HGTPU-Tree: An Improved Index Supporting Similarity Query of Uncertain Moving Objects for Frequent Updates

Mengqian Zhang[1(✉)], Bohan Li[1,2,3(✉)], and Kai Wang[1]

[1] College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, China
{mengqianz, bhli}@nuaa.edu.cn, hxyrqxl23@sina.com
[2] Collaborative Innovation Center of Novel Software Technology
and Industrialization, Nanjing, China
[3] Jiangsu Easymap Geographic Information Technology Corp, Ltd,
Yangzhou, China

Abstract. Position uncertainty is one key feature of moving objects. Existing uncertain moving objects indexing technology aims to improve the efficiency of querying. However, when moving objects' positions update frequently, the existing methods encounter a high update cost. We purpose an index structure for frequent position updates: HGTPU-tree, which decreases cost caused by frequent position updates of moving objects. HGTPU-tree reduces the number of disk I/Os and update costs by using bottom-up update strategy and reducing same group moving objects updates. Furthermore we purpose moving object group partition algorithm STSG (Spatial Trajectory of Similarity Group) and uncertain moving object similar group update algorithm. Experiments show that HGTPU-tree reduces memory cost and increases system stability compared to existing bottom-up indexes. We compared HGTPU-tree with TPU-tree, GTPU-tree and TPU^{2}M-tree. Results prove that HGTPU-tree is superior to other three state-of-the-art index structures in update cost.

Keywords: Position uncertainty · Moving objects · HGTPU-tree · Group partition · Update cost

1 Introduction

In the era of mobile computing, effective management of moving objects is a guarantee of high-quality location services. Due to the inaccurate data collection, the delayed updating of moving objects and privacy protection, position uncertainty of moving objects is widespread [1]. Since the position of moving object changes with time, the specific position of the storage space object in the traditional spatial index structure cannot adapt to the updating operation of a large number of spatial objects. Thus it is not suitable for storage and retrieval of moving object [2].

In order to obtain more accurate query results, moving object position information needs to be updated frequently [3]. The existing position update strategies are mainly divided into the following two types: 1. Periodic update: updating the position

J. Li et al. (Eds.): ADMA 2019, LNAI 11888, pp. 135–150, 2019.
https://doi.org/10.1007/978-3-030-35231-8_10

information of the moving object every n cycles; 2. Speculative positioning update [4]: the position information of the moving object is updated as long as the actual position of the moving object and the position recorded in the database exceed a certain threshold. However, the above strategies are focused on managing the position of a single moving object, instead of the relationship of the moving objects.

The motion trajectories of some moving object sets in the real scene often have certain similarity and regularity. If the trajectories of moving objects are similar, they can be divided into a group. For the members in the same group, because the position information of the moving objects is similar to each other, only one moving object's position needs to be updated. Real-time explicit updates of the position information of each moving object are not required. Through this update strategy, the number of updates of moving objects is reduced, thereby decreasing the update cost of moving objects. The main contributions of our work are as follows:

1. We develop an index structure HGTRU-tree that supports moving object group partition on the basis of the existing index TPU-tree supporting moving object uncertainty;
2. We propose the moving object group partition algorithm STSG by comparing and analyzing the historical trajectories of moving objects. STSG uses Spatial Trajectory of Similarity (STS) to describe the similarity of moving objects trajectories;
3. We use hash table as the primary index to support the HGTPU-tree bottom-up query. When the moving objects positions update, hash function is first used to query hash table. The group number of moving object is used to find the leaf node where the moving object is stored;
4. We propose a hybrid trajectory-dependent moving object position update strategy, which combines the update strategy of periodic update and speculative positioning update.

The remainder of the paper is organized as follows: Sect. 2 provides the related work. Our proposed HGTPU-tree is presented in Sect. 3. Section 4 proposes experimental results. Section 5 concludes the paper. Frequently used symbols are listed in Table 1.

Table 1. Symbol description.

Symbol	Explanation
th	Threshold
M_i, M_j	Moving objects
L	Label of moving objects
RD	Relative Direction
SR	Speed Ratio
SD	Spatial Distance
flag	Leaf node mark
MBR	Minimum Boundary Rectangles
ptr	Pointers to the next layer

(*continued*)

Table 1. (*continued*)

Symbol	Explanation
G_i	Group
g_id	Group mark
ptr_r	Space layer leaf pointers
ptr_g	Data layer pointers
time_update	Next position update time
oid	Mark of the moving object *Mi*
$PCR(p_i)$	Position recorded in the database
v	Velocity
pdf_ptr	Probability density distribution function
next_flag	Token of whether there is a next leaf node
n	Number of moving objects
m	Number of moving object groups
H	Height of index tree
L	Number of leaf nodes in index tree

