# HTPR*-Tree: An Efficient Index for Moving Objects to Support Predictive Query and Partial History Query

Ying Fang, Jiaheng Cao, Junzhou Wang, Yuwei Peng, and Wei Song

School of Computer, Wuhan University, Wuhan, China
{fangying,jhcao,ywpeng,songwei}@whu.edu.cn

**Abstract.** Developing efficient index structures is an important issue for moving object database. Currently, most indexing methods of moving objects are focused on the past position, or on the present and future one. In this paper, we propose a novel indexing method, called HTPR*-tree (History Time-Parameterized R-tree), which not only supports predictive queries but also partial history ones involved from the most recent update instant of each object to the last update time. Based on the TPR*-tree, our HTPR*-tree adds creation or update time of moving objects to leaf node entries. This index is the foundation of indexing the past, present and future positions of moving objects. In order to improve the update performance, we present a bottom-up update strategy for the HTPR*-tree by supplementing three auxiliary structures which include hash index, bit vector, and direct access table. Experimental results show that the update performance of the HTPR*-tree is better than that of the TD_HTPR*- and TPR*-tree. Moreover, the HTPR*-tree can support partial history queries compared with TPR*-tree although the predictive query performance is a bit less.

**Keywords:** moving object indexing, HTPR*-tree, predictive query, partial history query, bottom-up update strategy.

## 1 Introduction

Developing efficient index structures is an important research issue for moving object database. Traditional spatial index structures are not appropriate for indexing moving objects because the constantly changing locations of objects requires constant updates to the index structures and thus greatly degrades their performance.

Some index structures have been proposed for moving objects. They can be classified into two major categories depending on whether they deal with past information retrieval or future prediction. Currently, some indices suitable for history and future information retrieval of moving objects have also been studied. However, they are too complicated and could not efficiently handle queries involved from the past to the future.

In general, indexing about past trajectories of moving objects only stores history information from some past time until the time of the most recent position sample ($t^{o}_{mru}$) of each object $o$. However, indexing of the current and anticipated future positions can only supports the query from the last update time ($t_{lu}$=$max( t^{o}_{mru}|o\in O)$) to the future. In other words, the history trajectories of moving objects from the most recent update instant to the last update time (or the current time CT) are omited. In order to support the queries involved from the past to the future in moving object databases, current and future positions indexing structure such as TPR-tree [1] and TPR*-tree [2] should be extended to support partial history queries.

The TPR*-tree is the most useful indexing method which indices the current and future position of moving object through Time-Parameterized Rectangles. In the TPR*-tree, partial history trajectories of moving objects which do not update at the last update time are implicit, but they couldn't be queried. In order to query history trajectories in the TPR*-tree, in this paper, we develop a novel index structure which not only supports predictive queries, but also supports partial history queries involved from the most recent update instant of each object to the last update time. This novel index structure is very important to support the queries involved from the past to the future. In order to support frequent update, bottom-up update strategy is also applied to the new index structure.

The main contributions of this paper can be summarized as:

1. We present an new index structure, named History Time-Parameterized R-tree (HTPR*-tree), which takes into account moving object creation time or update time in the leaf node entry, and supports partial history query.
2. We propose a bottom-up update approach referencing the R-tree update technique[3] to support frequent update operation of the HTPR*-tree.
3. We prove through extensive experiments that the update performance of the HTPR*-tree is better than that of the TD_HTPR*- and TPR*-tree.

The organization of this paper is as follows. Section 2 presents related works. Section 3 shows the basic structure of the HTPR*-tree, the corresponding dynamic maintenance, and query algorithms. In section 4, we discuss the bottom-up update algorithm of the HTPR*-tree. Section 5 contains an extensive experimental evaluation, and section 6 is the conclusion of our work.

## 2 Related Works

A number of index structures have been proposed for moving object database. Most of these index structures are classified into two categories; one of them is to handle past positions or trajectories [4-6], and the other is to handle current and future positions [1,2,7-12]. In addition, some indices suitable for history and future queries of moving objects have also been studied [13,14].

History trajectories indices such as STR-tree and TB-tree [4] are used to index positions for an object only up to the time of the most recent sample. They can play an important role in supporting the queries involved from the past to the future.

The search of indexing methods for current and future positions of moving objects are very challenging. In general, there are three approaches to study the indices for predictive queries. One is indexing the future trajectories of objects moved in $d$-dimensional space as lines in $(d+1)$-dimensional space [7]. Another is mapping the trajectories to points in a higher-dimensional space which are then indexed [8]. The third is to index the original time-space with parametric rectangles [1,2,9].

