PAPER

ATTI: Workload-Aware Query Adaptive OcTree Based Trajectory Index

Xiangxu MENG[†a)], ***Nonmember***, **Xiaodong WANG**[†], ***Member***, ***and*** **Xinye LIN**[†], ***Nonmember***

SUMMARY The GPS trajectory databases serve as bases for many intelligent applications that need to extract some trajectories for future processing or mining. When doing such tasks, spatio-temporal range queries based methods, which find all sub-trajectories within the given spatial extent and time interval, are commonly used. However, the history trajectory indexes of such methods suffer from two problems. First, temporal and spatial factors are not considered simutaneously, resulting in low performance when processing spatio-temporal queries. Second, the efficiency of indexes is sensitive to query size. The query performance changes dramatically as the query size changed. This paper proposes workload-aware Adaptive OcTree based Trajectory clustering Index (ATTI) aiming at optimizing trajectory storage and index performance. The contributions are three-folds. First, the distribution and time delay of the trajectory storage are introduced into the cost model of spatio-temporal range query; Second, the distribution of spatial division is dynamically adjusted based on GPS update workload; Third, the query workload adaptive mechanism is proposed based on virtual OcTree forest. A wide range of experiments are carried out over Microsoft GeoLife project dataset, and the results show that query delay of ATTI could be about 50% shorter than that of the nested index.

key words: *spatial-temporal database, trajectory, workload-aware, adaptive*

1. Introduction

The popularity of GPS-enabled handsets makes collecting user trajectory data much easier. Trajectories contain abundant information, including users' history positions and current position. Furthermore, some other useful information may be inferred, e.g, users' health status, social relations and so on. Based on those information, many services could be developed, including route planning for tourists, best position finding for advertisement displays, etc. CarTel [1], a project of Massachusetts institute of technology (MIT) has collected trajectory data of about 20 taxis since 2007. CarTel could provide city traffic status query service for citizens, and support individuals to browse their historical trajectories; GeoLife project [2] of Microsoft Asia Research Institute has collected massive trajectory data of hundreds of users since 2007. To use those data effectively, Geolife designed a series of techniques such as the trajectory index [3], background map matching [4] and similarity trajectory mining. With these data, Geolife developed intelligent tour guide system, intelligent driving guide system and a tourist spot search application.

Manuscript received April 25, 2012.
Manuscript revised November 22, 2012.

[†]The authors are with National University of Defense Technology, China.
a) E-mail: yumengkk@qq.com

DOI: 10.1587/transinf.E96.D.643

As the basic resources of many location-based services (LBS), trajectories need to be retrieved frequently by many users in a variety of applications. Spatio-temporal range queries are the most effective trajectory retrieval method, i.e., "return all the trajectories between T1 and T2 within a particular special spatial region". The result sub-trajectories set may be very large when the amount of trajectory data is massive. Therefore, how to find sub-trajectories efficiently and load them into memory for future processing with minimal delay and disk I/O is an urgent problem to be solved. This paper focuses on designing and implementing adaptive trajectory storage and index mechanisms for processing spatio-temporal range query.

2. Related Work and Motivation

2.1 Related Work

Trajectory indexes belong to spatio-temporal indexes which are divided into current (future) position indexes and historical trajectory indexes. According to the underlying model, spatial indexes are divided into free space indexes and network indexes. This article focuses on the trajectory index which belongs to the free space historical trajectory index. Table 1 lists related works belonging to this class.

Free space historical indexes can be further divided into two categories according to building methods, namely space division indexes and R-tree family indexes (data driven index). The major disadvantages of R-tree family indexes are:

Table 1 Free space historical trajectory indexes.

3DRTree [5] (1998)	(R-Tree family) The 3D extension of R-tree.
MV3RTree [6] (2001)	(R-Tree family) Merged 3DR-tree and multi-version R-tree.
SETI [7] (2003)	(space division + R*-tree) Divide space into grids, and every grid cell builds an R*-tree.
BBXTree [8] (2005)	(Time division + B-tree) Divide time into intervals and present trajectory's position by space-filling curve. Every interval builds a B-tree.
CSE-tree [3] (2008)	(Space grid + B-tree) Divide space into grids and based on sub trajectory's end time build a B-tree for every cell.
PIST [9] (2008)	(Space grid + B-tree) Divide space into grids, and every cell build a time index.
QuadTree [10] (2010)	(Quadtree + time index) Build an unbalanced quad tree based on overload, every sub-region covered by a quad tree leaf node builds a time index.

(1) The node split cost is very high when data are inserted frequently; (2) The query process performance is low when interior regions overlap heavily. 3DR-tree and MV3R-tree, which are extensions of 3D version R-tree, are affected more heavily by overlapping caused by uneven data distribution. Actually, trajectory data often update frequency and distribute unevenly. Therefore, almost all indexes belong to space division based index after MV3DR-tree, as shown in Table 1. Test results of literature [1] and [7] show that their indexes have higher performance than 3DR family indexes when processing spatial range queries.

2.2 Motivation of ATTI

Space division indexes are also facing some challenges:

- How to synchronize index temporal and spatial dimensions;
- How a space division adapts to the dynamic change of data;
- The query process cost is closely related with query size, when the query size changes heavily, how to guarantee the performance of query processing;
- Some space partitions which intersect with a query region contain some disk pages whose data does not belong to the query result. How to reduce the additional cost caused by processing such irrelative disk pages.

Existing space division indexes are mainly nested indexes [10], which index the time dimension and spatial dimensions consequently. In other words, nested indexes do not consider temporal and spatial factors simultaneously. Therefore, trajectories adjacent both in time and space may not been stored in adjacent disk pages. Meanwhile, the 3DR trees, which simultaneous index time and space dimensions, are thought to have higher performance for processing spatio-temporal range query theoretically, but that was not the case as Sect. 2.1 shows.

To obtain a relatively high spatio-temporal range query processing efficiency, indexes should index three dimensions (longitude, latitude and time) simultaneously and avoid high maintenance overhead. Fortunately, GPS points received online could be treated as discrete three-dimensional points, and the already existed trajectory data could be split into sub-trajectories. These are all different from 3D objects with fixed shapes and will effectively enhance the efficiency of query and storage. This paper is motivated by this idea and the main contributions are listed below:

- The cost model of trajectory spatio-temporal range query is introduced. Base on such a model, a workload-aware adaptive spatio-temporal OcTree index approach is proposed (Sect. 3);
- A virtual OcTree forest through nodes sharing model is proposed, which could deal with the query adaptively with extremely low costs (Sect. 5);
- Disk page level indexes are built to reduce the overhead of retrieving redundant pages, which are not query results but belongs to the 3-dimentional cube intersecting with the query zone. Furthermore, sub-trajectories are clustered based on trajectory ID and time in disk space to speed up postprocessing (Sect. 5);
- Extensive experiments are carried out based on the Microsoft GeoLife project real data sets and GSTD synthetic data sets. The results show that ATTI can reduce the processing delay by about 50% compared with nested indexes of literature [1] (Sect. 6).

3. Cost Model of Spatio-Temporal Range Query

3.1 Formulation of Trajectory and Spatio-Temporal Range Query

Consider a set containing n trajectories: $TS = \{T_1, T_2, \ldots, T_n\}$, where T_i denotes the i^{th} trajectory:

$$T_i = \{p_{i0}, p_{i1}, p_{i2}, \ldots, p_{ij}, \ldots\}, \tag{1}$$

here, $p_{ij} = \{x_{ij}, y_{ij}, t_{ij}\}$ represents a trajectory point in T_i.

