

# Efficient Index Maintenance for Moving Objects with Future Trajectories

Rui Ding, Xiaofeng Meng, Yun Bai  
Information School, Renmin University of China  
Beijing 100 872, P. R. China  
[xfmeng@mail.ruc.edu.cn](mailto:xfmeng@mail.ruc.edu.cn)

## Abstract

*Recently, more research has been conducted on moving object databases (MOD). Typically, there are three kinds of data for dynamic attributes in MOD, i.e., historical, current and future. Although many index structures have been developed for the former two types of data, there is not much work to deal with the future data. In particular, the problem of index update has not been addressed with efficient methods. This paper proposes a novel spatio-temporal index based on PMR quadtree, which is called Future Trajectory Quadtree (FT-Quadtree). FT-Quadtree adopts a trajectory segment shared structure and depicts an efficient update algorithm. The performance studies have shown that FT-Quadtree has superiority to the traditional one in index maintenance.*

## 1. Introduction

With the rapid development of wireless communications as well as positioning technologies, the concept of moving objects has become more and more important. The need for storing and processing continuously moving data arises in a wide range of applications, including traffic control or monitoring, transportation and supply chain managements, digital battlefields, and mobile e-commerce [14].

Typically, there are three kinds of data with dynamic attributes in MOD, i.e., historical, current and future. Present research has mostly focused on the management of historical and current positions of objects, while not many studies are carried out about the future status of moving objects. When querying a moving object, the user may be more interested in the future location. For instance, the driver of a broken car may be more interested in “which service car will arrive within the next half hour?” To enable such queries, each object is required to provide a moving plan. The moving plan depicts the route that the object will follow and the speed on each route segment. According to the moving plan, a future trajectory is maintained in MOD. Thus, the future location at a specific time can be determined without polling its location at regular intervals. However, with so many future trajectories, how to manage them is a significant issue.

Among so many moving plans submitted to the database system, there may be a lot of redundant information. For example, considering a city traffic

control system, it is possible that some vehicles move on nearly the same route segment during about the same period. Thus, their moving plans include some approximate information, and the corresponding trajectory segments may be close, even overlap in the index space. In addition, moving plans are usually changed unexpectedly in real world. For example, a bus has to stop for some reasons, e.g. traffic congestion, accidents, etc., then its following moving plan submitted before should be modified. Consequently, it is necessary to update the original index.

As presented above, there are many problems to be solved in this research area. Because of the “mobility” in mobile computing environment and “dynamic attributes” [8] of objects in MOD, a good index structure is needed. Furthermore, the modification of the movement of objects may result in adjusting the index structure, which requires an efficient index update work. Therefore, the index maintenance becomes the key issue to be taken into account. Basically, there are two main issues related to index update: one is “when-to-update” and the other is “how-to-update”. Many papers [8, 9, 12, 13] have studied the first one. When referring the update algorithm, they simply deploy the reconstruction method. There are two major problems for the reconstruction method. First, rebuilding indices brings a large amount of unnecessary update cost. Second, querying cannot be conducted during the update period.

In this paper, we propose a novel index called Future Trajectory Quadtree (or FT-Quadtree for short), which efficiently indexes moving objects and reduces update cost greatly. Based on the idea of PMR quadtree, we design a trajectory segment shared structure aiming at the problem of redundant moving plans. As for the index update, we adopt a different strategy of updating index dynamically instead of rebuilding index periodically. Different from traditional “delete-insert” based update method, our new update method is based on the “insert-delete” operation. It adopts space reused technique, i.e., the space of deleted data can be reused for the insertion of modified trajectory segments.

The rest of the paper is organized as follows. Section 2 describes the problem of index update we focus on. Section 3 introduces the FT-Quadtree in details, including the novel index structure and the algorithms of insertion, deletion and update. The experiment of performance evaluation and the results are given in Section 4. Section 5 describes the related work on the moving object indexing. Finally, the summary and future work are presented in Section 6.

## 2. Problem Statement

First, we will present a sample of a traditional application scenario, the system for tracking automobiles [4].

A traffic navigation system is important to a city. At the central site, there is a database system for managing all data of vehicles. Each vehicle is equipped with a GPS device. Vehicles measure their current locations with the GPS device and transmit their positions to the central site using either radio communication links or cellular phones. The database system stores the current positions of all vehicles and process queries. In this way, the navigation system can only query the past and current locations of moving objects.

In order to query the future locations of moving objects, such as "Which taxi will arrive at Friendship Hotel?" and "Retrieve all taxies within 1k meters of Friendship Hotel in the future half hour.", the navigation system must store some more complicated information, such as moving plans, which will be described in Section 2.1 in detail. Traffic congestions are common phenomena occurred frequently in highly populated areas. Once a car is delayed by some unexpected reasons, such as traffic accidents, its following moving plan should be adjusted. The index built before should also be updated consequently. The problem we focus on is the reduction of update cost in such a situation.

### 2.1 Preliminary Definitions

Before introducing the FT-Quadtree, we would like to define and explain some notions, which will be used throughout the paper.

**Definition 1: Moving Segment (MS)** is an 8-tuple  $\langle MOID, Number, StartX, StartY, EndX, EndY, Time, Velocity \rangle$  in 2-dimensional space, which depicts a n instance that a moving object on a specific route segment. *MOID* is a unique object identifier; *Number* is the sequence of the MS in the Moving Plan (MP, see Definition 2); *StartX* and *StartY* (*EndX* and *EndY*) are two values to denote the start (end) point of a specific route segment in 2-dimensional space; *Time* is the starting time of the moving object; *Velocity* is the speed of the moving object on the route segment.