By introducing parametric bounding rectangles in R-tree, the TPR-tree provides the capability to answer the queries about current positions and future positions. The TPR*-Tree improved upon the TPR-Tree by introducing a new set of penalty functions based on a revised query cost model. In recent years, based on the $B^+$-tree, indices for moving objects not only supporting queries efficiently but also supporting frequent updates are proposed. Jensen et al. proposed the $B^x$-tree [10], which employs space partitioning and data/query transformations to index object positions and has good update performance. Chen et al. also proposed the $ST^2B$-tree[11], a Self-Tunable Spatio-Temporal B+-Tree index for moving object database, which is amenable to tuning.

Pelanis et al. proposed $R^{PPF}$-tree [13] to support both the past and the future movement of the objects. The implemented query types are only timestamp ones. Raptopoulou [14] et al. extended the XBR-tree to deal with future prediction as well. But it only can support timestamp queries involved from the past to the future and history window queries.

Our work aims to extend the current and future positions indexing structure to support partial history queries. This new index structure is the foundation of indexing the past, present and future positions of moving objects. In addition, bottom-up update strategy is applied to the new index structure in order to support frequent update.

## 3 The HTPR*-Tree

In this section, we will discuss a novel index structure called History TPR*-tree (HTPR*-tree). First, we describe the basic structure of the HTPR*-tree. Then, the insertion and deletion algorithms are shown. The query algorithm is given at the end.

### 3.1 Index Structure

The HTPR*-tree is a height-balanced tree similar to a R-tree. Leaf nodes of the HTPR*-Tree contain entries of the form (*oid, MBR, VBR, st*). Here *oid* is the identifier of the moving object, and *st* is creation or update time of object. *MBR* denotes object extent at time *st,* and *VBR* denotes velocity bounding rectangle of object at time *st.*

For example, a two-dimensional moving object is represented with *MBR* $o_R=\{o_{R1-}, o_{R1+}, o_{R2-}, o_{R2+}\}$ where $o_{Ri-}$ ($o_{Ri+}$) describes the lower (upper) boundary of $o_R$ along the $i$-th dimension ($1 \le i \le 2$), and *VBR* $o_V=\{o_{V1-}, o_{V1+}, o_{V2-}, o_{V2+}\}$ where $o_{Vi-}$ ($o_{Vi+}$) describes the velocity of the lower (upper) boundary of $o_R$ along the $i$-th dimension ($1 \le i \le 2$). Figure 1 shows the MBRs and VBRs of 4 objects $a,b,c,d$. The arrows (numbers) denote the directions (values) of their velocities. The *MBR* and *VBR* of $b$ are $b_R=\{3,4,4,5\}$ and $b_V=\{1,1,1,1\}$, respectively.

The structure of each non-leaf node entry of $R_s$ is in the form of *(ptr, MBR, VBR, st1, st2).* Here *ptr* is a pointer that points to the child node. *St1* is the minimal creation or update time of moving objects included in the child node pointed by *ptr,* and *st2* is the maximum value compare to *st1*. *MBR* is the minimum bounding rectangle at *st1*, and *VBR* is the velocity bounding rectangle at *st1*. Figure 2 shows a leaf node *e* including four point objects $\{o_1,o_2,o_3,o_4\}$ in one-dimensional HTPR*-Tree. So, in parent node of *e,* the corresponding entry includes *MBR, VBR*, *st1*, *st2* and *ptr* that points to node *e.* Here *MBR*={3,4}, *VBR*={-0.2, 0.3}, *st1*=2 and *st2*=4.

## 3.2 Insertion and Deletion

### 1. Insertion

Because the creation or update time of moving objects is included in leaf node entries of HTPR*-Tree, and non-leaf node entry is different from leaf node entry, the insertion algorithm of HTPR*-Tree is a bit more complicated than that of TPR*-Tree.

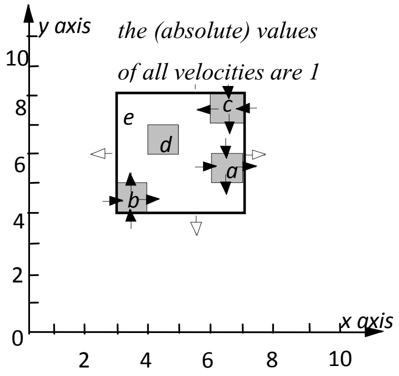


**Fig. 1.** MBRs and VBRs at reference time

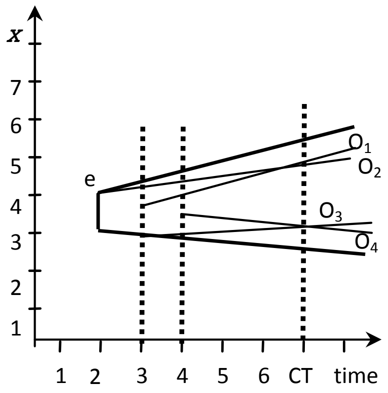


**Fig. 2.** A leaf node *e* including four point objects

Algorithm 1 shows the insertion of a moving object in the HTPR*-Tree. First, the insertion algorithm initializes two empty re-insertion lists $L_{reinsert1}$ and $L_{reinsert2}$ to accommodate re-insertion moving objects and non-leaf node entries, respectively. Then, the algorithm calls different functions according to whether the root of HTPR*-tree is a leaf node or not. Finally, the algorithm inserts each object in $L_{reinsert1}$ and each entry in $L_{reinsert2}$ to the HTPR*-tree.