2 Related Work

2.1 Moving Object Index

In order to solve the problem of how to efficiently manage the precise position information of moving objects in real time, a series of index structures were proposed. For example, TPR* tree [5], STAR [6] tree, and R^{EXP} tree [7] are all parameterized indexing methods that manage current and future position information. The top-down update mode of TPR* tree leads to a large I/O cost. R^{EXP} tree improves the update performance of the invalid data by adding data time valid attributes on the node. The historical position, current position and future position information are combined to propose index models such as $PPFN^x$ tree [8] and R^{PPF} tree [9]. In [10], R-tree-based bottom-up update idea is proposed. The update process starts from the leaf node of the tree, which saves the query time. However, the disadvantage lies in the maintenance of the index. [11] proposed the first SFC-based packing strategy that creates R-trees with a worst-case optimal window query I/O cost. The above index models can't deal with the problem of the frequent position updates of the moving object.

Tao [12] et al. proposed a U-tree index model. U-tree has a good dynamic structure. But U-tree is only suitable for static moving object uncertainty indexing. [13] proposed a U-tree-based TPU-tree for efficient current and future uncertain position information retrieval. TPU-tree adds a data structure for recording the uncertain state of moving objects on the U-tree structure. In [14], based on TPU-tree, an update memo (UM) memory structure for recording the state characteristics of uncertain moving objects is added. An uncertain moving object indexing strategy TPU^2M-tree supporting frequent position updates and an improved memo (MMBU/I) based update/insert algorithm are proposed. However, TPU^2M-tree needs extra memory space to store the information of the memo (UM).

The above index structures only consider one single moving object when the position of moving objects changes. The motion trajectories between moving objects are not considered. In addition, most of the uncertain moving object index structures adopt the traditional top-down. It causes a large disk I/O cost. Even if partial index structures have a bottom-up update idea, it needs to sacrifice a large amount of memory resources, resulting in low system stability.

2.2 Trajectory Similarity Calculation

The trajectory data of moving object in the environment is usually discrete. In this paper, the historical position and velocity are used to describe the moving parameters of object, and moving objects are grouped by analyzing the trajectories of moving objects by moving parameters.

The spatio-temporal coordinate of the moving object *Mi* is a quad (l, x, y, t). In the case where the labels of two moving objects *Mi* and *Mj* are known, if *Mi* and *Mj* have the same semantic label, they can be directly divided into one group. But in many cases, the semantic label of the moving object *Mi* cannot be directly obtained. In this situation, the trajectory data of the moving object needs to be analyzed. x, y, t means that the spatial coordinate of the moving object *Mi* is (x, y) at time t.

The position information of the moving object *Mi* at time *t0* is (l, *x0*, *y0*, *t0*), and after ΔT the coordinate of the time *t1* becomes (l, *x1*, *y1*, *t1*). Let $\Delta x, \Delta y$ be the change amount of motion in the direction of x, y, the moving speed be v, and the moving direction be θ:

$$v = \frac{\sqrt{\Delta x^2 + \Delta y^2}}{\Delta T} \tag{1}$$

$$\theta = \begin{cases} \varphi.\mathrm{sgn}(\Delta y)\ \Delta x > 0 \\ \pi/2.\mathrm{sgn}(\Delta y)\ \Delta x = 0 \\ (\pi - \varphi).\mathrm{sgn}(\Delta y)\ \Delta x < 0 \end{cases} \tag{2}$$

$$\Delta x = x1 - x0, \Delta y = y1 - y0, \Delta T = t1 - t0, \tan(\varphi) = |\Delta y/\Delta x|, \theta \in (-\pi, \pi)$$

For the moving characteristics of moving objects, some researches have focused on Relative Direction (RD) [15] and Speed Ratio (SR) [16]. We propose Spatial Distance (SD) and the Spatial Trajectory of Similarity (STS).

$$RD(M_i, M_j, t) = \cos\left(\theta_{M_i}(t) - \theta_{M_j}(t)\right) \tag{3}$$

The relative direction *RD* of the moving objects *Mi* and *Mj* at time t is calculated as Eq. 3, which is defined as the cosine of the angle of the velocity.

$$SR(M_i, M_j, t) = \frac{\min\left(v_{M_i}(t), v_{M_j}(t)\right)}{\max\left(v_{M_i}(t), v_{M_j}(t)\right)} \tag{4}$$

The calculation of the speed ratio *SR* of the moving objects *Mi* and *Mj* at time *t* is shown in Eq. 4, which is defined as the ratio of the minimum speed to the maximum speed. *SR* reflects the speed difference between *Mi* and *Mj*.