**Definition 2: Moving Plan (MP)** is a series of moving segments (MS) of an object. It depicts the movement of the object on all its route segments. MP is a semantically ordinal queue. Any MS in the MP has its specific position determined by the element *Number* in the 8-tuple defined above. Figure 1 illustrates an MP instance. Six lines with arrow specify a MP of a moving object. Each line with an arrow represents a MS.

**Definition 3: A Trajectory (T)** is a polyline in a high dimensional space, which combines the spatial and temporal information of a moving object. If the object moves in a  $d$ -dimensional space, the future trajectory is a polyline in a  $(d+1)$ -dimensional space. A trajectory can be denoted by a series of 6-tuples  $\langle MOID, Number,$

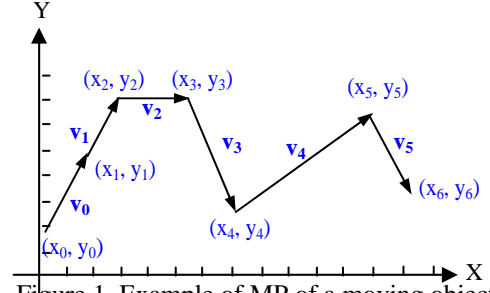


Figure 1. Example of MP of a moving object

$\langle Start, StartT, End, EndT \rangle$ , which is a **Trajectory Segment (TS)**. *MOID* is a number uniquely identifying the object. *Start* and *End* are two vectors specifying the start-point and end-point of the corresponding route segment. The vector *Start* (*End*) can be expressed as  $(StartX(EndX), StartY(EndY))$  if the object moves in 2-dimensional space  $(X, Y)$ , and  $StartX(EndX)$  if in 1-dimensional space  $X$ . *StartT* and *EndT* specify the starting and ending time. *Number* indicates where the TS is in the whole  $T$  of *MOID*. The 6-tuple means an object *MOID* moves from *Start* to *End* in the period of  $(StartT, EndT)$ . One example of an object moving in two-dimensional space ( $d=2$ ) is shown in Figure 2. The projection of the trajectory on the  $(X, Y)$  plane is just the route of the object, as the left polyline shown in Figure 2.

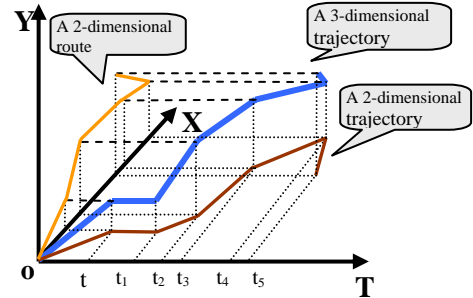


Figure 2. A trajectory in  $(X, Y, T)$  and one in  $(X, T)$

These 6-tuples are not originally submitted by mobile users, but obtained according to the MP. There are two differences between a Trajectory (T) tuple and a Moving Segment (MS) tuple. Firstly, MS includes speed information, while a T tuple includes starting and ending spatio-temporal information. Secondly, MS is used for the route map, added by users' anticipant speed, while T is used for indexing. Ts are stored in the central management system and are the basis of index entries.

Since the objects we consider in this paper move in the two-dimensional space, thus their trajectories are three-dimensional. It is complicated to deal with such trajectories, so we decompose the movement of the object into two directions,  $X$  and  $Y$ . Accordingly, there are two sub-trajectories respectively in two spatio-temporal planes,  $(X, T)$  and  $(Y, T)$ . The right polyline in Figure 2 illustrates the one in  $(X, T)$ , which can be denoted by a series of 6-tuples  $\langle MOID, Number, StartX, StartT, EndX, EndT \rangle$ . It is similar to the projection of

the middle trajectory onto (Y, T). For simplicity, if not specified, the term “trajectory” in the following paper refers to the sub-trajectory in (X, T).

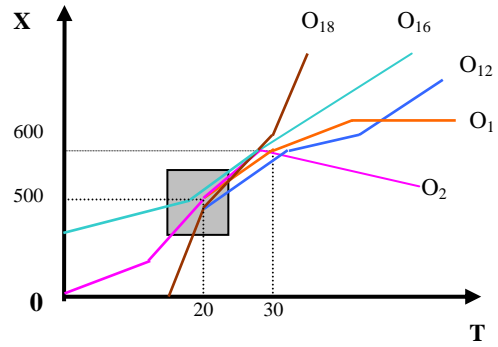
## 2.2 Two problems about the naive quadtree-based spatio-temporal index

The indexing methods for trajectories of moving objects can be based on the spatial index, such as R-tree and quadtree etc. As the technique proposed in this paper builds on the basic idea of PMR quadtree, which can be referred in [7], we will explain the reasons of our choice. PMR quadtree is an index built on line segments in 2-dimensional plane, while the movement of moving objects is described by series of line trajectories. Because of coincidence of the indexing data types, we consider about PMR quadtree firstly. J. Tayeb, etc. applied PMR Quadtree on dynamic attribute index for the first time [11]. They proposed an efficient index construction method and indicated that using quadtree while indexing moving objects is also an adequate choice.