A spatio-temporal query could be expressed as $Q = <s_{ext}, t_{ext}>$, where s_{ext} is the space extension, usually represented by minimum bounding rectangle (MBR) which is defined using lower left and upper right points; $t_{ext} = <t_s, t_e>$ is the time duration determined by start and end time.

3.2 Time Cost for Processing Spatio-Temporal Range Query

Given a spatio-temporal query, the volume of its corresponding three-dimensional (time, space) cube is $V_Q = s_{ext} \times t_{ext}$. Here, the cube is formed by user-defined spatial and temporal extensions of interest, e.g. “Beijing city, from 1-1-2008 to 12-30-2012”.

We assume the trajectory points distribute evenly, and the density of the trajectory points is denoted as D, which indicate how many points exist in a unit cube. Each point occupies K bytes disk space. The number of disk pages occupied by a query result is described as follows:

$$N_{page} = \left\lceil \frac{|V_Q \times D| \times K}{PageSize} \right\rceil, \tag{2}$$

where $PageSize$ is the capacity of a disk page.

Let T_{rot}, T_{seek} and $T_{transfer}$ denote the rotational delay, seek delay and transfer delay of reading one disk page respectively, according to the working mechanism of disk [10]. $T_{transfer}$ are roughly same. To facilitate the following analysis, T_{rot} are set to half of the rotational delay of a circle, and T_{seek} is determined by the number of tracks n between two pages. Let s_i be the page interval, which is the number of pages between two consequent disk pages of the result set. Here, we assume that the number of tracks is proportional to the number of disk pages (in fact, $n = \left\lceil \frac{s_i}{m} \right\rceil$, m is the number of pages stored in a track). Therefore, the average time of reading a disk page, with s_i disk page need

to be move over, is represented as $T_{rot} + s_i \times T_{move} + T_{transfer}$, where T_{move} is the time of moving over one track. Finally, the total time cost of a query can be expressed as:

$$TC_Q = T_{index} + \sum_{i=1}^{N_{page}} (T_{rot} + s_i \times T_{move} + T_{transfer}), \quad (3)$$

where T_{index} is the time to address the result blocks. Therefore, the query performance could be optimized by reducing T_{index}, N_{page} and s_i. The average value of the page intervals and transfer delay of a disk page is defined as follows:

$$A = \frac{\sum_{i=1}^{n} (T_{rot} + s_i \times T_{move} + T_{transfer})}{n}. \quad (4)$$

For a set of N GPS points, the cost is described as:

$$C = T_{index} + A \times \left\lceil \frac{N \times K}{PageSize} \right\rceil. \quad (5)$$

Equation (5) shows that the query delay consists of two parts. The first part T_{index} is affected by many factors, including whether to establish an index, using memory based index or disk based index, computational complexity of the query processing and the computing capability of processors, etc. In this paper, memory based index is used as in [1], assuming that the capacity of the main memory is large enough to store the whole index. Therefore, disk I/O operations, which are difficult to quantify, are not involved in indexing operations. Instead, we focus on the second part, which is often far greater than the first type. Particularly, we focus on reducing the time of disk I/O operations when loading the query results, since the result set of query is often vey large and need to be loaded by many disk I/O operations.

$$C_{I/O} = A \times \left\lceil \frac{|V_Q \times D| \times K}{PageSize} \right\rceil. \quad (6)$$

As Eq. (6) shows, I/O overhead is mainly affected by the distribution and quantity of disk pages. We propose the workload-aware adaptive OcTree which index spatial and temporal dimensions at the same time and also cluster adjacent trajectory data on near disk pages.

4. Workload-Aware Spatio-Temporal OcTree

Based on the query cost model, we use the dynamic spatial and temporal division method to build spatio-temporal OcTree cluster index.

4.1 Dynamically Spatio-Temporal Division

As shown in Fig. 1, an entire cube, whose volume is V, is divided into eight equal octants, and each one can be divided recursively as needed.

Note that spatial dimensions and temporal dimension have different characteristics. The area size of spatial dimension usually does not change, while time dimension is apt to extend (grow) infinitely. In our system, the administrator should define spatial extensions and time range according to real-world needs. For example, we define Beijing city as spatial extensions and five years as the time range (from the begin time of Geolife project 1-1-2008 to the expected end time 12-30-2012) to form the initial cube of our system. When the temporal dimension grows beyond the end time, a new cube using old spatial extension and new time range (e.g. from 1-1-2013 to 12-30-2017) will be created. If a query crosses two cube, our system will divide it into two sub-queries based on the time dimension, keeping spatial dimensions unchanged. ATTI index is used to process the two sub-queries separately, and the final query result is obtained by combining two sub-query results together. This method is different from nested index, such as BBXTree [8] which divides time dimension into intervals and represents trajectories' positions by spatial dimensions index, because it always index time and space simulatenously.

The core of the division is how to dynamically adjust divide depth according to the change of trajectory data to get minimal query processing cost. We will start from the simplest scene: 3D uniform Grid, where the minimal division is named a division unit (grid cell), represented as a three-dimensional cube: $cell = \{r_x, r_y, r_t\}$. Here r_x, r_y and r_t denotes space extensions and time interval respectively. Suppose the density of trajectory points in the cube is also D. The time cost of reading those points from disk to memory is computed as Eq. (7) shows:

$$C_{cell} = A \times \left\lceil \frac{r_x \times r_y \times r_t \times D \times K}{PageSize} \right\rceil. \quad (7)$$

For an arbitrary query $q = \{qr_x, qr_y, qr_t\}$, where $\{qr_x, qr_y, qr_t\}$ are intervals in each dimension. The I/O overhead is:

$$C_q = A \times \sum_{cells} P(q \cap cell) \times C_{cell} \quad (8)$$

where $P(q \cap cell)$ is the probability of the random query region intersecting with a cell. This probability depends on the spatial and temporal extents of both the query and the cell. Clearly, a query region intersects a cell not only when its center falls within the boundaries of the cell, but also intersects the cell when its center falls just outside of the cell, up to a distance $\frac{qr_x}{2}$ and $\frac{qr_y}{2}$ of the spatial edges of the cell, and up to $\frac{qr_t}{2}$ outside the temporal edges [1]. Neglecting border affects that happen at the edges of the region (outside of which queries would not be allowed), we denote the probability of a random query q intersecting a given cell as Eq. (9).

$$P(q \cap cell) = \frac{(qr_x + r_x)(qr_y + r_y)(qr_t + r_t)}{V}. \quad (9)$$

Seen from Eqs. (7), (8) and (9), the query overhead is mainly affected by the query size, the grid unit size and the data density D. We discuss how to determine the optimal grid cell size first. Suppose the query size is already given,

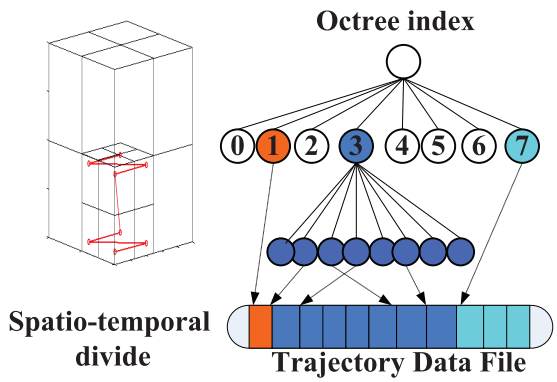


Fig. 1 OcTree index and disk file structure.

and the optimal grid cell size is

$$cell_{opt} \approx argmin_{cell}(C_q). \tag{10}$$

As Eq. (10) shows, the optimal division is closely related to trajectory data density D. However, trajectory workload is not evenly distributed, and is updated frequently. The insertion of new trajectory points will affect data distribution dynamically. Obviously, the static division could not always guarantee the minimal query cost. Therefore, the optimal division should be changed adaptively according to the trajectory data. We will introduce workload adaptive spatio-temporal OcTree index in the next section.