The *SD* of the moving objects *Mi* and *Mj* at time *t* is defined as the spatial distance difference between the two moving objects, and is calculated by Euclidean Distance. The calculation formula for *SD* is as follows:

$$SD(M_i, M_j, t) = \sqrt{\left(M_i.x(t) - M_j.x(t)\right)^2 + \left(M_i.y(t) - M_j.y(t)\right)^2} \tag{5}$$

STS describes the spatial trajectory of similarity between moving objects, and the value of *STS* depends on *SR*, *RD*, and *SD*. From the previous formula, the more consistent the velocity and direction of moving objects, the larger the value of *RD* and *SR*, and vice versa. The *STS* is positively correlated with *SR* and *RD* and is negatively correlated with *SD*, and its calculation formula is as follows:

$$STS(M_i, M_j, t) = \frac{RD(M_i, M_j, t) * SR(M_i, M_j, t)}{SD(M_i, M_j, t)} \tag{6}$$

3 HGTPU-Tree

3.1 Model

HGTPU-tree implements a bottom-up update with a zero-level index hash table and the entire index structure is divided into three layers: a space layer, a group layer, and a data layer.

Hash Table. HGTPU-tree implements bottom-up query with a hash table. When moving object performs position update, it first queries hash table to find the address of the group where moving object is stored, and then directly locates leaf node, and determines whether the updated position exceeds the MBR range of leaf node. If the range is not exceeded, the leaf node is updated directly.

The hash function takes the group number of the moving object as input. The record in the hash table contains 2 parts, one part is the output value of the hash function, and the remaining part is the address corresponding to the group number *Gi*. The hash table in HGTPU-tree ensures that the address of group object is recorded in real time by adopting a synchronous update with leaf node.

The Space Layer. The space layer describes the position of the space in which moving object is stored. The record form of the node in the space layer is <*flag*, *MBR*, *ptr*>. Since the position of the moving object changes at any time, the spatial position of the group in which the moving object is stored also changes. When the spatial distance between the moving object of the group *Gi* in the group layer and the position of the

currently recorded leaf node is larger than the threshold *th*, a position update operation is performed. The pointer of the group layer and the leaf node is disconnected, so that the group layer points to the leaf node where the current moving object is actually located.

The Group Layer. The record form of the HGPTU-tree node in the group layer is <*g_id, ptr_r*, *ptr_g*, *MBR*, *th*, *time_update*>. HGTPU-tree adopts a hybrid update strategy combining periodic update and speculative positioning update. When the position information *MBR* and the space layer node indicated by *ptr_r* in the group layer exceeds the threshold *th*, the data is updated and *ptr_r* is reassigned. In order to make the motion trajectory of moving object in the same group as consistent as possible, it is necessary to periodically update the grouping. Update strategy determines whether the trajectory of the group members can still be divided into a group by *ptr_g* in the past.

The Data Layer. The form of each HGTPU-tree leaf node in the data layer is <*oid*, *ptr*, *PCR(pi)*, *MBR*, *v*, *pdf_ptr*, *next_flag*>. The moving objects in the same group in the HGTPU-tree are continuously stored with each other. When the moving objects in the group are periodically detected, it is not only judged whether the spatial position and the speed deviation of the moving object exceed the threshold, but also needs to update the probability-restricted area of the moving object *Mi*.

3.2 Spatial Trajectory of Similarity Group

In the HGTPU-tree, moving objects with similar motion trajectories in a historical period are divided into one group, and then the moving objects in the same group are stored in the same leaf node in the HGTPU-tree. Regarding the group partition of moving object, a Spatial Trajectory of Similarity Group (STSG) algorithm is proposed. Some definitions in the STSG algorithm are as follows:

Definition 3 (directly reachable): The minimum spatial trajectory of similarity STSMin is the judgment threshold of the direct reach of the node and it is a constant. When STS (*Mi*, *Mj*, *t*)>STSMin, it is considered that *Mi* and *Mj* are directly reachable at time *t*, which is recorded as $M_i \leftrightarrow M_j$, otherwise, *Mi* and *Mj* are not directly reachable, and are recorded as $M_i \nleftrightarrow M_j$.

Definition 4 (dependency reachable): For any two nodes *Mi* and *Mj* satisfy $M_i \nleftrightarrow M_j$ but there is M_k, let $M_i \leftrightarrow M_k$ and $M_i \leftrightarrow M_k$ then *Mi* and *Mj* are dependency reachable, denoted as $M_i \simeq M_j$, otherwise *Mi* and *Mj* are not dependency reachable and is recorded as $M_i \not\simeq M_j$.

Definition 5 (connection): For any two nodes *Mi* and *Mj* satisfy $M_i \nleftrightarrow M_j$ and $M_i \not\simeq M_j$, but there is a node set S(M_1,…, M_n), $n > 1$, so that *Mi* and *Mj* can be reached by S dependence, then *Mi* and *Mj* are connected, which is denoted as $M_i \approx M_j$, otherwise *Mi* and *Mj* are not connected which is recorded as $M_i \not\approx M_j$.

Definition 6 (group): Divide moving objects with similar motion trajectories into a group denoted as g, if and only if g satisfies the following two conditions:

(1) Any node *Mi* and *Mj*, if $M_i \in$ g and $M_i \leftrightarrow M_j \mid M_i \simeq M_j$, then $M_j \in$ g;
(2) Any node *Mi* and *Mj*, if $M_i \in$ g and $M_j \in$ g, then $M_i \approx M_j$.