For each quadrant, there is accordingly a leaf node in the quadtree. And if a TS of an object intersects a quadrant, there is to be an index entry in the corresponding quadtree node. However, there exist two problems that affect the efficiency of the quadtree updating: one is about the index structure, the other is about the updating algorithm.

**(1) Redundant information and infinitely splitting.** In the city traffic, although some objects move on different route segments during the nearly same period, it is really possible that the projection of their starting and ending positions onto either direction are nearly the same. As shown in Figure 3(a),  $O_1$ ,  $O_2$ ,  $O_{12}$ ,  $O_{16}$  and  $O_{18}$  are five moving objects, which share the approximate movement in the x-direction regardless of y-direction. That is to say, they all approximately move from 500 to 600 in the x-direction in the period from 20 to 30. Thus, their trajectory segments are close to each other and may intersect with the same quadrant, e.g., the grey square in Figure 3(a). Since their trajectory segments belong to different objects, they have different index entries, as shown in Figure 3(b), though they have many similarities.

Obviously, the index organization will occupy excessive space. What’s more, if the number of objects in the above case is close to, even greater than, the splitting threshold, the quadrant is quite possible to overflow, i.e., the sum of trajectory segments intersecting with it is larger than splitting threshold. Thus, the quadrant is sure to be split into four sub-quadrants. And if the index entries corresponding to the five adjacent trajectory segments are still in one of the four sub-nodes, the sub-node may still overflow and continue to be split. However, the splitting process does not have any effect on the five adjacent trajectories, and so the splitting will go on until the quadrant is small



(a) Example of trajectories

<math>\langle O_1, 1, 500, 600, 20, 30, * \rangle</math>
<math>\langle O_2, 3, 500, 600, 20, 28, * \rangle</math>
<math>\langle O_{12}, 1, 480, 600, 20, 32, * \rangle</math>
<math>\langle O_{16}, 2, 500, 600, 18, 28, * \rangle</math>
<math>\langle O_{18}, 2, 480, 640, 20, 30, * \rangle</math>
<math>\langle O_{18}, 1, 0, 500, 15, 20, * \rangle</math>
.....

(b) Corresponding index entries

Figure 3. Example of trajectory and index entries

enough to make the adjacent trajectory segments intersect with different quadrants. As a result, such a continuous splitting makes some embranchment of the quadtree very deep, which affects the efficiency of the query and index update.

**(2) Low update efficiency by unnecessary split and merge operations.** The general updating (named GNRUPT) algorithm consists of two steps: firstly, deleting the corresponding index entries of the trajectory segments; secondly, adding the corresponding index entries of the modified segments. However, in some critical situation, this way makes the update very inefficient.

For example, as shown in Figure 4(a), a quadtree is created given the indexed space shown on the left. When the trajectory of an object  $O_3$  changes at  $t_u$ , all the subsequent trajectory segments are deleted from the quadtree nodes that including them (grey leaf-nodes in the figure), which just makes the amount of index entries in some node and its siblings less than the splitting threshold and then results in two times of merge (see Figure 4(b)). Some of the new versions of deleted segments may intersect the quadrant derived from the previous merge when inserting them into the corresponding node. As the result, the index entries in it will increase and become greater than the splitting threshold, so two splits occur (see Figure 4(c)). Finally, the quadtree after updating and the originally created one are much alike. The merge-and-split process wastes elapsed time and causes low updating efficiency.

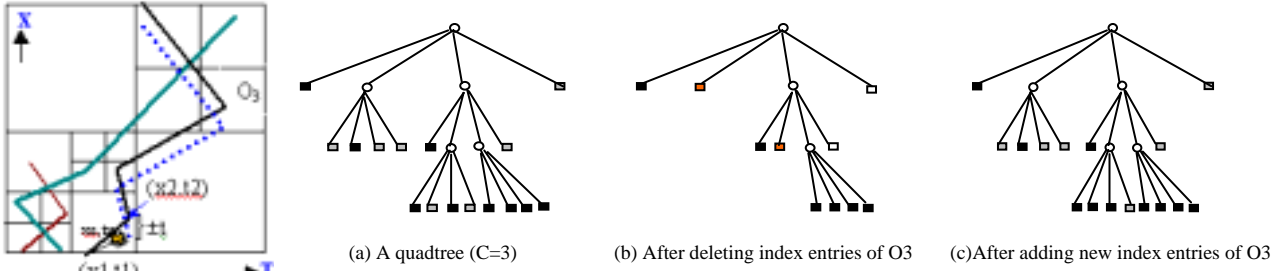


Figure 4. Illustration of traditional update method

### 3. FT-Quadtree

To solve the two problems discussed above, we propose a novel quadtree named *Future Trajectory Quadtree* (FT-Quadtree for short). In addition to the improvement in the index structure, FT-Quadtree adopts a new updating algorithm. Since location is a typical dynamic attribute, we mainly discussed the location index problem when mentioning dynamic attribute index. As a  $d$ -dimensional location attribute  $(l_1, l_2, l_3, \dots, l_d)$ , it has  $d$  sub-attributes, which are  $l_1, l_2, l_3, \dots, l_d$ . When building the temporal index on such an attribute, we build the sub-index in each sub-space of 2-dimensional: the location sub-attribute and time plane. In this way, we can get  $d$  sub-indices. The paper presents FT-Quadtree method only in sub-index plane, i.e., the index plane composed of 1-dimensional location attribute and a derived time attribute. As the indexing approach in the sub-space is the same, we might as well refer to the case in  $(X, T)$ .