4.2 Data Structure of Adaptive OcTree

Basic idea: as shown in Fig. 1, the data-intensive regions are divided into more cells. On the contrary, the data-sparse regions are divided into less cells.

An OcTree based space division example is shown in Fig. 1. The leaf nodes point to the addresses of disk blocks which store trajectory points belonging to the nodes' spatio-temporal cube. When adding a new point, each division unit may be adjusted dynamically either by: (1) Splitting (divide the unit into eight new sub-units); (2) Merging (Merges the unit with seven other sibling nodes). The criterion for deciding whether to perform the operation or not is that whether the operation achieves lower query overhead than current division.

Figure 2 shows the data structure of an OcTree leaf node. Each OcTree node has a variable *PointNumber* which indicates the number of GPS points belongs to the corresponding subregion. When splitting a node, the numbers of GPS points contained by its eight octants are computed and stored in the *ChildrenPointNumber* array. Meanwhile, the *SbilingPointNumber* array is used to save the numbers of GPS points contained by its seven sibling octants.

Given a query, we can calculate the probability of each cell intersecting with it according to Eq. (9), and further compute the I/O overhead of processing this query as Eq. (11). Let P_i be the number of points which fall on its i^{th} suboctant, which is stored in the *ChildrenPointNumber*

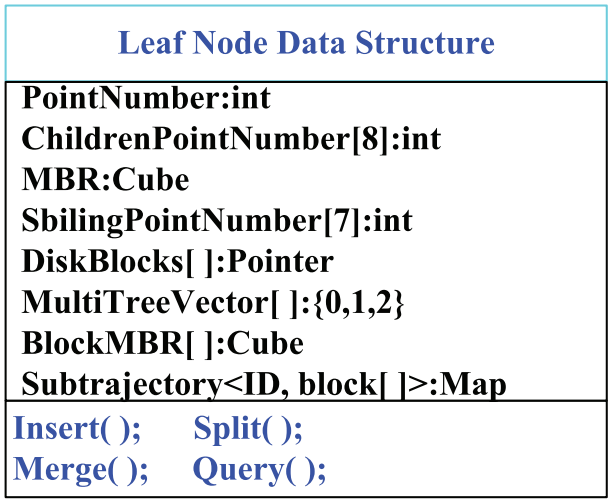


Fig. 2 The structure of OcTree node.

array. Please note that the variable *PointNumber* is the sum of the eight elements of *ChildrenPointNumber*.

$$C_q = A \times P(q \cap cell) \left\lceil \frac{K \times \sum_{i=1}^{8} P_i}{PageSize} \right\rceil \tag{11}$$

When inserting a new trajectory point, we first compute the overhead of processing the same query in the virtual **splitting** division. Supposing the node is divided into eight suboctants, the overhead after splitting is the sum of eight virtual sub-nodes' query process overheads as below:

$$P(q \cap soncell) = \frac{\left(qr_x + \frac{r_x}{2}\right) \times \left(qr_y + \frac{r_y}{2}\right) \times \left(qr_t + \frac{r_t}{2}\right)}{V}$$
$$VirtualC_q = \sum_{i=1}^{8} A \times P(q \cap soncell) \left\lceil \frac{K \times P_i}{PageSize} \right\rceil \tag{12}$$

here *soncell* is used to denote the virtual sub-node corresponded temporal-spatio space, whose volume is $\frac{cell}{8}$ of the current cell. Only when $VirtualC_q < C_q$ will a cell be divided (**splitting** operation) to minimize the query overhead.

Next, we compute the overhead of processing the same query in the virtual **merging** division, and compare the overheads before and after a merge operation to determine whether the merge operation is valuable. Supposing the eight nodes are merged into one sup-octants, the overhead after merging can be computed as below:

$$P(q \cap parentcell) = \frac{(qr_x + 2r_x) \times (qr_y + 2r_y) \times (qr_t + 2r_t)}{V}$$
$$ParentC_q = A \times P(q \cap parentcell) \left\lceil \frac{K \times \sum_{i=1}^{8} PSible_i}{PageSize} \right\rceil \tag{13}$$

here *parentcell* is used to indicate the merged node corresponded space, whose volume is $8 \times cell$ of the current cell. Meanwhile, the sum of overheads of eight sibling nodes before merged is $LevelC_q$.

$$LevelC_q = \sum_{i=1}^{8} A \times P(q \cap cell) \left\lceil \frac{K \times PSible_i}{PageSize} \right\rceil, \qquad (14)$$

where $PSible_i$ indicates the i^{th} sibling nodes. Only If $ParentC_q < LevelC_q$ will a cell be merged with its sibling nodes.

To avoid the jitter phenomenon, i.e., split and merge back and forth, a jitter parameter a is introduced into our system. Only when $VirtualC_q \times (1 + a) < C_q$ or $ParentC_q \times (1 + a) < LevelC_q$ will the **splitting** or **merging** operations be performed. This could speed up the building process of the index. Jitter parameter is a universal scheme that could improve the index building speed, but it is closely related to the workload of specific applications. The proper value of jitter parameter should be decided by the system administrator according application context.

When a trajectory point is inserted into the OcTree, the insertion algorithm of the nodes are invoked recursively from the root (top) node to the children nodes, until a leaf node is reached. Then this point is saved to the leaf node's corresponding disk block. The insertion algorithm is shown in algorithm 1. As aforementioned, the leaf node may be adjusted dynamically by **splitting** and/or **merging** whenever the operation achieves lower query overhead than current node. In algorithm 1, Line 5 is the **splitting** operation, which creates eight new children nodes based on the leaf node. Next, reinsert the points belong to this node into the root node. Line 8 is the **merging** operation, the of algorithm 1 shows that the merge operation will do, which inserts points belonging to itself and the eight siblings into the parent node, and sets the parent node's children to be null.

Both the **splitting** and **merging** operations will introduce disk block rewrite operations/costs. Three steps are involved. First, read all blocks belonging to the leaf nodes which need changing to memory. Second, insert new GPS points and reorganize the new nodes. Third, write all the data of the new nodes into adjacent disk blocks. In fact, the new GPS points are arrived with time elapsed, which makes that only a little new nodes in a time interval need splitted and merged frequently. Furthermore, the mechanism of write buffer helps reduce the cost of rewrite greatly.