The goal of the STSG algorithm is to divide all moving objects that are dependency reachable or directly reachable into the same group, and then store them in the same leaf node in the HGTPU-tree.

As shown in Algorithm 1. First, the vertex array *V* and the adjacency matrix *E* (lines 2–3) are initialized. And secondly, in the moving object array *M*, the spatial trajectory of similarity relationships between any two moving objects are calculated, and the results are recorded in *V* and *E* and an undirected graph (lines 4–10) is constructed. Then find the moving object *Mi* of the group and initialize a group *g* for *M*. Add these objects to *g* (rows 13–17) by traversing all objects that are dependency reachable or directly reachable by *Mi* through breadth-first traverse. Objects and finally return the group set *G*.

Algorithm 1 STSG

Input: Moving object set M, The minimum spatial trajectory of similarity STSMin
Output: Group set G (g1, g2, ... gn)
Sub-function description: The Judge (Mi, Mj, STSMin) function is to determine the spatial trajectory of similarity between the two object Mi and Mj. BFS (V, E, Mi) is to add all objects that are dependency reachable or directly reachable to the object Mi to g
Variable description: M_num : number of moving objects, V : vertex array, E : adjacency matrix, g : a group

```
1.  STSG(M,STSMin)
2.  Init V;
3.  Init E;
4.  for i←0 to M_num
5.    for j←0 to M_num
6.      edgs←Judge(Mi,Mj,STSMin);
// Calculate the spatial trajectory of similarity of Mi and Mj
7.      E.add(edgs);
8.    end for j
9.    V.add(Mi);
10. end for i
11. Init G;
12. for i←0 to M_num
13.   if Mi ∉ G then   // Find object Mi that are not yet grouped
14.   Init g;       // Initialize a group g for Mi
15.     g.sons←BFS(V,E,Mi);
// Find all objects that are dependency reachable and directly reachable to Mi
16.     g.id←Get_Id();
17.     G.add(g);
18.   end if
19. end for i
20. return G;
```

3.3 HGTPU-Tree Update Algorithm

When the moving object issues a position update request, the new record information is inserted into the HGTPU-tree, and the old position information needs to be deleted. HGTPU-tree synchronizes the update mechanism of the hash table and the space layer to ensure that the latest group address information is stored in the hash table in real time without saving old records.

At the space layer, update is performed by speculative positioning update. When the positional deviation of actual and the recorded position of moving object in the HGTPU-tree exceed the threshold *th*, an update operation is performed. As shown in Algorithm 2, the algorithm is mainly divided into three steps: 1. Substitute the group number of the moving object *Mi* into a hash function to obtain the address of the group in the hash table (lines 1–3); 2. Determine whether the updated position exceeds the *MBR* range of the leaf node. If the range is not exceeded, the leaf node is directly updated. Otherwise, the update process is equivalent to deleting and inserting new records in the HGTPU-tree (lines 4–16); 3. After the space layer data is updated, the address of the group in which *Mi* is stored is synchronously written back to the hash table (line 17).

Algorithm 2 UpdateTree

Input: Uncertain moving object Mi, HGTPU_tree
Output: updated HGTPU_tree'
Sub-function description: FindLeaf finds the leaf inserted by Mi in the space layer according to the address. CondenseTree deletes the leaf node and compresses the tree. AdjustTree performs structural adjustment operation on the tree after the node is split.

```
1. key←Gi
2. address←hash_fun(key);
3. L←FindLeaf(address)
4. if MBR(Mi) not beyond MBR(L)
5.   delete L
6.   CondenseTree(HGTPU_tree)
7.   L'←ChooseLeaf(HGTPU_tree,Mi) // Leaf node to be inserted
8.   if L' have free space then
9.     L'.add(Mi)        // If there is free space, insert directly
10.  else
11.    SplitNode L' to L' and LL
// The split node divides the L' node into L' and LL
12.    AdjustTree L' and LL
13.  end if
14  else
15   update L  // Directly update the L node
16  end if
17  write address(Gi) back to hash table
// Write the new group address back to the hash table
18. return HGTPU_tree'
```

The moving trajectory of the moving object that was divided into the same group, after the motion for a period of time, changes. And some moving objects deviate from the group. At this time, the group needs to be re-divided. HGTPU-tree uses a periodic detection strategy to detect moving objects in the data layer. As shown in Algorithm 3, first, the current time *t_now* (line 1) of the system is obtained, and the next update time

recorded in each group in the HGTPU-tree is compared with *t_now*. If a group needs to be updated, all the objects of the group are stored in the set *M*. The STSG algorithm is called to re-group *M* (lines 2–5), compare the new group *G'* and the old group *G*. If a change occurs, the new group *G'* is added as the data layer to HGTPU-tree (lines 6–7). Finally, update the time of the next update (line 9).