Algorithm 2 describes inserting a moving object to a leaf node. However, algorithm 3 describes inserting a moving object to the HTPR*-tree rooted by a non-leaf node. Because the insertion of a node entry can cause node split, the insertion algorithm (algorithm 4) of non-leaf node entry in the HTPR*-Tree is also important.

**Algorithm 1. Insert** (*r, o*)

```
/*Input: o is a moving object with oid, MBR, VBR, st; r is the HTPR*-tree */
1.  root=Root(r)  /*achive the root of the HTPR*-tree
2.  re-inserted_i=false for all levels 1≤i≤h−1 (h is the tree height)
3.  initialize two empty re-insertion list L_reinsert1 and L_reinsert2
4.  if root is leaf node  invoke Leaf Node Insert (root, o)
5.  else                  invoke Non-Leaf Node Insert (root, o)
6.  for each data o' in the L_reinsert1
7.      if root is leaf node  invoke Leaf Node Insert (root, o')
8.      else                  invoke Non-Leaf Node Insert (root, o')
9.  for each entry e in the L_reinsert2
10.     invoke Non-Leaf Node Insert_e (root, e)
End Insert
```

**Algorithm 2. Leaf Node Insert** (*N*, *o*)

```
/* Input: N is the leaf node where object o is inserted */
1.  enter the information of o
2.  if N overflows
3.      if re-inserted_0=false   //no re-insertion at leaf level yet
4.          invoke Pick Data Worst to select a set S_worst of objects
5.          remove objects in S_worst from N; add them to L_reinsert1
6.          re-inserted_0=true
7.      else
8.          invoke Leaf Node Split to split N into itself and N'
9.          obtain entry e describe node N'
10.         invoke Non-Leaf Node Insert_e (P,e)   /*P be the parent of N*/
11. adjust the MBR/VBR/st1/st2 of the node N
End Leaf Node Insert (N, o)
```

**Algorithm 3. Non-Leaf Node Insert** (*N*, *o*)

```
/* Input: N is the root node of tree rooted by N */
1.  obtain the son node N' of N to insert o through the path achieve by Choose Data
Path
2.  if N' is the leaf node  invoke Leaf Node Insert (N', o)
3.  else                    invoke Non-Leaf Node Insert (N', o)
4.  adjust the MBR/VBR/st1/st2 of the node N
End Non-Leaf Node Insert(N, o)
```

**Algorithm 4. Non-Leaf Node Insert_e** (*N*, *e*)

```
/* Input: e is non-leaf node entry
1.  if e.level=N.level
2.      enter e to N
3.      if N overflows
4.          if re-inserted_i=false   //no re-insertion at i level (e.level)yet
```

```
5.          invoke **Pick Entry Worst** to select a set $S_{worst}$ of entrys
6.          remove entrys in $S_{worst}$ from $N$; add them to $L_{reinsert2}$
7.          $re\text{-}inserted_i$=true
8.        else
9.          invoke **Non-Leaf Node Split** to split $N$ into itself and $N'$
10.         obtain entry $e$ describe node $N'$
11.         invoke **Non-Leaf Node Insert_e** ($P$, $e$)    /*$P$ be the parent of $N$*/
12.   else
13.       obtain the son node $N'$ of $N$ through the path achieve by **Choose Entry Path**
14.       invoke **Non-Leaf Node Insert_e** ($N'$, $e$)
15.   adjust the MBR/VBR/st1/st2 of the node $N$
**End Non-Leaf Node Insert_e**($N$, $e$)
```

Similiar to that in the TPR*-tree, algorithm *Choose Path* in the HTPR*-tree aims at finding the best insertion path globally with a minimum cost increment (minimal increase in equation 1). If a moving object is inserted, *Choose Path* is instantiated by *Choose Data Path*, and non-leaf node entry inserting calls *Choose Entry Path.* Because the creation or update time of moving objects is included in leaf node entries, the enlarge of entry (caused by inserting a moving object in HTPR*-tree node) involves history information, and the enlarged entry can support history query. This is the major difference between HTPR*-tree and TPR*-tree. Of course, the predictive query performance is somehow less than that of TPR*-tree.