The above methods can adjust the structure of OcTree according to the overhead change as new points arrive, in a **workload adaptive** way. Currently, query sizes are assumed as already known, however, it is impossible to predict the sizes of all users' queries. Some researchers use average size or locality average size defined by system administrators as real query size, and build only one index for all users' queries. Those methods can not provide optimal query processing performance for every query. We will discuss how to implement query adaptive index for all users in the next session.

Algorithm 1 Insert-Method of nodes

```
Input: GPSPoint: Point, OcTree-Node: Node
Output: Whether insert success
Variables: Costs according Eqs. (11), (12), (13), (14):
Cost, VirtualCost, ParentCost, LevelCost
1: if (Node is leaf node) then
2:     add Point into Node;
3:     compute Cost, VirtualCost, ParentCost, LevelCost;
4:     if (VirtualCost < Cost) then
5:         call Split-Method;
6:     end if
7:     if (ParentCost < LevelCost) then
8:         call Merge-Method;
9:     end if
10:    return true;
11: else
12:    get SubNode to which Point belongs;
13:    call Insert-Method of SubNode;
14: end if
```

5. Query Adaptive Virtual Forest

5.1 Impact of Query Size to Index

Workload-aware OcTree can dynamically adjust its structure according to the changing of workload. However, another factor of cost formula, namely the query size, is still indicated by a presetting value [1], [9]. For a query whose size is significantly different from the presetting value, the cost to process it will be high. In Ref. [1], a weighted average value, other than the average value given by the administrator, is used. This method assumes that query size has local similarity, and the system maintains a weighted average value of dynamically arrived queries. When the variation of the average value exceeds some threshold, the index will be rebuilt using the new query size average value. Obviously, this assumption does not hold in a multi-user scenario. For example, one user's query may be: "Get all the trajectories which go through an intersection within the last hour". While another user's query may be: "Get all the trajectories within the whole downtown this year". As mentioned above, static or dynamic average values of the query sizes are just approximation of all query sizes. This does not mean that all queries can be effectively processed using a uniform OcTree. As shown in Fig. 3, trees built according to the minimum, average and maximum query size are significantly different.

Figure 3 shows the status when the 10^{th} point of right trajectory is inserted into the OcTree, where numbers in circles indicate the number of points belong to this node, and numbers besides circles are called "node indicator". Here,

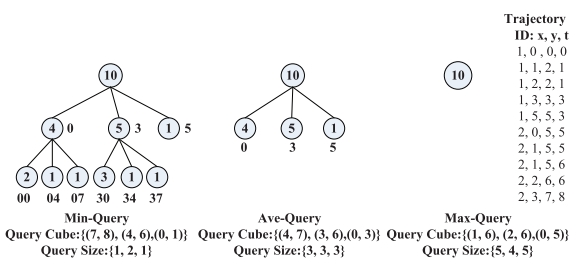


Fig. 3 The impact of different query sizes to OcTree.

Table 2 The case using different OcTrees processing different queries.

-	Min-query	Ave-query	Max-query
Min-tree	Null	07, 30, 34, 35	00, 04, 07, 30, 34, 5
Ave-tree	3	0, 3	0, 3, 5
Max-tree	Top	Top	Top

the page size is set to 5, and the initial cube is $\{(0, 8), (0, 8), (0, 6)\}$ which means spatial coordinates range from 0 to 8, and time interval ranges from 0 to 6. Figure 3 shows the three OcTrees built according to the minimum, average and maximum query size respectively. Leaf nodes accessed by each query are shown in Table 2.

As the first column of Table 2 shows: Using the Min-tree executes the Min-query needs to access three interior nodes, and directly returns "null"; Using Ave-tree executes the Min-query needs to access two interior nodes and all five data points belong to the second branch; Using the Max-tree executes the Min-query needs to access the root node and all leaf nodes. Using the min-tree executes Min-query is optimal, only need to access three interior nodes. The third column shows that using the Min-tree executes the Max-query needs to access 10 interior nodes, and 9 data are returned; using Ave-tree executes the Max-query needs to access 4 interior nodes, and 10 data are returned; using Max-tree executes Max-query just needs to visit the top node, and all 10 data are returned. Ave-tree gets the same number of data as Max-tree gets, but it visits four more interior nodes.

The above discussion shows that using the proper query tree to process query can get the best performance (such as the diagonal items in Table 2 show). Using a larger query tree to process a smaller query will return too much redundant data, and using smaller query tree to process large query need access more interior nodes which also introduces overhead. The ideal solution of this problem is to build a query tree for each query size. This solution is equivalent to re-establish an index for every query, almost need to traverse the entire data set, and even more expensive than sequential search. Through the above analysis, the best method to implement query adaption should be the one that can keep the query efficiency of multiple indexes, and reducing the storage and maintenance cost of multiple indexes at the same time.

5.2 Virtual OcTree Forest

Basic idea: Each tree is built by space division. The result is that a tree branch corresponding to a deep division must contain the branches built by shallower division. Therefore, more trees can share nodes between each other, which can reduce the maintenance and storage costs.

The administrator should set the minimal query size, the maximum query size (can be absolute value or percentages of entire cube) and the number of virtual trees (n_{tree}), according to application context at the system initial time. Using those three parameters, the system can calculate a set of query sizes and save them in a vector. The minimal query is $Q_{min} = <r_{min}, t_{min}>$, the maximum query is $Q_{max} = <r_{max}, t_{max}>$, so the $\Delta r = \frac{r_{max}-r_{min}}{n_{tree}-1}$ and $\Delta t = \frac{t_{max}-t_{min}}{n_{tree}-1}$. The k^{th} element in the query size vector is $Q_k = <r_{min} + (k-1) \times \Delta r, t_{min} + (k-1) \times \Delta t>$. System will build a bunch of virtual OcTrees (these are not real OcTrees as shown in Fig. 4, thus are named virtual index tree) according to each query size in vector, and form a query forest. In order to reduce maintenance and storage costs, all trees share nodes between each other, building a "virtual" OcTree forest.

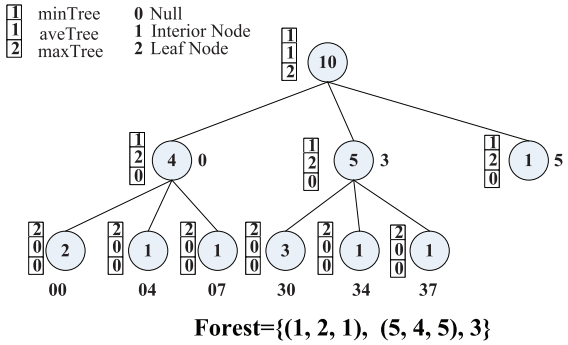


Fig. 4 The structure of virtual forest.

The minimal or maximal query size does not correspond to the OcTree whose branch is the deepest according to formulas (11), (12), (13), (14). It is difficult to determine which query size is corresponding to the tree which contains all the other trees' nodes. Therefore, in order to generate an OcTree which can contain all the virtual nodes, we need to modify the native **splitting** and **merging** operations in algorithm 1 as follows:

- As long as there is a node of any virtual tree needs splitting, the node begins to divide;
- Only when all the virtual tree nodes, corresponding to the same spatial-temporal cube as this node, need merging, the nodes begin to merge.

Obviously, the OcTree built according to this strategy must be the one whose branches are all the deepest. We call this OcTree a real OcTree and cluster data on disk using this tree. While the other query sizes generated virtual query tree are secondary indexes, only used for query processing, called virtual OcTree.