Algorithm 3 Update_Group

Input: HGTPU_tree before update
Output: updated HGTPU_tree'
Sub-function description: Adjust_Group adds a new group to the index tree when a group changes
Variable description: G_num : the number of groups, update_time : records the next update time, T : update cycle.

```
1. t_now←Get_localtime
2. for i←0 to G_num
3.   if t_now=Gi.time_uodate then
// Get system time compared to update time recorded in the group
4.     M←Gi
5.     G'=STSG(M,STSMin)// Regrouping
6.     if G < > G'
7.       HGTPU_tree'←Adjust_Group(HGTPU_tree,G')
// Insert the regrouped group into the HGTPU-tree
8.     end if
9.     update_time←update_time+T
10.  end if
11. end for i
12. return HGTPU_tree'
```

3.4 Update Cost Analysis

The cost analysis of 3 different update strategies for one update of n moving objects is given in turn: top-down update, bottom-up update, and disk I/O times required for group-based update. As shown in Table 2, the top-down update cost consists of two parts: (1) the cost of querying and deleting old records; and (2) the cost of inserting new records. Since there is a possibility of overlapping of regions between the nodes of the index, querying a record requires accessing H nodes in the best case, and in the worst case, accessing L*(L-1)/4 nodes. The old record position is searched for deletion and written back to the disk. At least one disk write operation is required in the absence of a node overflow. Before inserting a new record, at least H nodes need to be accessed to find a suitable leaf node for insertion. As a consequence, an update using the top-down update strategy requires 2n*(H+1) disk reads and writes in the best case, and nL2/4+nH +2n disk reads and writes in the worst case.

There are two cases for the bottom-up update strategy. When the new record can be directly inserted into the original leaf node where the old record is stored, the best case requires 3n disk I/O: read the secondary index (1) to locate the original leaf node, then read the leaf node (1) and write back the node (1). When the new record conflicts with the MBR of the old record, it is the worst case that H+6 disk I/O is needed: read the secondary index (1) to locate the original leaf node, then read (1) and write back (1).

Access H nodes to find the appropriate leaf node for insertion, then read (1) and write back (1) the leaf node, and finally update the secondary index (1), a total of n* (H+6).

The bottom-up update strategy based on group partition is improved on the existing bottom-up update strategy. By updating only one representative object for the moving objects of the same group, the number of updates is reduced. Consequently, for the cost of one update of n moving objects, the best case requires m*3 disk I/O, and the worst case requires m*(H+6) disk I/O. Since n is generally much larger than m, the bottom-up update strategy based on group partitioning has a minimal update cost.

Table 2. Disk I/O times for 3 update policies.

Update strategy	Number of disk I/Os updated by n moving objects at one time		
	Best case	Worst case	Average situation
Top-down	2n(H+1)	$nL^2/4+nH+2n$	$nL^2/8+1.5nH+2n$
Bottom-up	3n	n(H+6)	nH/2+4.5n
Group partition	3m	m(H+6)	mH/2+4.5m

4 Experimental Evaluation

Our experiments used Gist [17] to compare the algorithmic efficiency of index structures based on HGTPU-tree, GTPU-tree, TPU-tree and TPU^2M-tree, and gave evaluation and analysis. The experimental data set is a real-world large scale taxi trajectory dataset from the T-drive project [18, 19]. It contains a total of 580,000 taxi trajectories in the city of Beijing, 5 million kilometres of distance travelled, and 20 million GPS data points. We randomly pick 100,000 taxi trajectories from this dataset to be the query trajectories. Experimental hardware environment: CPU Intel Core i5 1.70 GHz, memory 6 GB; operating system Windows7, development environment VS2010.

4.1 Impact of STSMin on Group Partitioning and Updating

The algorithm STSG utilizes the size of STS to measure the similarity of historical trajectories of moving objects. In the STSG algorithm, the minimum spatial trajectory of similarity STSMin size directly affects the effect of group partitioning. As STSMin increases, the number of divided groups increases positively with STSMin. This is because as STSMin increases, the number of directly reachable and dependency reachable of moving objects is reduced, so that the number of groups in the dividing result increases.

A good group partition should have the characteristic that the number of deviations from the group is small for a long period of time in the future. As the STSMin increases, the number of deviations from the group gradually decreases. As the number of STSMins increases, the number of moving objects of the group increases, but the closer the historical motion trajectory of the moving object of the group is. Therefore,

the probability that moving objects will remain similar increases, and the number of moving objects that deviate from the group gradually decreases.

When STSMin is between [6, 8], the STSMin value in this interval reduces the number of divided groups and the number of moving objects that deviate from the group. Considering comprehensively, the minimum spatial trajectory of similarity STSMin is set to 6 in subsequent experiments.