The query cost model of the HTPR*-tree is the average number of node accesses for answering query $q$:

$$Cost(q)=\sum_{\text{every node } N} A_{SR}(N', q_T) \tag{1}$$

where $N$ is the moving rectangle (interval as for one-dimensional object) representing a node, $N'$ is the transformed rectangle (interval as for one-dimensional object) of $N$ with respect to $q$, and $A_{SR}(N',q_T)$ is the extent of region swept by $N'$ during $q_T$.

Figure 3 shows two leaf nodes *e1* and *e2* that are sons of the root *e0*. Here the entry corresponding to *e1* is {*pt1,*{3.2,4},{-0.2,0.3},2,3}, and the entry to *e2* is {*pt2,*{5,6},{0,0.4},3,4}. Consider the insertion of point object $O_6$={ $O_6$,4.6,0.5,7} at current time 7. *Choose Data Path* returns the insertion path with the minimum increment in equation 1. The cost increment is 1.2 and 0.9 when *o6* is inserted to *e1* and *e2,* respectively. Figure 4 describes insertion *o6* to *e2*, which is the best insertion node.

Insertion to a full node *N* generates an overflow, in which the HTPR*-tree uses *Pick Worst* algorithm that selects a fraction of the entries from the node *N* and re-inserts them. *Node Split* algorithm splits a full node *N* into *N1* and *N2*. The split algorithm selects split axis and split position minimizing equation 2:

$$\Delta A_{SR}=A_{SR}(N_1',q_T)+A_{SR}(N_2',q_T)-A_{SR}\,(N',q_T) \tag{2}$$

Another important difference between the insertion of HTPR*-tree and that of TPR*-tree is that the *MBR*, *VBR*, *st1* and *st2* of node *N* after inserting have to be modified.

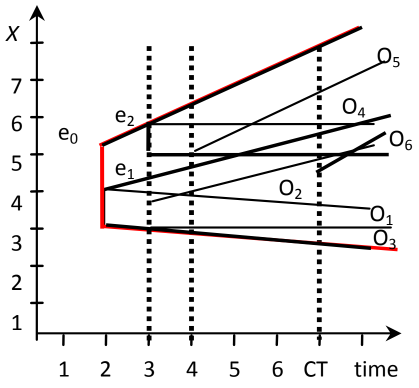


Fig. 3. Insert a moving point object *o6*

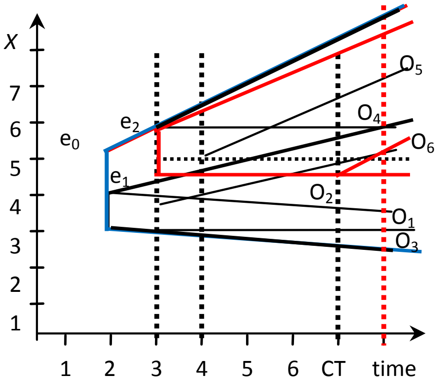


Fig. 4. Insert *o6* to node e2

**2. Deletion**

Algorithm 5 describes deleting a moving object in HTPR*-Tree. To remove an object *o,* the deletion algorithm first identifies the leaf node that contains *o*. In algorithm 5, two empty re-insertion lists $L_{reinsert3}$ and $L_{reinsert4}$ are initialized to accommodate re-insertion leaf node entries (moving objects) and non-leaf node entries, respectively.

```
Algorithm 5. Delete (r, o)
/*Input: o is a moving object with oid, MBR, VBR, st; r is the HTPR*-tree */
1.  root=Root(r)   /*achive the root of the HTPR*-tree
2.  initialize an empty re-insertion list $L_{reinsert3}$ and $L_{reinsert4}$
3.  if root is leaf node     invoke Leaf Node Delete (root, o)
4.  else                     invoke Non-Leaf Node Delete (root, o)
5.  for each data o' in the $L_{reinsert3}$
6.          invoke Non-Leaf Node Insert (root, o')
7.  for each entry e in the $L_{reinsert4}$
8.          invoke Non-Leaf Node Insert_e (root, e)
End Delete (r, o)
```

Deletion of moving object in leaf node *N* may generate an underflow, in which case the HTPR*-tree removes all objects in node *N* to $L_{reinsert3}$, and deletes entry *e* describes *N* in parent node *N'*. If moving object *o* is deleted in the HTPR*-tree root by non-leaf node *N*, the deletion algorithm calls *Leaf Node Delete* or *Non-Leaf Node Delete* in all son node *N'* of *N* until *o* is deleted. In algorithm *Leaf Node Delete* or *Non-Leaf Node Delete,* if deletion of moving object *o* in the HTPR*-tree root by node *N* changes node *N*, adjustment is needed from parent node *N'* of *N* to root.