As shown in Fig. 4, the vector on the left of each node is the "MultiTreeVector", which is described in "Data Node Structure" (Fig. 2). Figure 4 shows the virtual forest containing all the three query OcTrees in Fig. 3. The "0" in "MultiTreeVector" indicates that the node does not belong to the virtual query tree corresponding to the element; "1" indicates that the node in the corresponding virtual query tree is "interior nodes"; "2" indicates that the node in the corresponding virtual query tree is "leaf node". For example, the first vector elements are used to represent the ave-tree in Fig. 3. The second element of top node's vector is "1" indicates that the node in the original tree is interior nodes; vector value of the second layer nodes is "2" indicating that it is a leaf node in the ave-tree; vector value of

Algorithm 2 RealInsert-Method of real-tree nodes

Input: GPSPoint: ***Point***, OcTree-Node: ***Node***
Output: Whether insert success
Variables: Whether split: ***SplitIndicator***
Whether merge: ***MergeIndicator***
Index of Virtual tree: ***index***
Costs of index-th virtual tree according Eqs. (11), (12), (13), (14): ***Cost[index]***, ***Virtualcost[index]***, ***ParentCost[index]***, ***LevelCost[index]***

1: ***SplitIndicator***= false;
2: ***MergeIndicator***= true;
3: **if** (***Node*** is leaf node) **then**
4: add ***Point*** into ***Node***;
5: **for all** index **do**
6: compute ***Cost[index]***, ***VirtualCost[index]***, ***ParentCost[index]***, ***LevelCost[index]***;
7: **if** (***VirtualCost[index]*** < ***Cost[index]***) **then**
8: ***SplitIndicator***= true;
9: **end if**
10: **if** (***ParentCost[index]*** > ***LevelCost[index]***) **then**
11: ***MergeIndicator***= false;
12: **end if**
13: **end for**
14: **if** (***SplitIndicator*** is true) **then**
15: call ***Split-Method*** to create 8 ***NewNode***;
16: **for all** index **do**
17: set ***MultiTreeVector[index]*** of ***NewNode*** to 0;
18: **end for**
19: get all points of ***NODE*** to do ***reinsert-method***;
20: **end if**
21: **if** (***MergeIndicator*** is true) **then**
22: add all the points belong to ***Node*** and its 7 ***SiblingNode*** to her ***ParentNode***;
23: set ***ChildrenPointer*** of ***ParentNode*** to null;
24: **for all** index **do**
25: **if** (***MultiTreeVector[index]*** $\neq$ 0) **then**
26: set ***MultiTreeVector[index]*** of ***ParentNode*** to 2;
27: **end if**
28: set ***MultiTreeVector[index]***= 0;
29: **end for**
30: **end if**
31: **return** true;
32: **else**
33: get ***SubNode*** to which ***Point*** belongs;
34: call ***RealInsert-Method*** of ***SubNode***;
35: **end if**

the third layer is "0" indicates that this node does not belong to the ave-tree. What's important is that the disk addresses of trajectory data agree with the order of the splitting string. So the data pointer of a virtual tree node only points to the first physical page of all the data contained by corresponding leaf nodes in the real tree. And every virtual node records the number of disk pages occupied by all the data cover by the spatial-temporal cube corresponding to this node. Then every node can ensure that all data covered by its corresponding cube are stored in adjacent disk pages. Obviously, the insert, splitting and merging operations in the workload-aware query tree should be modified.

As shown in algorithm 2 and 3, When a new point is added, the insert algorithms of both the real tree nodes and the virtual tree nodes are invoked. The cost values used in virtual insert algorithm is acquired from the real tree nodes, so the insert algorithm of the real tree nodes must be invoked before the insert algorithm of the virtual trees. When a virtual tree satisfies the splitting condition, the corresponding node in the real tree must have been split and its 8 new sub-nodes must have been created already. Therefore, the virtual splitting and merging operations only change the virtual vector, but will not create or clear any real node (The depth of real branches must be deeper than or equal to the virtual tree).

Algorithm 3 VirtualInsert-Method of virtal-tree nodes

Input: GPSPoint: ***Point***, OcTree-Node: ***Node***
Output: Whether insert success
Variables: Index of Virtual tree: ***index***
Costs of index-th virtual tree according Eqs. (11), (12), (13), (14): ***Cost[index]***, ***Virtualcost[index]***, ***ParentCost[index]***, ***LevelCost[index]***

1: **if** (***Node.MultiTreeVector[index]==2***) **then**
2: **if** (***VirtualCost[index]*** < ***Cost[index]***) **then**
3: set ***Node.MultiTreeVector[index]=1***;
4: set all ***SubNode.MultiTreeVector[index]=2***;
5: **end if**
6: **if** (***ParentCost[index]*** > ***LevelCost[index]***) **then**
7: set ***Node.MultiTreeVector[index]=0***;
8: set all ***SiblingNode.MultiTreeVector[index]=0***;
9: set ***ParentNode.MultiTreeVector[index]=2***;
10: **end if**
11: **return** true;
12: **end if**
13: **if** (***Node.MultiTreeVector[index]==1***) **then**
14: get ***SUBNODE*** to which ***POINT*** belongs;
15: call ***VirtualInsert-Method*** of ***SUBNODE*** with ***index***;
16: **end if**

5.3 Data Organization According Trajectory ID

The GPS points belonging to a leaf node are sorted by trajectory ID and time, and stored into disk. In the disk space pointed by a leaf node, all points of a trajectory will be sorted according time in same blocks to be a sub-trajectory. As shown in Fig. 2, every leaf node has a map structure to store the trajectory ID and disk blocks of a sub-trajectory. This data structure can speed up trajectory display and post-processing through connecting all sub-trajectories belong a same ID. Furthermore, ID based filter of spatio-temporal range query result could be supported.

5.4 Disk Page Level Index

When generating the real tree, trajectory data covered by a spatial-temporal division unit may be stored on multiple disk pages. However, the disk pages corresponding to a division unit which intersects with the query region should be loaded into memory for further processing. As Fig. 5 shows, if a query region (pink) intersecting with a division unit (gray), the gray disk pages in the data file should be loaded into memory. In fact, not all the spatial-temporal cubes corresponding to the disk pages intersect with the query region, so reading all three to memory is redundant. It is reasonable to record every page's spatial and temporal cube in the index as it can avoid the cost of loading unrelated pages.

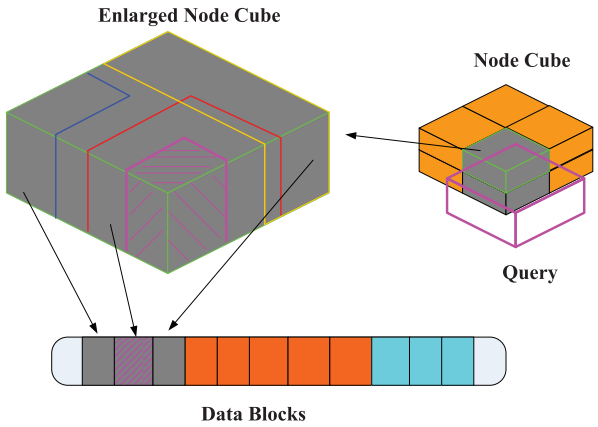


Fig. 5 Disk page level index sketch.