4.2 Effect of the Number of Moving Objects on Node Relocation

HGTPU-tree implements bottom-up update with zero-level index hash table. When the number of moving objects increases, the number of moving objects that need to be relocationed after position updating is gradually increased. Especially at 50 K, the rate of increasing is the biggest. And finally it tends to be stable.

The space layer of HGTPU-tree is based on the R-tree. As the number of moving objects increases, the number of nodes in the index tree increases. The number of child nodes in each non-leaf node has a limit, so the MBR of each non-leaf node gradually decreases. When the MBR of the node decreases, the probability that the newly inserted node exceeds the MBR where the original record is stored increases, so the probability that the updated node needs to be relocationed increases.

4.3 Query Performance

Range query is one of the most common queries in moving object data management. We examine the query performance of HGTPU-tree through range query. The average query time of HGTPU-tree is slightly higher than that of R-tree. This is because the entire index structure of HGTPU-tree is divided into three layers. And the index tree of HGTPU-tree has more levels, so the query performance will be reduced. However, the query performance of HGTPU-tree is still roughly equivalent to R-tree on the basis of reducing the update cost.

4.4 Insertion Performance

For the moving objects of different numbers, the insertion time of HGTPU-tree, GTPU-tree, TPU-tree and TPU^{2}M-tree is shown in Fig. 1. With the increase of the number of moving objects, the required time for these four increases steadily. HGTPU-tree takes less time than TPU-tree and TPU^{2}M-tree, indicating that the insertion performance of HGTPU-tree is better than TPU-tree and TPU^{2}M-tree. This is because as the number of moving objects increases, the space layer level in the HGTPU-tree gradually increases and the free space in the index increases. Compared with the pre-grouping operation of TPU-tree and TPU^{2}M-tree, the moving objects of the same group in HGTPU-tree can be directly inserted into the nodes of the group layer, avoiding the one by one-insertion of TPU-tree and TPU^{2}M-tree. HGTPU-tree not only needs to insert it into the index tree but also needs to record it into the secondary index structure when inserting a new node. As a result, the insertion performance of GTPU-tree is slightly better than that of HGTPU-tree.

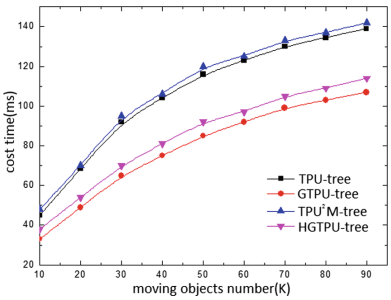


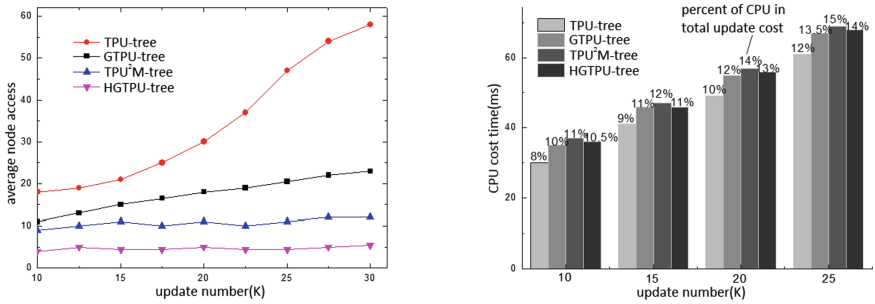
Fig. 1. Performance comparison of insert.

4.5 Update Cost

In order to ensure the accuracy of query results, it is necessary to simultaneously update the data in the database and the index. The update cost of moving objects with frequently updated positions is huge. Disk I/O and CPU are two major concerns when considering the update cost. The number of update times and the number of moving objects are the main reasons that affect the update cost of uncertain moving objects.

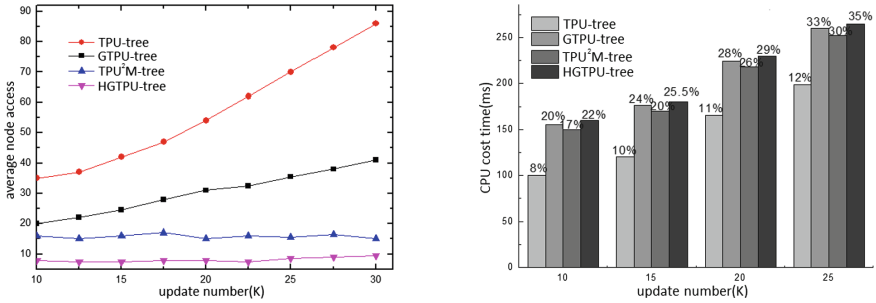
As illustrated in a of Figs. 2 and 3, HGTPU-tree greatly reduces the number of node access times compared to the other three index structures, regardless of whether it is trajectory stability or frequent group deviation. And the number of node access times directly affects disk I/O, which shows that HGTPU-tree has good performance in reducing disk I/O for moving objects with frequent position updates. HGTPU-tree improves the moving object grouping processing compared with TPU-tree and TPU^{2}M-tree, and saves the moving object of the same group in the same leaf node in the data layer. When update positions, only one moving object needs to be updated for the moving objects of the same group, reducing the number of update times. Moreover, HGTPU-tree reduces the disk I/O required for the query compared with GTPU tree, because the HGTPU tree implements bottom-up node access strategy by means of a hash table, thereby improving the update efficiency.