## 3.3 Search Procedure

The HTPR*-tree supports three kinds of predictive queries: timeslice query *Q=(R, t)*, window query *Q=(R, $t_1$, $t_2$)*, and moving query *Q=($R_1$, $R_2$, $t_1$, $t_2$)*. At the same time, the HTPR*-tree supports partial history query. Figure 5 describes timeslice query and spatio-temporal range query of moving objects. The dashed parts of objects trajectories are stored in the HTPR*-tree, and support predictive queries and partial history queries.

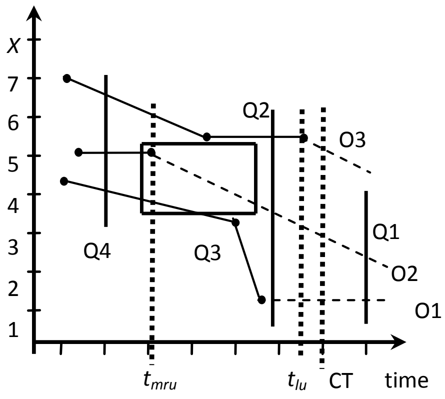


**Fig. 5.** Querying the Positions of Moving Objects

For example, query *Q1* is predictive timeslice query, and gets object *O1* and *O2*. Query *Q2*, *Q3,* and *Q4* are history queries. Query *Q2* intersects with partial history trajectories after the most recent update instant ($t^o_{mru}$) of objects *O1* and *O2* stored in HTPR*-tree. However, HTPR*-tree couldn't realize query *Q2* completely because the query time is earlier than the last update time ($t_{lu}$). In order to realize the queries involved from the past to the future, HTPR*-tree should be combined with some indices for describing history trajectories such as TB-tree.

Algorithm 6 is an illustration of spatio-temporal range query in the HTPR*-Tree. In algorithm *RQuery*(r, w, $T_1$, $T_2$), if *st1* of root is larger than $T_2$, query is unsuccessful. Else query calls *a*lgorithm *LeafNodeRQuery* (root is leaf node) or *nonLeafNodeRQuery* (root is non-leaf node). If $T_1$ is larger than *st2* of root and $T_2$ is small than or equal to the current time, algorithm *RQuery* can realize history range query completely, which is very similar to that find in predictive range query. Algorithm *IN*(*o*, w, $T_1$, $T_2$) determines whether object *o* is located in range w from time $T_1$ to $T_2$.

**Algorithm 6. RQuery**(r, w, $T_1$, $T_2$)

```
1. root=Root(r)  /*achive the root of the HTPR*-tree
2. get st1 and st2 of root
3. if  T2<st1 return null
4. else if root is leaf node   invoke LeafNodeRQuery (root, w, T1, T2)
5.     else                    invoke nonLeafNodeRQuery (root, w, T1, T2)
   ENDRQuery
```

```
**Algorithm 7. Leaf NodeRQuery** (N, w, T1, T2)
1. for each o of N
2.     if IN(o, w, T1, T2)   return o
ENDLeaf Node RQuery
```

```
**Algorithm 8. nonLeafNodeRQuery** (N, w, T1, T2)
1. for each son node N' of N
2.   if N' is leafnode     Leaf NodeRQuery (N', w, T1, T2)
3.   else                  nonLeafNodeRQuery (N', w, T1, T2)
ENDnonLeaf Node RQuery
```

## 4 Bottom-Up Update

It is well known that the update efficiency of TPR*-tree is not very high since it is worked in a top-down manner. This is also the case for the top-down update HTPR*-tree. In order to support frequent update, bottom-up update strategy is adopted by the HTPR*-tree.

To support bottom-up update strategy, the HTPR*-Tree supplements auxiliary structures which include hash table, and compact main memory summary structures such as bit vector and direct access table. Figure 6 shows auxiliary structures for HTPR*-Tree.

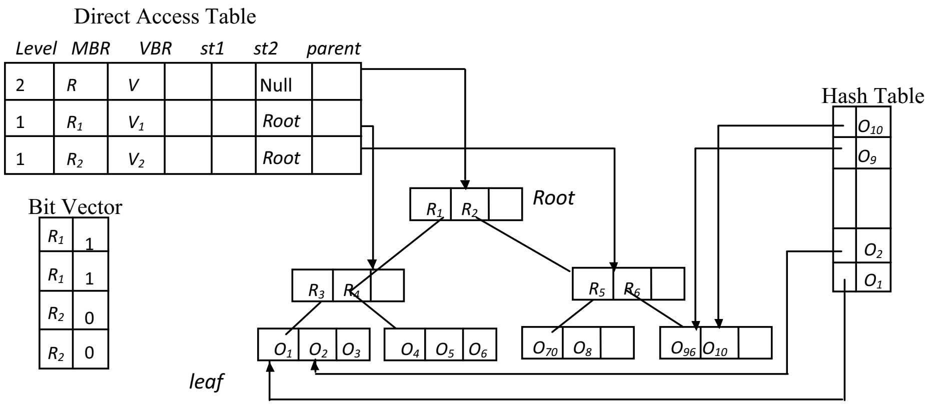


**Fig. 6.** Auxiliary structures for the HTPR*-Tree to support bottom-up update

Hash table allows us to locate the leaf node where the updated object *o* resides in one disk I/O instead of doing the expensive query on the tree.