Spatial-temporal cube description: The method adds BlockMBR array (see Fig. 2) into the data structure of a leaf node to preserve the spatial and temporal scales of every disk page. This block level MBRs can be compared with query MBR to decide whether this block contains points of the query result.

5.5 Adaptive Query Algorithm

First, we should determine the most suitable virtual query tree before processing a query using an OcTree forest. Then call the virtual query algorithm (algorithm 4) of the root node in the optimal virtual OcTree to get all the disk pages whose spatio-temporal cube intersects with the query region.

The optimal virtual OcTree selection: search query vector to get the most similar element with query region, and its corresponding virtual index tree is the optimal one. This paper uses Euclidean distance, which is the square root of spatial and temporal dimensional intervals, to find the most similar one. The subscript (parameter "index") of the most similar one in vector is used as the input of virtual query. The similar virtual tree calls the virtual query algorithm of its top node. The virtual query algorithm of each node in a virtual tree is shown in algorithm 4.

Algorithm 4 Query-processing Method of virtual node

```
Input: Virtual-Node: Node, Spatio-temporal region: Cube
Output: Disk pages: Vector
Variables: Index of Proper Virtual tree: S
if (Node.MultiTreeVector[S]==2) then
    for all (DiskPage belongs to Node) do
        if Region of DiskPage intersect with Cube then
            add DiskPage to Vector;
        end if
    end for
    return Vector;
else
    if (Node.MultiTreeVector[S]==1) then
        for all i=1:8 do
            if Region of ChildrenNode[i] intersect with Cube then
                call QueryProcessing-Method of ChildrenNode[i];
            end if
        end for
    end if
end if
```

6. Performance Analyses

6.1 Experiment Environment

Extensive experiments have been carried out to evaluate the performance of the proposed adaptive OcTree indexes. The experimental setting, including the hardware and software configurations, the metrics, etc, are described as follows:

1. Hardware configuration: A four-core 2.27 GHz Intel i3 CPU with 2 G RAM and a 320 G 5400 RPM hard disk whose page size is 4 KB (4096 Byte).
2. Software configuration: Ubuntu 11.04 (Linux 2.6.16). All indexes are implemented using JAVA 1.6 without optimization by multi-threads technology (CPU usage ratio is always around 25%). The available memory of JVM is 1 G. Data structures used while implementing indexes, including vector, array, map and set, are all from package java.util.
3. Indexes to be compared: Nested index implemented in Cartel project [1], which built a quadtree according to spatial division, and built a temporal index for data within every space cell. Performance evaluation of literature [1] shows that nested index is superior to two-dimensional grid, basic two-dimensional R-tree indexes and clustering (ClustSplit) index [12]. Nested index in Cartel project is also superior to other existing nested indexes, including the SETI index. Meanwhile, the performance of SETI is higher than 3DR-tree index [7]. Therefore, a comparison of our work with nested index in Cartel project is enough.
4. Performance metrics:
   - Time and memory overhead of index creation;
   - Time delay of processing spatio-temporal range query;
   - Disk I/O times during query processing.
5. Effect of buffer: To avoid the impact of cache to test results, we use */proc/sys/vm/drop_caches* to clear the buffer and cache after each query.
6. Dataset: The synthetic datasets are built using GSTD [11], and the real dataset is obtained from Geolife project which collected GPS trajectory data of 165 users from 2008 to 2009, including 23 million GPS points, amounts to 1.02 GB. In the Geolife project, the spatial region mainly covers Beijing and its surrounding areas, but some users' activities are around the whole area of Europe and the Americas. We normalize the spatial and temporal interval to a cube whose sides are all one unit.
   - Synthetic datasets: The initial distribution and moving mode of users are all random, snapshot

Table 3 Index size (KB) and index time (ms, Values in brackets) of synthetic datasets.

Query	Min-query		Ave-query		Max-query	
Type	ATTI	Nested	ATTI	Nested	ATTI	Nested
GSTD-1	2.3(51.3)	3.57(46.87)	2.3(50.1)	3.6(46.9)	2.3(50.0)	3.9(46.8)
GSTD-2	12.8(255)	35.6 (226)	12.8(256)	38.3(234)	12.8(256)	39.0(235)
GSTD-3	20.6(472)	68.1(469)	20.6(471)	70.8(477)	20.6(471)	71.5(478)

Table 4 Index size (KB) and index time (ms, Values in brackets) of Geolife datasets.

Query	Min-query		Ave-query		Max-query	
Type	ATTI	Nested	ATTI	Nested	ATTI	Nested
G-25%	12.63(282)	14.07(445)	12.63(284)	13.27(445)	12.62(282)	12.91(452)
G-50%	23.64(589)	26.98(915)	23.63(580)	27.04(909)	23.63(580)	26.72(526)
G-75%	35.02(883)	40.72(799)	35.02(884)	41.17(800)	35.03(887)	41.05(807)
G-100%	46.4(1180)	54.5(1078)	46.4(1181)	54.0(1091)	46.4(1194)	54.6(1083)

number is 10000, and the number of users in GSTD is set to 100, 500 and 1000 respectively. Three datasets are built: GSTD-1; GSTD-2; GSTD-3. The numbers of GPS points in each dataset are 88.0245 million, 440.0807 million and 880.1098 million respectively.

– GeoLife dataset: Four datasets are built by separately cut 25%, 50%, 75% and 100% of the first 20 millions GPS points from the original dataset. They are named GeoLife-25%, GeoLife-50%, GeoLife-75% and GeoLife-100% respectively. The numbers of trajectory points are 5 million, 10 million, 15 million and 20 million, respectively.
– Query dataset: Randomly generate three spatio-temporal query datasets whose spatial region area are separately 0.01, 0.05 and 0.25 (The whole area is 1), and length/width is smaller than 4 and bigger than 1/4. Their time interval is separately 0.1, 0.23 and 0.5 which are square root of the spatial area. So the volume of each query is 0.1%, 1% and 12.5% of the entire spatio-temporal cube. Each dataset contains one hundred queries, which are named query-1, query-5 and query-25. The fourth dataset is query-random dataset which contains one hundred queries too, but the spatio-temporal range of each query is changing randomly from 0.00001% to 12.5%. We use the average processing delays of 100 queries in one dataset as a standard for different indexes to compare with.

6.2 Performance Analysis of Workload-Aware Adaptive OcTree Forest

The workload datasets used in this section is GeoLife-25% and query datasets is "query-random".

1. Impacts of the number of virtual trees and page level index to performance of ATTI. As Fig. 6 shown, query processing delay using virtual OcTree forest (Forest: circle marker) decrease about 5%–10% than the delay of using native spatio-temporal OcTree (Join: square marker) which is built based on query size 10%. But there is only 5% reduce of query delay as the number of OcTree changes from 2 to 20. This result shows that the number of virtual trees has a little effect on query performance. The reason is that ATTI is memory based index. In this experiment, the native OcTree only has 820 leaf nodes and the real tree in forest has 2360 leaf nodes, which means using query vector replace an average value can bring depth division of spatio-temporal cube. From analysis in Sect. 5.1 we know that the best benefit of the proper query trees is accessed interior nodes reduced. Here, this benefit is not obvious because nodes of ATTI are always in memory. It will be obvious when storing the index into disk as dataset in-

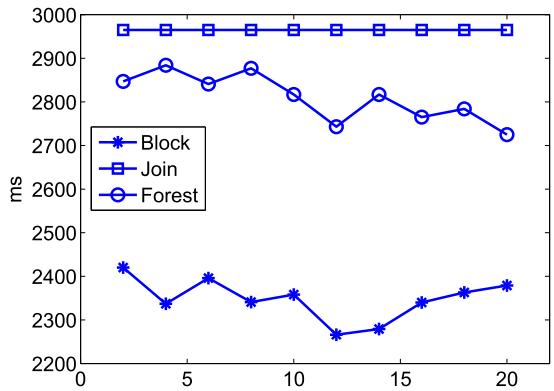


Fig. 6 The number of virtual trees vs. query delay.