The FT-Quadtree is used directly for the query of objects moving in 1-dimensional space. In section 3.3, we will show how to apply it to query moving objects with multi-dimensional location attribute.

#### 3.1 Index Structure

**Definition 4:** A **Shared Trajectory Segment** is a trajectory segment shared by different moving objects. It is in the form of  $\langle StartX, EndX, StartT, EndT \rangle$ , in which  $StartX, EndX, StartT$  and  $EndT$  have the same meaning as in the TS. It can be used by a single object or shared by a number of objects. Different from previous work, shared trajectory segments serve as our index entries. A shared trajectory segment followed by an **object list** is an index entry. If the shared trajectory segment represents only one object's movement, it is equivalent to the objects' trajectory segment entry. However, if a number of objects share the same trajectory segment, the situation is different.

The shared trajectory segments are derived from Trajectories directly and from Moving Plans indirectly. If the movements in the TSs of different objects are approximate and the difference is less than a threshold, we regard these TSs as having "common" information, including the starting and ending spatial-temporal points.

Then a shared trajectory segment is formed by the extracted "common" information.

With such an information extracting idea, we design the PMR quadtree based dynamic attribute index. In the FT-Quadtree, an index entry is denoted by  $\langle StartX, EndX, StartT, EndT, pmo \rangle$ .  $\langle StartX, EndX, StartT, EndT \rangle$  is just a shared trajectory segment.  $pmo$  is a pointer that points to an object list in the form of  $\langle MOID, Number \rangle$ .  $MOID$  and  $Number$  are the same as in the TS. The objects in the list pointed by  $pmo$  are those who share this index entry, i.e., each of these objects has a trajectory segment which is approximate, even equal, to the shared trajectory segment  $\langle StartX, EndX, StartT, EndT \rangle$ .

As shown in Figure 5, the left list is some example of MSs. They have nearly the same movement in the  $x$ -direction, but different in  $y$ -direction. After the process of projection respectively on  $x$ - and  $y$ -direction, the corresponding index entries in two naive PMR quadtrees are derived, as shown in the middle two pictures. By applying the information extracting idea on the trajectory segments in  $(X, T)$ , the index entry, i.e.,  $I_1$  in the FT-Quadtree is derived, as shown in the right picture. With the pointer  $pmo$ , all the objects sharing the identical index entry can be found.

Comparing with those un-preprocessed index methods, the number of index entries reduces enormously in the FT-Quadtree. Figure 6 displays the difference. In the figure, the left part is the naive index tree and the index entries are moving object trajectory segments. The right part is FT-Quadtree structure.  $I_1$  is the Trajectory Segment (TS) shared by  $X_1, X_2, X_3, X_4, X_5$ , i.e. the  $I_1$  shown in Figure 5.

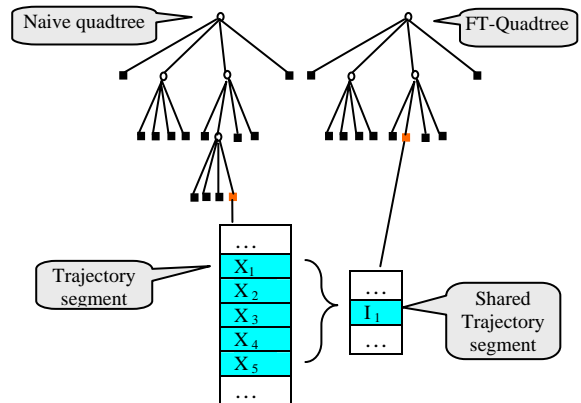


Figure 6. Comparison of two kind of index tree

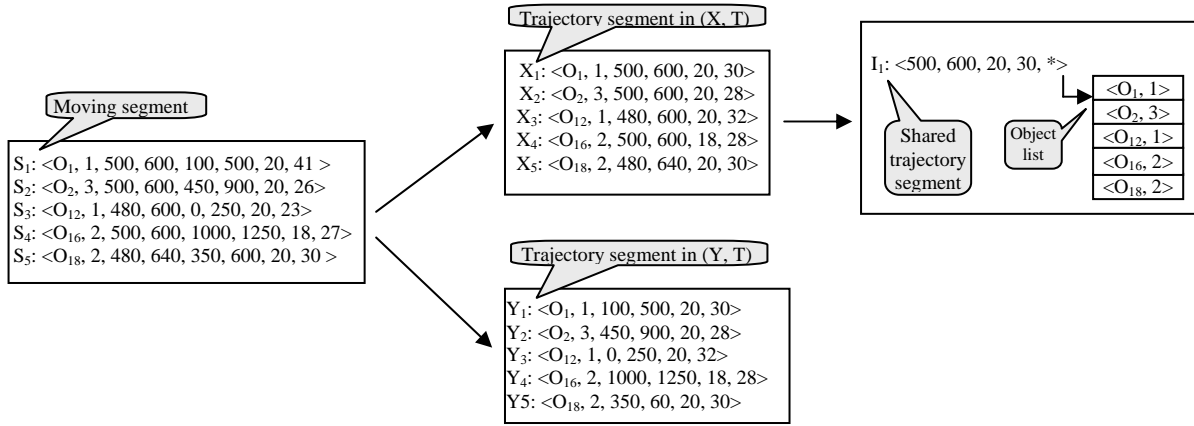


Figure 5. Index entries in naive PMR quadtree and FT-Quadtree

### 3.2 Algorithm

In the FT-Quadtree, the inserting, deleting and updating operations will be different. Compared with the algorithms in the traditional quadtree-based index, the inserting and deleting algorithms are accordingly modified due to the new-designed index structure, and the updating algorithm (named RUSUPT) mainly aims at the solution of the second problem discussed in subsection 2.2.

