of MPB-Tree is inferior to the R*-Tree's, as shown in Fig. 8b. Because the MPB-Tree is parallel, its performance is also limited by the parallel performance of the CPU. When the number of the cores of CPU is larger than the dimensions of the space, the MPB-Tree insertion performance improves significantly and may exceed that of the R*-Tree, as shown in Fig. 9b.

5. Conclusion

We presented a new parallel access approach for handling general multi-dimensional spatio-temporal data, and more generally, multi-dimensional interval data. Based on a two-dimensional interval space representation of intervals, we innovatively extended it to multi-dimensional parallel space and presented the set formulas for the extended space. Through this process, spatio-temporal query can be transformed into parallel interval set operations. This can reduce multi-dimensional object relationship problems to simpler two-dimensional spatial intersection problems. This proposed method is shown to have better query performance than the R*-tree. When the number of computer's cores is larger than the space dimensions, the insertion performance of this new approach is also superior to R*-tree. This approach provides a potential parallel index solution for fast data retrieval of massive three-dimensional, four-dimensional or even higher dimensional data.

For the pure point data, we must transform a 4D point into a 4D MBR with a tolerance radius. This transformation will increase the data size and incur significant costs in terms of processing time and storage space costs. Therefore, the indexing method proposed in this paper is not very suitable for managing point data. For moving point data, we can index their trajectories but not the pure points. We will develop a data partitioning method to compensate for the purely spatial partition method in this approach. Further research will focus on two aspects. One is how to improve the insertion performance and the parallelism of the index methods when the number of computer's cores is less than space dimensions. The other is how to modify this approach to be suitable for point data.

Acknowledgments

The work described in this paper was supported by STAMP research project (Faculty of Geoinformation Science and Earth Observation, University of Twente), National Natural Science Foundation of China (41101368) and National High Technology Research and Development Program of China (2012AA121401).

References

Allen, J.F., 1983. Maintaining knowledge about temporal intervals. Communications of the ACM 26 (11), 832–843.
Ang, C.H., Tan, K.P., 1995. The interval B-tree. Information Processing Letters 53 (2), 85–89.
Asanovic, K., et al., 2006. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183, University of California at Berkeley.
Balasubramanian, L., Sugumaran, M., 2012. A state-of-art in R-tree variants for spatial indexing. International Journal of Computer Applications 42 (20), 35–41.
Berchtold, S., Keim, D.A., Kriegel, H.P., 1996. The X-tree: an ender structure for high-dimensional data. In: Proceedings of the International Conference on Very Large Data, Bases, pp. 28–39.
Betty, S., Vassilis, J.T., 1999. Comparison of access methods for time-evolving data. ACM Computing Surveys 31 (2), 158–221.
Brochhaus, C., Enderle, J., Schlosser, A., Seidl, T., Stolze, K., 2005. Efficient interval management using object-relational database servers. Informatik – Forschung und Entwicklung 20 (3), 121–137.
Choi, Y.J., Min, J.K., Chung, C.W., 2004. A cost model for spatio-temporal queries using the TPR-tree. Journal of Systems and Software 73 (1), 101–112.
Ciaccia, P., Patella, M., Zezula, P., 1997. M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of the Twenty-Third International Conference on Very Large Databases, pp. 426–435.
Curtis, P.K., Michael, S., 1991. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. SIGMOD Record 20 (2), 138–147.

Egenhofer, M., Litwin, W., Schek, H.-J., 1989. A formal definition of binary topological relationships, FODO 1989: foundations of data organization and algorithms. Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 457–472.

Eo, S.H., Pandey, S., Kim, M.K., Oh, Y.H., Bae, H.Y., 2006. FDSI-tree: a fully distributed spatial index tree for efficient & power-aware range queries in sensor networks. In: Sofsem 2006: Theory and Practice of Computer Science, Proceedings. Lecture Notes in Computer Science, pp. 254–261.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. SIGMOD Record 14 (2), 47–57.

Ibrahim, K., Christos, F., 1993. On packing R-trees. In: Proceedings of the Second International Conference on Information and Knowledge Management. ACM, Washington, DC, United States.

Jensen, C. et al., 2009. Indexing the trajectories of moving objects in symbolic indoor space, advances in spatial and temporal databases. Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 208–227.

Kim, J., Im, S., Kang, S.-W., Hwang, C.-S., Lee, S., 2007. SQR-tree: a spatial index using semi-quantized MBR compression scheme in R-tree. Journal of Information Science and Engineering 23 (5), 1541–1563.

Kim, S.W., Jang, M.H., Lim, S., 2010. Active adjustment: an effective method for keeping the TPR*-tree compact. Journal of Information Science and Engineering 26 (5), 1583–1600.

Knysh, D., Kureichik, V., 2010. Parallel genetic algorithms: a survey and problem state of the art. Journal of Computer and Systems Sciences International 49 (4), 579–589.

Kriegel, H.-P. et al., 2001. Object-relational indexing for general interval relationships, advances in spatial and temporal databases. Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 522–542.

Kumar, A., Tsotras, V.J., Faloutsos, C., 1998. Designing access methods for bitemporal databases. IEEE Transactions on Knowledge and Data Engineering 10 (1), 1–20.

Lin, H.-Y., 2008. Efficient and compact indexing structure for processing of spatial queries in line-based databases. Data & Knowledge Engineering 64 (1), 365–380.

Lin, H.Y., 2012. Using compressed index structures for processing moving objects in large spatio-temporal databases. Journal of Systems and Software 85 (1), 167–177.

Lin, B., Su, J.W., 2004. On bulk loading TPR-tree. In: 2004 IEEE International Conference on Mobile Data Management, pp. 114–124.

Ni, J.F., Ravishankar, C.V., 2007. Indexing spatio-temporal trajectories with efficient polynomial approximations. IEEE Transactions on Knowledge and Data Engineering 19 (5), 663–678.

Nievergelt, J., Hans, H., Kenneth, C.S., 1984. The grid file: an adaptable, symmetric multikey file structure. ACM Transactions on Database Systems 9 (1), 38–71.

Norbert, B., Hans-Peter, K., Ralf, S., Bernhard, S., 1990. The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD Record 19 (2), 322–331.

Ramaswamy, S., Afrati, F., Kolaitis, P., 1997. Efficient indexing for constraint and temporal databases. In: Database Theory—ICDT '97. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, pp. 419–431.

Saltenis, S., Jensen, C.S., 2002. Indexing of now-relative spatio-bitemporal data. VLDB Journal 11 (1), 1–16.

Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A., 2000. Indexing the positions of continuously moving objects. SIGMOD Record 29 (2), 331–342.

Stantic, B., Topor, R., Terry, J., Sattar, A., 2010. Advanced indexing technique for temporal data. Computer Science and Information Systems 7 (4), 679–703.

Tao, Y., et al., 2003. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: Proceedings 2003 VLDB Conference, Morgan Kaufmann, San Francisco, pp. 790–801.

Zhu, Q., Gong, J., Zhang, Y., 2007. An efficient 3D R-tree spatial index method for virtual geographic environments. ISPRS Journal of Photogrammetry and Remote Sensing 62 (3), 217–224.