Direct access table facilitates quick access to a node's parent in the HTPR*-Tree. An entry of the direct access table corresponds to a non-leaf node of the HTPR*-Tree, and all the entries are organized according to the levels of the internal nodes they correspond to. An entry in the direct access table is a 7 tuple of the form < *Level, MBR, VBR, st1, st2, parentptr, ptr*>, where *Level* is the level of the node, *MBR* and *VBR* is the bounding box and the velocity bounding rectangle of the node at time *st,* respectively, *parentptr* is a pointer to the node' parent, *ptr* is a pointer to the node itself, *st1* is the minimal creation or update time of moving objects included in node, and *st2* is the maximum value compare to *st1.* Bit vector on the leaf nodes indicates whether they are full or not, which avoids additional disk accesses to find a suitable sibling.

The maintenance cost for the main-memory summary structure is relatively inexpensive. Since most of the node splits occur in the leaf level due to the high node fan-out, inserting a new entry into the direct access table will be very infrequent. Meanwhile, only when the leaf node is split or deleted, a new entry need be inserted into bit vector or be deleted.

The size of each entry in the direct access table is a small fraction of the size of the corresponding HTPR*-Tree node. And the size of bit vector is much smaller than that of direct access table, and even neglectable. Overall, the space consumption of the main-memory summary structure is very low relative to HTPR*-Tree.

The bottom-up strategy aims to offer a comprehensive solution to support frequent updates in the HTPR*-Tree. The main idea of bottom-up update algorithm is described as follows: when an object issues an update request, the algorithm adopts different update method according to object position and velocity after updating, and update time. The detailed update method is as follows:

1. If new position lies outside MBR of root or new velocity lies outside VBR of root, the algorithm issues a top-down update.

2. If new position and velocity of moving object lie in the MBR and VBR of current leaf node, the algorithm modifies the object entry in leaf node directly. At the same time, the algorithm constructs update path from leaf to root using the direct access table and tightens all nodes on that path.

3. If new position and velocity of moving object lie outside the MBR and VBR of current leaf node, and the removal of moving object causes leaf node to underflow, the algorithm issues a top-down update.

4. If new position and velocity of moving object lie in the MBR and VBR of non-null sibling node, and the removal of moving object couldn't cause leaf node to underflow, the algorithm deletes old entry and inserts new entry in right sibling node. At the same time, the algorithm constructs update path from leaf to root using the direct access table and tightens all nodes on that path.

5. If the new position and velocity of moving object lie in the MBR and VBR of a subtree (intermediate node), the algorithm ascends the HTPR*-tree branches to find a local subtree and performs a standard top-down update.

Algorithm 9 describes bottom-up strategy of the HTPR*-Tree.

**Algorithm 9. Update** (*o, o',r,DAT,BV,H)*

```
1.  entry=DAT[0]
2.  if $o'.MBR \not\subset$ entry.$MBR$ or $o'.VBR \not\subset$ entry.$VBR$
3.      TD_Update($o, o',r,DAT,BV,H$)
4.      return
5.  leaf=$H$.getLeafNode($o$)
6.  if $o'.MBR \subset$ leaf.$MBR$ and $o'.VBR \subset$ leaf.$VBR$
7.      write out leaf
8.      non-leaf=getParent(leaf)
9.      if non-leaf is not tigten
10.         tigten non-leaf
11.         non-leaf'=$DAT$.GetParent(non-leaf)
12.         while (non-leaf' is not tigten and non-leaf' is not null)
13.             tigten non-leaf' , non-leaf =non-leaf'
14.             non-leaf'=$DAT$.GetParent(non-leaf)
15.     return
16. if leaf.numEntry=m      TD_Update($o, o',r,DAT,BV,H$)
17.     return
18. leaf.delete($o$)
19. construct update-path from $leaf$ to root using the $DAT$, and tighten all nodes on
      that path similar to lines 8-14
20. {$siblings$}=$BV$.getsiblings($leaf$)
21. for $S_i \in$ {siblings}
22.     if $o'.MBR \subset S_i.MBR$ and $o'.VBR \subset S_i.VBR$ and $BV$.notFull($S_i$)
23.         if $o'.st > S_i.st_2$
24.             $s_i$.insert($o'$ )
25.             construct update-path from $s_i$ to root using the $DAT$, and change $st_2$ of
              all nodes on that   path into $st$ similar to lines 8-14
26.         else   $s_i$.insert($o'$ )
27.     return
28. $non\text{-}leaf$ =getParent($leaf$)
29. while ($o'.MBR \not\subset non\text{-}leaf.MBR$ or $o'.VBR \not\subset non\text{-}leaf.VBR$)
30.         $non\text{-}leaf$ =$DAT$. GetParent($non\text{-}leaf$)
31. TD_Update($o, o', non\text{-}leaf, DAT,BV,H$)
32. construct update-path from $non\text{-}leaf$ to root using the $DAT$, and tighten all
    nodes on that path similar to lines 8-14
33. return
END  Update
```