Comparing b of Figs. 2 and 3, we can find that the CPU calculation cost of HGTPU-tree is slightly higher than that of TPU-tree and GTPU-tree, and accounts for a larger proportion of the overall update cost. But when the moving object group trajectory is stable, the CPU calculation cost of HGTPU-tree is lower than that of TPU^{2}M-tree. This is because when the node update is performed, TPU^{2}M-tree needs to query the memo first. Furthermore, when the number of records in the memo increases, additional space cleaning operations are required, which increases the CPU calculation cost. In HGTPU-tree, the moving objects of same group maintain the same motion trajectory before the next group update. However, for the moving object trajectory is uncertain, periodic regroups are required in order to ensure the similarity of the moving object trajectory of same group, which increases the CPU calculation cost. HGTPU-tree has higher CPU cost than GTPU-tree. This is because HGTPU-tree needs to read and query the hash table when performing position update, and needs to update to the hash table synchronously, which increases CPU cost.



a. Performance comparison of I/O cost **b.** Performance comparison of cost

Fig. 2. Compare I/O+CPU cost with group trajectories stable.



a. Performance comparison of I/O cost **b.** Performance comparison of cost

Fig. 3. Compare I/O+CPU cost with group frequent updates.

As Figs. 4 and 5 show, the HGTPU-tree overall update cost is smaller than the other three index structures whether it is in the scene where the moving object group trajectory is stable or the group frequently deviates. With the increase in the number of moving object pairs, the advantages of HGTPU-tree are more obvious.

For TPU^{2}M-tree with bottom-up update, as the number of moving objects increases, the number of nodes in the index tree increases and the MBR of each non-leaf node gradually decreases. When the MBR of the node decreases, the probability that the newly inserted node exceeds the MBR where the original record is stored increases. If the new node exceeds the original recorded MBR, it is equivalent to inserting a new record in the index tree, and the update efficiency is reduced. Compared with GTPU-tree that is also based on group partition, HGTPU-tree reduce disk I/O and the update cost by means of hash table.

For HGTPU-tree, when the number of moving objects increases, the number of moving objects of each group increases correspondingly, which is more conducive to the overall update. Especially in the case of stable group trajectory, the advantage of HGTPU-tree is more obvious. Because the update period T of group re-partition can be appropriately increased in the case where the trajectory of the moving object group is

stable compared to the frequent deviation of the group. Therefore, in the same time period, the group trajectory is stable, which can reduce the CPU cost caused by group re-partition, thereby reducing the overall update cost.

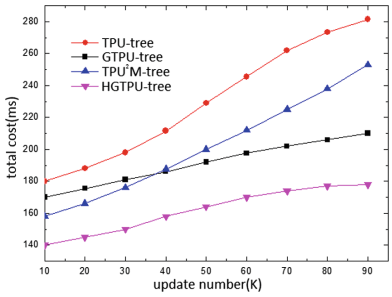


Fig. 4. Total cost with trajectories stability.

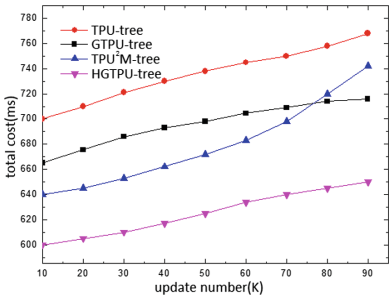


Fig. 5. Total cost with trajectories deviation.

4.6 Memory Cost

Figure 6 shows the memory cost of HGTPU-tree and TPU^{2}M-tree under the different numbers of moving object. As the number of moving objects increases, the memory cost of TPU^{2}M-tree fluctuates periodically. TPU^{2}M-tree is based on the UM structure. The old records are retained when the moving object updates position. Meanwhile, TPU^{2}M-tree cleans up old records periodically. HGTPU-tree uses synchronize update mechanism with the index tree to update the address content in the hash table every time the moving object is updated, thus there is no need to save the old record. Moreover, HGTPU-tree only records the address of one single object for the same group of moving objects. Therefore, HGTPU-tree can greatly reduce the memory cost and improve the system stability.

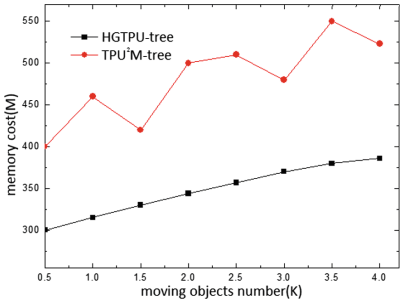


Fig. 6. The performance of memory cost.