**3.2.1 Insertion.** The FT-Quadtree is constructed by inserting the Trajectory Segments (TS) of each object one-by-one into an initially empty structure consisting of one quadrant. When a new TS(*SEG*) of a moving object(*MO*) needs to be inserted into the FT-Quadtree, for every leaf node, the insertion function is called (see Algorithm 1). The basic idea is to check whether *SEG* has “common” movement information with an existing one.

#### Algorithm 1: Insertion

```

insert (SEG, Q):
Parameter descriptions:
  a TS SEG,
  a FT-Quadtree node Q.
begin
  if (SEG does not intersect Q) then
    return;
  else
  {
    for each index entry I in the Q
      if (SEG.moveinfor = I.information) then
      {
        add an item for SEG.MOID in I->object link list;
        return;
      }
    add a new index entry NI in Q;
    add an item for SEG.MOID in NI->object link list;
  }

```

For a leaf node *Q* intersected by *SEG*, if the movement information in *SEG* is approximate to that in an index entry *I* in *Q*, the only thing that should be done is to add the object information into the object link list of *I*, which specifies that *SEG* of *MO* shares the existing *I*. Otherwise, a new index entry *NI* with an object link

list containing only one object record should be created and inserted into *Q*.

The amount of index entries in a node is denoted by *CurrentSize*, which is also used in the following deleting and updating algorithms. If the insertion causes the result that the *CurrentSize* of *Q* exceeds the splitting threshold, the splitting operation (see Algorithm 2) is required to split *Q* into four smaller nodes of equal size. After splitting, *SEG* and all the trajectory segments in *Q* are to be inserted into each one of the four sub-nodes, i.e. the function *insert()* is called four times respectively for each sub-node.

#### Algorithm 2: Split

```

split (Q):
Parameter descriptions:
  a FT-Quadtree node Q.
begin
  split Q into Q1, Q2, Q3, Q4;
  for each TS SEG in Q
    for each Qi
      insert(SEG, Qi);
end

```

**3.2.2 Deletion.** The deletion process is simply an inverse process of insertion (see Algorithm 3). When deleting a TS(*SEG*) of a moving object(*MO*) from the FT-Quadtree, it is required to delete it from all the nodes that intersected by it. For a certain node *Q*, after making sure that *SEG* intersects *Q*, an important step is that if there are more than one items in the object list of the corresponding index entry *I* of *SEG*, the item including the information of *MO* is taken out. If there is only one item, its index entry *I* will be taken out of *Q*. The deletion of *I* perhaps makes the sum of the *CurrentSize* of *Q* and that of *Q*'s siblings less than the splitting threshold. Once it happens, *Q* and its siblings should be merged, i.e., the merge operation (see Algorithm 4) is executed on *Q* and its sibling.

**3.2.3 Update.** To implement our update method of “insert-delete”, the concept of “version” is introduced to distinguish the former TS and new TS. And the version of the former one is considered older than the latter. Thus, after insertion operation, although the former TS

### Algorithm 3: Deletion

```
delete (SEG, Q):
Parameter Descriptions:
  a TS SEG,
  a FT-Quadtree node Q
begin
  if (SEG does not intersect Q) then
    return;
  else
  {
    find the corresponding index entry I of SEG;
    n=the number of items in I->object link list;
    if (n>1)
      delete the item for SEG.MOID in I->object link list;
    else
      delete I;
  }
end
```

### Algorithm 4: Mergence

```
merge (Q):
Parameter Descriptions:
  a FT-Quadtree node Q
begin
  fine the father  $Q_0$  of Q;
  fine the sibling  $Q_1, Q_2, Q_3$  of Q;
  for each SEG in Q
    insert(SEG,  $Q_0$ );
  for i=1 to 3
    for each SEG in  $Q_i$ 
      insert(SEG,  $Q_0$ );
  delete Q,  $Q_1, Q_2, Q_3$ ;
end
```

and the new TS coexist, they will not be confused because of different versions. And after deletion operation, new TSs can reuse the space of original ones.

When a moving object stops because of unexpected situation, its subsequent TSs are to be updated as soon as possible. Concerning an object(MO) stopping at  $X$  at the time of  $T$ , the updating process has two phases: inserting the new TSs of MO (denoted as *New\_version*) and deleting the former TSs (denoted as *Old\_version*). In the first phase, after inserting a TS in *New\_version* into a node  $Q$  intersected by it, if none of the *Old\_version* is in  $Q$ , the following deletion process will not affects  $Q$ . Then if the *CurrentSize* of  $Q$  exceeds the splitting threshold, the split process will be executed. While if some of *Old\_versions* are in  $Q$ , the splitting will be ignored. In the second phase, after deleting a TS in *Old\_version* from a node  $Q'$  intersected by it, if the *CurrentSize* of  $Q'$  exceeds the splitting threshold, the splitting operation will be executed, and if the sum of the *CurrentSize* of  $Q'$  and that of its siblings is less than the splitting threshold, the mergence operation will be executed.