## 5 Performance Study

### 5.1 Experimental Setting and Details

In this section, we evaluate the query and update performance of the HTPR*-tree with the TPR*- and TD_HTPR*-tree (top-down update strategy). Due to the lack of real datasets, we use synthetic data simulating moving aircrafts like[2]. First 5000 rectangles are sampled from a real spatial dataset (LA/LB)[15] and their centroids serve as the “airports”. At timestamp 0, 100k aircrafts (point objects) are generated such that for each aircraft *o*, (i) its location is at one (random) airport, (ii) it (randomly) chooses a destination airport, and (iii) its velocity value *o.Vel* uniformly distributes in [20,50], and (iv) the velocity direction is decided by the source and destination airports. For each dataset, we construct a HTPR*- and TPR*-tree, whose horizons are fixed to 50, by first inserting 100k aircrafts at timestamp 0.

Since the HTPR*-tree only stores history trajectories after the most recent update of each object, and history trajectories index such as TB-tree only store trajectories before the most recent update instant, it is improper to compare the history query performance of HTPR*-tree with that of history trajectories index. In our experiments, we only study the predictive queries. However, HTPR*-tree can be combined with history trajectories index to support the queries involved from the past to the future. This will be our future work.

## 5.2 Performance Analysis

In order to study the deterioration of the indices with time, we measure the performance of the HTPR*-, TPR*- and TD_HTPR*-tree, using the same query workload, after every 10k updates.

**• Update Cost Comparison**

Figure 7 compares the average update cost (for datasets generated from LA and LB as mentioned above) as a function of the number of updates. The HTPR*- and TPR*-tree have nearly constant update cost. However the node accesses needed in the HTPR*-tree update operation are much less than the TPR*- and TD_HTPR*-tree. This is due to the fact that the HTPR*-tree adopts bottom-up update strategy to avoid the excessive node accesses for top-down deletion search and insertion search, and the TPR*- and TD_HTPR*-tree process update in top-down manner. Since node overlap in the TD_HTPR*-tree is larger than that in the TPR*-tree, the query cost increasing with the number of updates improves the update cost of TD_HTPR*-tree.

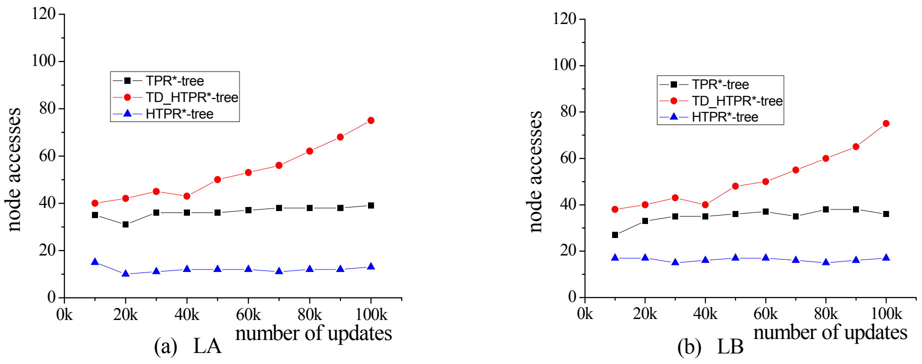
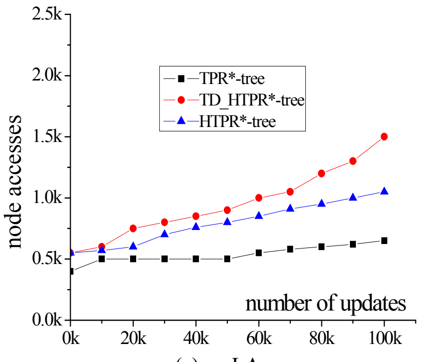


**Fig. 7.** Update cost comparison

- **Query Cost Comparison**

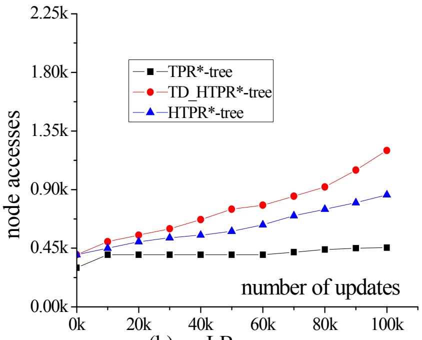
The query cost is measured again as the average number of node accesses in executing 200 predicted window queries with the same parameters $q_R len$, $q_V len$, $q_T len$.