5 Conclusion

In this paper, we developed an index structure HGTPU-tree that supports moving object group partition and bottom-up group update strategy. We proposed a group partition algorithm STSG and a moving object group update algorithm. Experiments based on real dataset analyzed the performance comparison of HGTPU-tree, GTPU-tree, TPU-tree and TPU^{2}M-tree in different situations. The results show that HGTPU-tree is better than TPU-tree and TPU^{2}M-tree in insertion performance. In terms of update cost, the update cost of HGTPU-tree is lower than other three index structures, especially when the moving object group trajectory is stable. HGTPU-tree increases the complexity of the index structure in terms of query performance. However, the query performance can still be approximately equivalent to the traditional index. HGTPU-tree solves the problem of the high memory cost of existing bottom-up update indexes with synchronous update mechanism.

References

1. Li, J., Wang, B., Wang, G., et al.: A survey of query processing techniques over uncertain mobile objects. J. Front. Comput. Sci. Technol. **7**(12), 1057–1072 (2013)
2. Saltenis, S., Jensen, C.S., Leutenegger, S.T.: Indexing the Positions of Continuously Moving Objects. ACM SIGMOD 2000, Dallas, Texas, USA (2000)
3. Li, B., et al.: Algorithm, reverse furthest neighbor querying, of moving objects. In: ADMA 2016, Gold Coast, QLD, Australia, pp. 266–279 (2016)
4. Güting, R.H., Schneider, M.: Moving Objects Databases, pp. 220–268. Elsevier (2005)
5. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: an optimized spatio-temporal access method for predictive queries. In: VLDB, pp. 790–801 (2003)
6. Procopiuc, Cecilia M., Agarwal, Pankaj K., Har-Peled, S.: STAR-tree: an efficient self-adjusting index for moving objects. In: Mount, David M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 178–193. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45643-0_14
7. Saltenis, S., Jensen, C.S.: Indexing of moving objects for location-based services. In: ICDE, p. 0463 (2002)
8. Fang, Y., Cao, J., Peng, Y., Chen, N., Liu, L.: Efficient indexing of the past, present and future positions of moving objects on road network. In: Gao, Y., Shim, K., Ding, Z., Jin, P., Ren, Z., Xiao, Y., Liu, A., Qiao, S. (eds.) WAIM 2013. LNCS, vol. 7901, pp. 223–235. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39527-7_23
9. Pelanis, M., Saltenis, S., Jensen, C.S.: Indexing the past, present, and anticipated future positions of moving objects. ACM Trans. Database Syst. (TODS) **31**(1), 255–298 (2006)
10. Lee, M.L., Hsu, W., Jensen, C.S., et al.: Supporting frequent updates in R-trees: a bottom-up approach. In: Proceedings of the 29th International Conference on Very large data bases-Volume 29. VLDB Endowment, pp. 608–619 (2003)
11. Qi, J., Tao, Y., Chang, Y., Zhang, R.: Theoretically optimal and empirically efficient r-trees with strong parallelizability. Proc. VLDB Endowment (PVLDB) **11**(5), 621–634 (2018)
12. Tao, Y., Cheng, R., Xiao, X., et al.: Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: Proceedings of 31st International Conference, VLDB 2005, pp. 922–933. Morgan Kaufmann Publishers, Inc. (2005)

13. Ding, X., Lu, Y., Pan, P., et al.: U-Tree based indexing method for uncertain moving objects. J. Softw. **19**(10), 2696–2705 (2008)
14. Ding, X.F., Jin, H., Zhao, N.: Indexing of uncertain moving objects with frequent updates. Chin. J. Comput. **35**(12), 2587–2597 (2012)
15. Sadahiro, Y., Lay, R., Kobayashi, T.: Trajectories of moving objects on a network: detection of similarities, visualization of relations, and classification of trajectories. Trans. GIS **17**(1), 18–40 (2013)
16. Ra, M., Lim, C., Song, Y.H., Jung, J., Kim, W.-Y.: Effective trajectory similarity measure for moving objects in real-world scene. In: Kim, Kuinam J. (ed.) Information Science and Applications. LNEE, vol. 339, pp. 641–648. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46578-3_75
17. Stamatakos, M., Douzinas, E., Stefanaki, C., et al.: Gastrointestinal stromal tumor. World J. Surg. Oncol. **7**(1), 61 (2009)
18. Yuan, J., Zheng, Y., Xie, X., Sun, G.: Driving with knowledge from the physical world. In: Proceedings of the KDD, pp. 316–324 (2011)
19. Yuan, J., et al.: T-drive: Driving directions based on taxi trajectories. In: Proceedings of the GIS, pp. 99–108 (2010)