### 3.3. Querying with FT-Quadtree in multi-dimensional space

We have mentioned above that the querying of dynamic attributes in multi-dimensional space can be implemented by the method used in one-dimensional space. Now we will discuss the details. Our method is to make projections onto every dimension of the multi-di-

### Algorithm 5: RUSUPT

```
update (X, T, MO):
Parameter Descriptions:
  the updating position X and time T of the object MO;
begin
  keep the segments after (X, T) of MO as Old-version;
  modify the segments in Old-version as New-version;
  for each SEG ∈ New-version
  {
    for each leaf node Q
      if (SEG intersects Q) then
      {
        insert(SEG, Q);
        if (none of Old_version is in Q) then
          if ( $Q.CurrentSize > \text{splitting threshold}$ ) then
            split(Q);
      }
    }
  for each SEG ∈ Old-version
  {
    for each leaf node Q'
      if (SEG intersects Q') then
      {
        delete(SEG, Q');
        if ( $Q'.CurrentSize > \text{splitting threshold}$ ) then
          split(Q');
        s= $Q'.CurrentSize + Q''s \text{ sibling}.CurrentSize$ ;
        if ( $s < \text{splitting threshold}$ ) then
          merge(Q');
      }
    }
  }
end
```

dimensional querying rectangle, and get the resulting objects set which goes through the projection on every dimension. Then it is necessary to verify that the objects going through the projection area in one dimension space can satisfy the querying conditions.

If a moving object satisfies querying request in multi-dimensional space, it must go through the projection area on every dimension of querying rectangle at the same time. So when we use the one-dimensional indexing method mentioned above to deal with the projection, we need return two result sets: The first one is the object set that satisfies the projection area on the attributes of  $i$ th dimension,  $O_i = \{ o_j \mid o_j \text{ crosses the } i\text{th projection area} \}$ . The second one is the crossing time period set derived from the object set. It can be represented as  $T_i = \{ [t_{j_1}, t_{j_2}] \mid t_{j_1}, t_{j_2} \text{ are the beginning and ending time of object } o_j \text{ crossing the } i\text{th projection area} \}$ . The way of dynamic attribute querying process in multi-dimensional space is described as follows.

- (1) Process the querying projection area in the first 1-dimensional space and then return  $O_1$  and  $T_1$ .
- (2) Process the querying projection area in the  $i$ th 1-dimensional space and then return  $O_i$  and  $T_i$ . When fixing  $O_i$ , it should be noticed that objects in  $O_i$  must appear in  $O_{i-1}$ , i.e.,  $O_i \subseteq O_{i-1}$ . The next step is to work out  $T_i$  accordingly.
- (3) Process further  $T_i$  ( $1 \leq i \leq d$ ) which has been worked out, that is to judge if there is a overlap among the time periods  $[t_{j_1}, t_{j_2}]$  belonging to the

same moving object in different time periods set  $T_i$ .

If the overlap exists, i.e.,  $\bigcap_{i=1, o_j \in O_d}^d T_i.[t_{j_1}, t_{j_2}] \neq \Phi$ ,  $o_j$

satisfies the final querying conditions and can be put into the final result set  $O$ .

(4) Finally return the querying result set  $O$ .

## 4. Performance Evaluation

In this section, we will present the results of some experiments to analyse the performance of the FT-Quadtree with respect to the update performance. We have implemented a system called FTMOD (Future Trajectory Moving Objects Database) and do our experiments on it. To verify the effectiveness of the FT-Quadtree, we compared it with the naive quadtree, which uses the regular update algorithms. The experimental settings and designs are described in subsection 4.1 and the results of experiments are given in subsection 4.2.

### 4.1 Experimental setup and testing design

The FT-Quadtree and the standard quadtree are both implemented using VC++ on a personal machine with PIII 550MHz CPU, 128MB of memory and 20GB HDD. Because the large amount number of testing data, the page size (the tree node size) is set to 12K. Thus a full leaf node can contain 1600 entries, which is the least node capacity for the testing data. We don't employ any buffer policy.

When generating moving objects and simulating updates, there is a route map to restrict both process. The route map is a 500\*400 rectangle. There is a destination every 10 along each side. Thus totally there are 2091 destination points in the map. Meanwhile, there are 4090 two-way routes in the map. When a moving object is introduced, it is placed at a random destination point in the route map. Its moving plan will also come into being randomly according the route map information in the following step. The number of trajectory segments is variable from 1 to 10. Totally 10,000 moving objects are introduced to our experiment and accordingly 54,808 trajectory segments are produced in our testing instance. Moving objects travel at a constant speed from the start to the end destination. We assume that objects accord with the plans while moving and only can be delayed by unexpected traffic incident. Once the moving objects deviate from the original plan and the deviation exceeds the accuracy threshold, moving plans and index updates will be happened.

FT-Quadtree is organized out of memory, so its size is not restricted by memory. Table 1 summarizes the parameters used in our experiment. Standard values are given in bold-face. MO and SEG determined the total

number of the testing data. CHGMO and CHGSEG are the number of changed objects and segments, while CHGMO/MO and CHGSEG/SEG depict the extent the update process involves. C is the capacity of one index node, which is the splitting threshold in the quadtree.