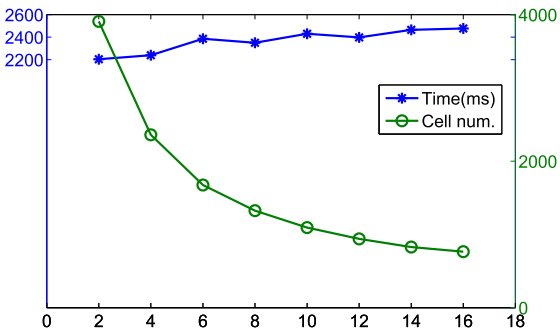


Fig. 7 Impact of disk page level index.

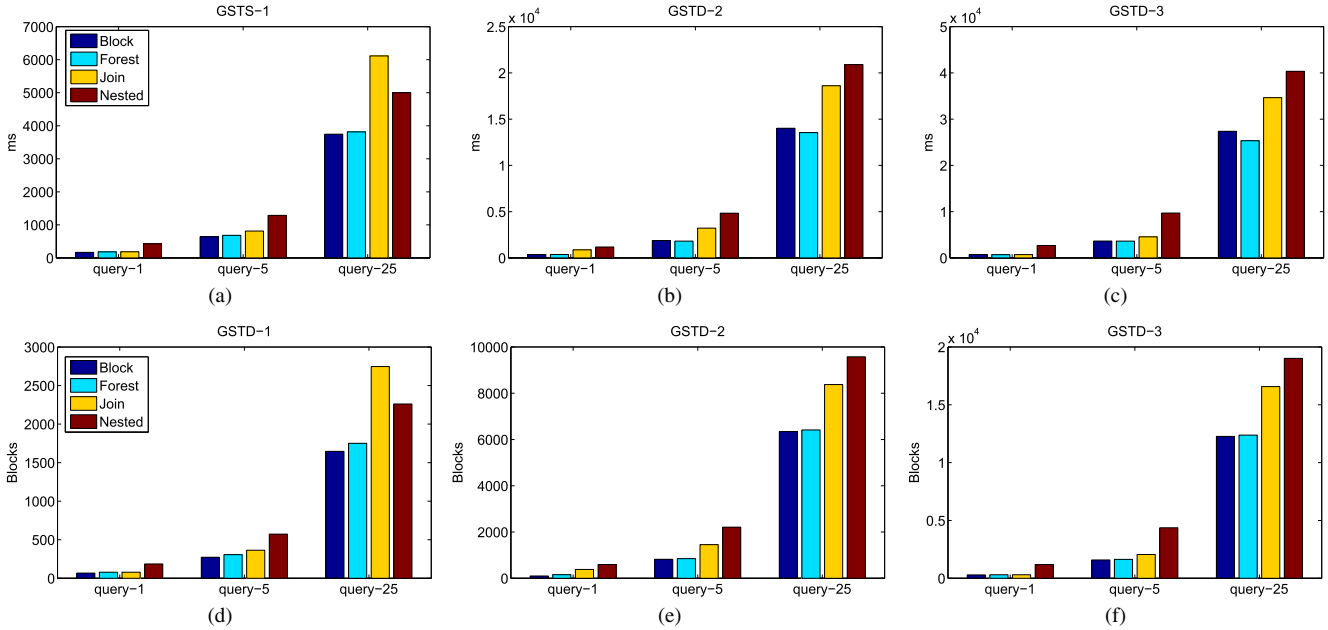


Fig. 8 Disk I/O and delay with synthetic datasets.

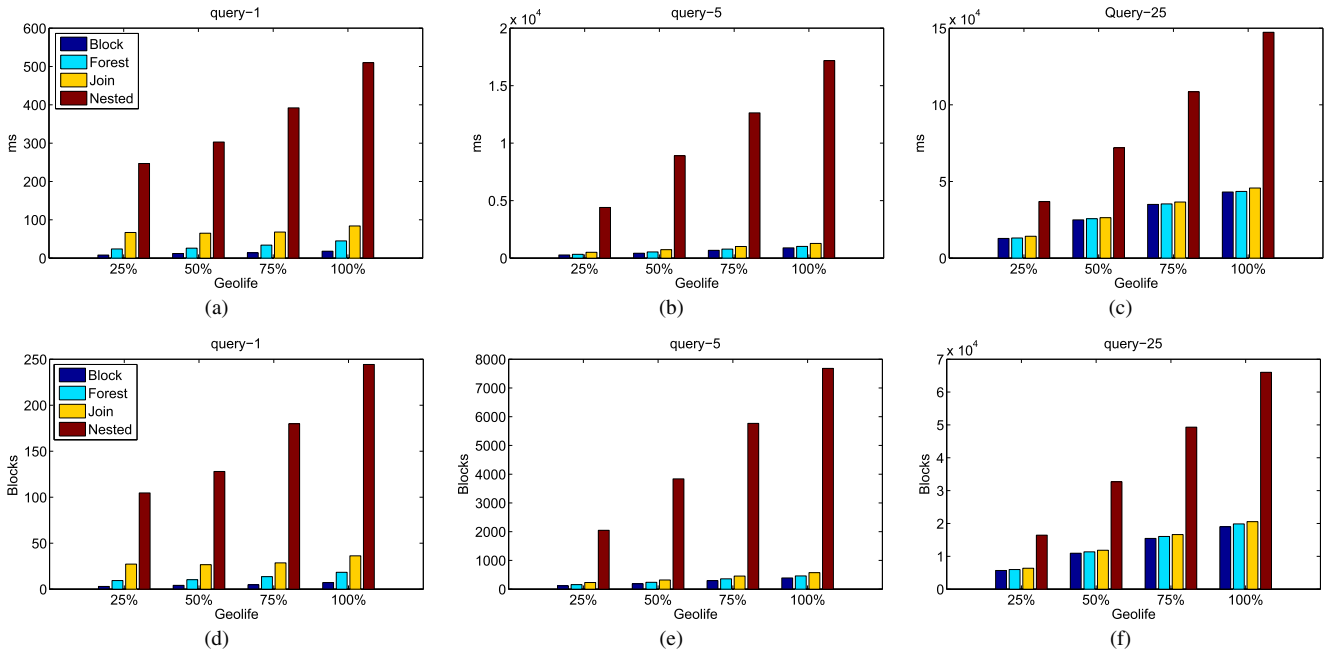


Fig. 9 Disk I/O and delay with Geolife datasets.

creases continuously. In addition, as shown in Fig. 5, the disk page level index (Block: star marker) can reduce about 20% of the query processing delay.