In Figure 8, we plot the query cost as a function of the number of updates, using workloads with different parameter: for Figure 8 (a) we fix parameters $q_T len$=50, $q_R len$=400 and $q_V len$=10, while the parameters $q_T len$=50, $q_R len$=1000 and $q_V len$=10 are fixed in Figure 8 (b).



(a) LA

$q_R len$=400， $q_V len$=10， $q_T len$=50



(b) LB

$q_R len$=400， $q_V len$=0， $q_T len$=50

**Fig. 8.** Query cost comparison

It is clear that the query cost increases with the number of updates. The query cost of the HTPR*-tree is less than that of the TD_HTPR*-tree. Since the node overlap in the HTPR*-tree is larger than that in the TPR*-tree, the query cost of the HTPR*-tree is a bit higher than that of the TPR*-tree. Despite this, the important fact is that HTPR*-tree can support history query.

## 6 Conclusion

In this paper, we develop a novel index structure named the HTPR*-tree which not only supports predictive queries but also partial history ones. At the same time, we propose a bottom-up update approach to support frequent update operation of the HTPR*-tree. Extensive experiments prove that the update performance of the HTPR*-tree is better than that of the TD_HTPR*- and TPR*-tree. Moreover, the HTPR*-tree can support history query compared with TPR*-tree although the predictive query performance is a bit less.

For the future work, we will combine the HTPR*-tree with history trajectory indices such as TB-tree to implement historical and future information retrieval.

**Acknowledgments.** This work is supported by the National Natural Science Foundation of China (Grant No.90718027) and the Natural Science Foundation of Hubei Province (Grant No.2008CDA007).

## References

[1] Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the Positions of Continuously Moving Objects. In: ACM SIGMOD, pp. 331–342 (2000)

[2] Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized spatiotemporal Access Method for Predictive Queries. In: VLDB, pp. 790–801 (2003)

[3] Lee, M., Hsu, W., Jensen, C., et al.: Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In: VLDB, pp. 608–619 (2003)

[4] Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel Approaches to the Indexing of Moving Object Trajectories. In: VLDB, pp. 395–406 (2000)

[5] Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-temporal Indexing for Large Multimedia Applications. In: Conf. on Multimedia Computing and Systems, pp. 441–448 (1996)

[6] Nascimento, M.A., Silva, J.R.O.: Towards Historical R-trees. In: Proc. of the ACM Symposium on Applied Computing, pp. 235–240 (1998)

[7] Tayeb, J., Ulusoy, O., Wolfson, O.: A Quadtree-Based Dynamic Attribute Indexing Method. The Computer Journal 41(3), 185–200 (1998)

[8] Jignesh, M., Yun, P., Chen, V., Chakka, P.: TRIPES: An Efficient Index for Predicted Trajectories. In: ACM SIGMOD, pp. 637–646 (2004)

[9] Liao, W., Tang, G.F., Jing, N., Zhong, Z.-N.: Hybrid Indexing of Moving Objects Based on Velocity Distribution. Chinese Journal of Computers 30(4), 661–671 (2007)

[10] Jensen, C.S., Lin, D., Ooi, B.C.: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In: VLDB, pp. 768–779 (2004)

[11] Chen, S., Ooi, B.C., Tan, K.L., et al.: ST$^2$B-tree: A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects. In: ACM SIGMOD, pp. 29–42 (2008)

[12] Chen, N., Shou, L.D., Chen, G., et al.: B$^s$-tree: A Self-tuning Index of Moving Objects. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 1–16. Springer, Heidelberg (2010)

[13] Pelanis, M., Saltenis, S., Jensen, C.S.: Indexing the Past, Present and Anticipated Future Positions of Moving Objects. In: ACM TODS, pp. 255–298 (2006)

[14] Raptopoulou, K., Vassilakopoulos, M., Manolopoulos, Y.: Efficient processing of past-future spatiotemporal queries. In: Proc. of the ACM Symposium on Applied Computing, pp. 68–72 (2006)

[15] `http://www.census.gov/geo/www/tigers`