## 4.2 Experiments results and analysis

**4.2.1 Comparing the efficiency of RUSUPT and GNRUPT.** Figure 7 shows the performance differences of GNRUPT and RUSUPT. Figure 7(a) shows the time consumed with varying numbers of updating objects once update process. The horizon beeline represents the index construction time. The upper declining line depicts the traditional update algorithm (GNRUPT) performance while the lower one depicts the space reusable update algorithm (RUSUPT). If it takes more time to update the index than construct it, it is better to reconstruct the index while the data derived from original values (exceeding the accuracy threshold). So we consider that updating lines above the horizon constructing line are invalid. The graph also tells us RUSUPT can be applied to more objects updating process while GNRUPT's extent is narrow. Figure 7(b) reveals to us the I/O operations with varying numbers of updating objects once update process. Because buffer policies are not applied to our experiment, the value of I/O operations is high. But the relative I/O operation gap also tells us RUSUPT has a better performance.

**4.2.2 Impacts of shared trajectory segment extraction.** As we know, shared trajectory segment extraction can reduce the number of indexing objects and further predigest our indexing work. For we construct indices in subattribute-time planes respectively, the experiments have been done in one of the subattribute-time planes. Table 2 can depict the instance and we will give some explanation.

The table presents the impact of shared segment extraction. The first column is the total amount of moving objects travelling in the system. The second column is the number of trajectory segments (about 5.5 segments per each object). The third is the number of shared segments in the original three-dimensional tempo-spatial space. The fourth and fifth column are the numbers of shared segments in (X, T) and (Y, T) planes, which are the numbers of indexing entries in FT-Quadtree after extraction in subattribute-time plane.

1,510 and 1,210 are the maximum number restriction in the subattribute-time planes respectively. That is to say whatever how many moving objects travelling on the route map in the system, there are at most 1,520 index entries in the QT-Quadtree in the (X, T) subattribute-time plane and the same case occurs to the Y-T plane. The value restriction is determined by the distribution of destination points along each side of the rectangle route map and the constant travelling speed assumption.

Parameter	Description	Values Used
MO	The total number of moving objects	5,000, <b>10,000</b> , 20,000, 30,000, 40,000, 50,000, 100,000
SEG	The total segment number of moving plans	27,881, <b>54,808</b> , 110,345, , 220,010, 275,177, 549,657
CHGMO	The number of changed objects	20, 50, 100, 150, 200, 250, 300, 350, 400, 600, 800, 1000, 1200
CHGSEG	The number of changed segment	87, 257, 501, 769, 1026, 1300, 1561, 1822, 2096, 3232, 4365, 5482, 6562
C	Capacity of one quad in the index space	1600

Table 1: Workload Parameters

## 5. Related work

Wolfson et al. [8] considered the management of collections of moving points in the plane. However, their model describes only the current and the expected position of a point in the near future, as represented by a motion vector. The main issue is to determine how often updates of motion vectors are needed to balance the cost of updates against imprecision in the knowledge of positions. Their model does not describe complete trajectories of moving objects and the index maintenance.

Forlizzi et al. [1] presented and formally defined a discrete data model that implements the data types in the abstract model of [2]. They also presented how algorithms can use these data structure. However, their work is mainly on the theoretic aspect.

In [11], a method to index moving objects based on the PMR quadtree was presented. When doing their work, they focused on using the index to support twotypes of range queries called instantaneous and continuous queries. Since the approach requires periodic reconstruction of the index, an efficient algorithm was proposed for index reconstruction. Different from their work, our focus is to improve index update efficiency instead of the requirement of periodic destruction and reconstruction of the index.

Kwon [4] proposed a R-tree based indexing technique called LUR-tree. It can reduce update cost, but the problem is that it does not oriented to the trajectory of moving objects. The LUR-tree seems like general spatial index method, without grasping the “mobility” characteristic of mobile computing.

Tao and Papadias [10] addressed the problem of the indexing and retrieval of moving regions’ past locations by proposing the MV3R-tree. The method does not seem to be applied to future location management.

Kollios et al. [3] employ the Duality method to transform the original trajectory function to a spatial point. Then build the Kd-tree based index that supports the efficient querying of the current and future positions of moving objects. They suggested but has not investigated in much detail, how this approach may be extended to two or higher dimensions. To solve the problem, Simonas et al. [5] proposed a R\*-tree based indexing technique that supports efficient queries. It carries out a projection method to apply the index method to querying the multi-dimensional moving objects. Next, Simonas et al. [6] improved the previous index and update method in order to provide much more efficient location-based service.

In [12], the *plain dead-reckoning (pdr)* method was proposed in which an update is generated to refresh the location of an object and redefine its location function whenever the deviation of its current location is greater than the last update by a pre-defined threshold. In [8, 9], the *adaptive dead-reckoning (adr)* was proposed by extending *pdr*. In [13], the methods were further extended to *disconnected detecting dead-reckoning (dtdr)* to deal with the problem of network disconnection. All the methods proposed above are used to solve the issue “when-to-update”, not mention the issue “how-to-update”.

## 6. Conclusion and future work

The paper addresses the problem of indexing dynamic attributes to support index maintenance efficiently. The spatio-temporal index (FT-Quadtree) is built on future trajectories based on PMR quadtree. There are two main techniques are proposed in our work, one is “sharing trajectory segment” and the other is “reusing index space”. Extensive experiments have been done and the results show that the FT-Quadtree is more effective and efficient in index maintenance and has superiority to the standard quadtree.