2. Impact of disk page size to performance of ATTI. As shown in Fig. 7, the number of division units (green curve with circle marker) declines from 3960 to 764 as page size increases from 1 kB to 16 kB. This variation doesn't bring dramatic changes to query processing performance (blue curve with star marker). The changes of query processing delay are within only 20%. Thus, we set page size to 4K in the comparison experiment as section 6.4 shows, which will not change the results of the comparisons.

6.3 Comparison Experiments

1. Time and space overhead for creating different indexes. As Tables 3 and 4 show, the time costs of creating (include reading GPS points from disk) ATTI and nested index described in literature [1] are almost the same, while ATTI index only consumes two-thirds memory

space of nested index. ATTI can index 20 thousands of GPS points in one second, which means a PC can process 20 thousands of people's location update when the update frequency is once per second. The index of 20 millions points (the whole Geolife dataset) only uses 50 MB memory space, which means it can always be stored in the memory. Actually, there will be million of users in a real product environment, and the memory will be insufficient to store all points. There are also many methods to solve this problem, such as distributed shared memory and dynamic backup and loading mechanism, which are all commonly used methods in large enterprises. The discussion is beyond the scope of this article because this paper only focuses on data structure and query processing method of ATTI.

2. Query processing performance comparison with synthetic datasets In Fig. 8, brown bars ("Nested" legend) indicate nested index, yellow bars ("Join" legend) indicate OcTree index (jointly index spatial and temporal dimensions), light blue bars ("Forest" legend) indicate OcTree index with adaptive virtual forest mechanisms and dark blue bars ("Block" legend) indicate ATTI index (OcTree index with active virtual forest and disk level mechanisms).
   When the data size is small (GSTD-1), the delay of processing a query with a wide range based on the native OcTree (upper left sub-figure) is higher than the nested index. This may be due to the too large gap between query size (10%) used when establish OcTree and real query size (50%). The query performance improved significantly after using the virtual forest, which proves the efficiency of the virtual forest. The times of disk I/O (yellow column) and query latency using native OcTree are both less than nested index when the data size is large (GSTD-2, GSTD-3). Virtual Forest can reduce 30 percents delay compared with the native OcTree. When the data is evenly distributed, the page-level index does not bring significantly performance improvement. On the contrary, the delay slightly increases because the increase of computational cost.
3. Query processing performance comparison with real datasets. Compare Fig. 8 and Fig. 9, we can find that the performance improvement with real dataset, which is brought by simultaneously index spatial and temporal dimensions, is higher than that with uniform distribution of the synthetic datasets. In dealing with smaller range queries, the delay of ATTI is only five percent of the nested index. This result is rational in the sense where the duration of time is large (two years in Geolife project) for the first level index in the nested index only considers the spatial dimensions and ignores the time dimension. Therefore, it is reasonable that the nested index has lower performance than join index when processing queries with smaller time intervals. Comparing of dark blue bars and brown bars of Figs. 8 and 9, we can conclude that the workload-aware mechanism has played a good role while using real datasets. The query processing time declines from 30% (query-1), 50% (query-5) and 80% (query-25) of nested index to 5%, 10% and 35% when using real dataset.

7. Conclusion

R-tree based indexes have a variety of applications because of their ability to deal with points and polygons. However, R-tree is not suitable to index highly dynamical and unevenly distributed data. Therefore, R tree and its variants have lower performance than space-division indexes when index trajectory data changes frequently and distributes unevenly. Currently, almost all space-division indexes belong to nested indexes, which do not simultaneously index spatial and temporal dimensions. Meanwhile, the space-division indexes face the problems of workload and frequent variation of query sizes.

This paper extends two-dimensional space-division indexes to a three-dimensional version which implements simultaneous index of spatial and temporal dimensions, dynamically adjusts space division according to query processing cost model to realize workload adaptation, implements query size adaptation based on virtual forest that shares nodes between different trees. Experimental results show that workload-aware query adaptive spatial and temporal synchronous index can effectively improve the performance of spatio-temporal range query processing. However, when the trajectory dataset is large, a query processing process of a wider spatio-temporal range may take tens of seconds, which is beyond the tolerance of online users. In the future, we will design distributed storage and parallel processing platform for massive trajectory data to reduce query processing time.

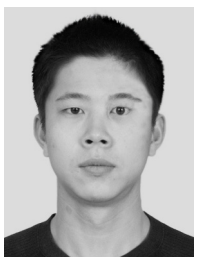
References

[1] P.C. Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," Proc. International Conference on Data Engineering (ICDE), Long Beach, CA, pp.109–120, 2010.
[2] Y. Zheng, X. Xie, and W.Y. Ma, "GeoLife: A collaborative social networking service among user, location and trajectory," IEEE Data (base) Engineering Bulletin - DEBU, vol.33, no.2, pp.32–39, 2010.
[3] L.H. Wang, Y. Zheng, X. Xie, and W.Y. Ma, "A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval," Proc. Mobile Data Management, pp.1–8, 2008.
[4] Y. Lou, X. Xie, C.Y. Zhang, W. Wang, and Y. Huang, "Map-matching for low-sampling-rate GPS trajectories," Proc. 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, New York, NY, USA, 2009.
[5] M. Vazirgiannis, Y. Theodoridis, and T. Sellis, "Spatio-temporal composition and indexing for large multimedia applications," Multimedia Systems - MMS, vol.6, no.4, pp.284–298, 1998.
[6] Y.F. Tao and D. Papadias, "MV3R-Tree: A spatio-temporal access method for timestamp and interval queries," Proc. Very Large Data Bases, pp.431–440, 2001.
[7] V.P. Chakka, A. Everspaugh, and J.M. Patel, "Indexing large trajectory data sets with SETI," Proc. Conference on Innovative Data Systems Research - CIDR, 2003.
[8] D. Lin and C.S. Jensen, "Efficient indexing of the historical, present,

and future positions of moving objects," Proc. MDM, Ayia Napa, Cyprus, 2005.
[9] V. Botea, D. Mallett, M.A. Nascimento, and J. Sander, "PIST: An efficient and practical indexing technique for historical spatio-temporal point data," GEOINFORMATICA, vol.12, no.2, pp.143–168, 2008.
[10] H. Garcia-Molina, J.D. Ullman, and J. Widom, Database System Implementation, Second ed., Prentice Hall, 2009.
[11] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento, "On the generation of spatiotemporal datasets," Proc. 6th Int'l Symposium on Large Spatial Databases (SSD), Hong Kong, China, July, 1999.
[12] S. Rasetic, J. Sander, J. Elding, and M.A. Nascimento, "A trajectory splitting model for efficient spatio-temporal indexing," Proc. 31st VLDB Conference, Trondheim, Norway, 2005.

Xiangxu Meng is Ph.D. candidate with National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. He has obtained the M.D. and B.S. in 2007 and 2004, respectively, all in computer science. His research interests include Location privacy and security; Location based service system; Spatial-temporal data query and mining.

Xiaodong Wang is Professor with National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. He has obtained the Ph.D., M.D. and B.S. in 2002, 1998 and 1996, respectively, all in computer science. His research interests include mobile computing, social network, wireless ad hoc network and wireless sensor networks.



Xinye Lin is Ph.D. candidate with National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China. He has obtained the M.D. and B.S. in 2011 and 2008, respectively, all in computer science. His research interests include data mining and recommendation system.