The work clarifies two different but related concepts, data update and index update. Data of dynamic attributes change over time and should be updated periodically. Once the data are updated, index built on them should also be reconstructed or updated. While many other works are on “when to update”, our work focuses on “how-to-update” problem. Both of them can reduce the index update cost and further improve the index maintenance. In our experiments, we have simplified the problem: the index update only occurs at the end of one trajectory segment. We will generalize it in our future work.

## Acknowledgement

This research was partially supported by the grants from 863 High Technology Foundation of China under grant number 2002AA116030, from the Natural Science Foundation of China(NSFC) under grant number 60073014, 60273018, and from the Excellent Young Teachers Program of M0E, P.R.C(EYTP). We would like to thank Zhiyong Huang and Beng Chin Ooi for their helpful comments to improve the technical quality and literary style of the paper.

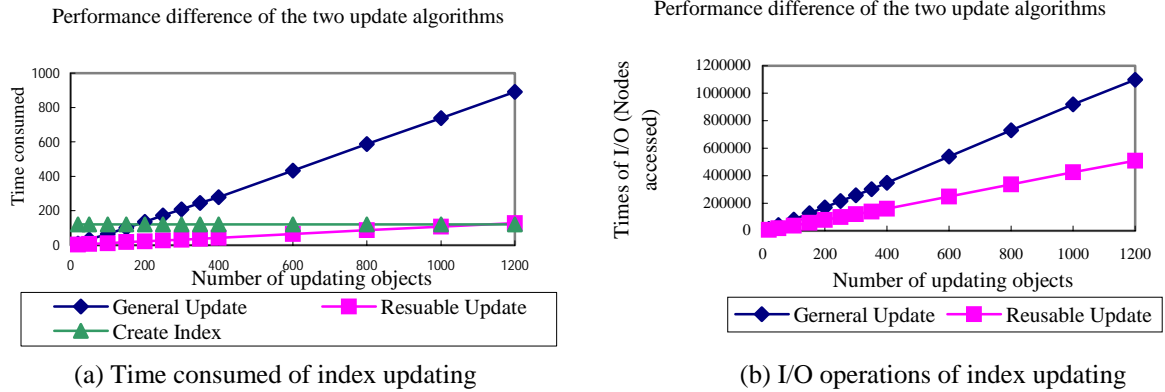


Figure 7. Performance of index algorithms

Number of moving objects	Number of trajectory segments	Number of shared segments in (X, Y, T) index space	Number of shared segments in (X, T)	Number of shared segments in (Y, T)
5,000	27,881	22,727	1,503	1,207
10,000	54,808	37,290	1,509	<b>1,210</b>
20,000	110,345	54,843	<b>1,510</b>	<b>1,210</b>
30,000	164,878	63,647	<b>1,510</b>	<b>1,210</b>
40,000	220,010	68,929	<b>1,510</b>	<b>1,210</b>
50,000	275,177	72,068	<b>1,510</b>	<b>1,210</b>
100,000	549,657	78,311	<b>1,510</b>	<b>1,210</b>

Table 2: Contrast of the amount of original trajectories and share segments

## References

- [1] L. Forlizzi, R. Guting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 319-330, Dallas, Texas, June 2000.
- [2] R. Guting, M. Bohlen, M. Erwig, C.S. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transaction on Database Systems*, 25(1), 2000.
- [3] G. Kollios, D. Gunopulos, and V. J. Tsotras, On Indexing Mobile Objects, In *Proc. of Principles of Database Systems*, pages 261-272, Philadelphia, USA, May 31-June 2, 1999,.
- [4] D. Kwon, Sa. Lee, and Su. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Proc. of 3<sup>rd</sup> International Conference on Mobile Data Management*, pages 113-120, Singapore, January 2002.
- [5] S. Saltenist, and C. S. Jentsent. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 331-342, Dallas, Texas, June 2000.
- [6] S. Saltenis and C. S. Jentsent. Indexing of Moving Objects for Location-Based Services. In *Proc. of 18<sup>th</sup> International Conference on Data Engineering*, pages 463-472, San Jose, California, February 2002.
- [7] H. Samet, Spatial Data Structure, In *Tutorials of 23<sup>th</sup> International Conference on Very Large Data Bases*, pages 131-151, Greece, August 1997.
- [8] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. 13<sup>th</sup> IEEE International Conference on Data Engineering*, pages 422-432, Birmingham, U.K, April 1997.
- [9] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the Uncertain Position of Moving Objects. *Temporal Database: Research and Practice*, Lecture Notes in Computer Science (Springer Verlag), pages 310-337, 1998.
- [10] Y. F. Tao and D. Papadias. The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proc. of 27<sup>th</sup> International Conference on Very Large Data Bases*, pages 431-440, Roma, Italy, September 2001.
- [11] J. Tayeb, O. Ulusoy, and O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3): 185-200, 1998.
- [12] O. Wolfson, S. Chamberlain, S. Dao, and L. Jiang. Location Mangement in Moving Objects Databases. In *Proc. of 2<sup>nd</sup> International Workshop on Satellite-Based Information Services*, pages 7-13, Budapest, Hungary, October 1997.
- [13] O. Wolfson, A. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Database that Track Mobile Units. *Distributed and Parallel Databases*, 7(3): 257-387, 1999
- [14] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. of the 10<sup>th</sup> International Conference on Scientific and Statistical Database Management*, pages 111-122, Capri, Italy, July 1998.