A Real-time Trajectory Indexing Method based on MongoDB

Yuxing Zhu
School of Software
Jiangxi Normal University
Nanchang, China

Jun Gong
School of Software
Jiangxi Normal University
Nanchang, China

Abstract—**Aiming at the inefficiency problems of existing trajectory database techniques especially in real-time access to latest trajectories, a real-time trajectory indexing method based on MongoDB and mixed with spatio-temporal R-tree, hash table and B-tree for searching leaf nodes is proposed in this paper. Time in spatio-temporal R-tree is used as another dimension of equal status to space, and a leaf node can only involve a moving object's consecutive trajectory points. In order to solve the problem of frequent updates and lack of memory, hash table is divided into two kinds: one caches leaf nodes of spatio-temporal R-tree, which are not inserted into spatio-temporal R-tree until they are full or out-dated in the hash table. This improves generation efficiency of real-time trajectory index; the other one caches in-memory nodes which are loaded from external memory, it avoids frequent operations related to external memory. We build B-tree based on object identification and time in leaf nodes, which benefits trajectory queries for moving objects. In comparison to SETI, the experimental results show that our method has good update efficiency and query performance, and it meets the demand of common trajectory queries in present applications.**

Keywords-component; MongoDB; spatio-temporal R-tree; real-time index; trajectory queries.

I. INTRODUCTION

In the recent years, with the widespread use of Global Positioning System (GPS) for civil use in Mobile Terminal (MT) and the development and popularization of the Location-Based Service (LBS) and Mobile Social Network (MSN), big trajectory data in daily life is increasingly accumulation and also services for different types of applications, and it's very important to manage and retrieve the large-scale trajectory data. In order to solve the problem of existing relational database techniques and efficiently manage large-scale and highly concurrent trajectory sets, MongoDB is a good choice for the problem. MongoDB is a product between relational database and nosql database which uses distributed technology. In comparison to relational database, nosql database's main features include: (1) Schema-free, (2) Automatic processing pieces to support the scalability of cloud computing levels. (3) Build-in GridFS, which supports the terabytes storage for big trajectory data. (4) Autosharding, which supports database cluster of horizontal scaling, dynamically adds and removes shards from the cluster.

National Natural Science Foundation of China(41261086); Jiangxi Postgraduate Innovation Foundation Project(YC2013-S108)

R-tree [1] has the features of dynamic updates and depth balancing, and it can easily involve in multi-dimensional objects. We propose a new spatio-temporal R-tree index, which processes trajectory points from moving objects. 3DR-tree [2], time is modeled as the third dimension in addition to the two location dimensions (x, y), but 3DR-tree's query performance is not good. SETI [3] divides the space into disjoint cells, and each cell maintains temporal index of B-tree for its objects' movements. TB-tree [4] strictly preserves trajectories as well as allows for R-tree typical range search for data, however, if some segments belonging to different moving objects are geographically adjacent, they have to preserve redundant cube in the different nodes. The other access methods have been introduced in references [5] [6]. Aiming at the efficiency of real-time access to the big trajectory data and trajectory queries in MongoDB, the performance of above-motioned indexes do not satisfy these requirements. Therefore, we propose a real-time trajectory index method mixed with spatio-temporal R-tree, hash and B-tree for searching leaf nodes. Time in spatio-temporal R-tree is used as another equal status dimension to space, and a leaf node preserves consecutive spatio-temporal points. This method not only ensures spatial proximity for moving objects, but also implements efficient retrieval to the whole or partial trajectories of moving objects.

The remainder of this paper is organized as follows: data structure of real-time trajectory index is presented in Section 2, and Section 3 presents basic operating algorithms of insertion and queries for real-time trajectory index. Section 4 analyzes some experimental results of spatio-temporal query operation in MongoDB between real-time index and SETI.

II. DATA STRUCTURE

In this paper, the properties of a moving object at a time stamp is called as object state. Suppose that space is expressed as two location dimensions *(x, y)*, then time is in addition to space of the form *(x, y, t)*. Moving objects' data is divided into two types: historical data (historical vision data) and current data (current vision data). The current data refers to an object state of recent updating, and it is frequently updated if a moving object constantly moves. The historical data refers to an object state updated. When a moving object moves, the current data will be out-dated and switched to be the historical data. This paper mainly focuses on real-time access to the historical data, although many methods like 3DR-tree and

MV3R-tree [7] propose the access method of maintaining moving objects proximity, their basic properties are all guaranteed I/O efficiency in query process. However, when we do trajectory queries, these methods can not guarantee high efficiency. Because segments in the same moving object may not be geographically adjacent, and they will not be preserved in the identical disk page.

With these above-motioned shortcomings, because of a large number of trajectories and trajectory-based queries, a new trajectory index scheme called real-time trajectory indexing method is proposed to increase update efficiency and query performance. Our method is based on MongoDB and mixed with spatio-temporal R-tree, hash table and B-tree for searching leaf nodes. In the spatio-temporal R-tree structure, time is used as another dimension identical to space, and a leaf node only involves consecutive trajectory points. In particular, we propose that leaf nodes in a target object are linearly queried through B-tree in MongoDB. As figure 1 and figure 2 show the concrete internal structure of index. When we are searching trajectories at a time stamp (or in a time slice), since the leaf node only belongs to a moving object and the trajectory consists of leaf nodes which are sorted by time dimension, we can build B-tree index on attributes of object identification and time, where we can find the leaf nodes we need and obtain the tuples from leaf nodes to the final results. Therefore, it increases the performance of trajectory-based queries.

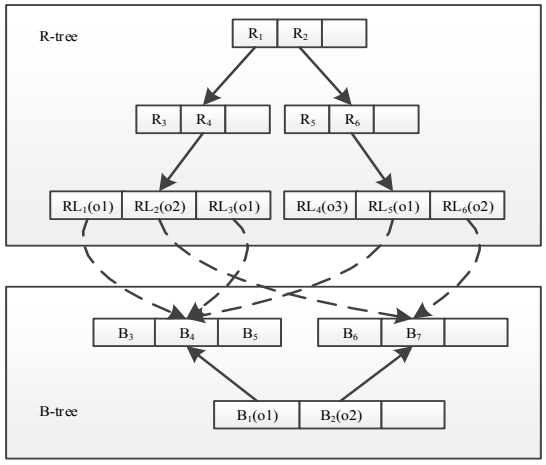

Figure 1. Internal structure of real-time trajectory index

As Table 1 shows, the fields in spatio-temporal index table consist of the unique identification for a leaf node, the information for node entry, the start time of node and the identification for a moving object. We should also set the unique identification to Globally Unique Identifier to keep R-tree index strong. Since most of the information is kept in node entry, we should be efficient to access this field. Therefore, the node entry which mainly includes the attributes of the spatio-temporal R-tree is preserved as the binary data. Notice that, the moving object identification will not exist in nonleaf nodes except for leaf nodes. Since we have built a B-tree index on the attributes of *Object_id* and *T_start* in leaf nodes, we can sort leaf nodes by object identification and time. Consequently, we can get the whole trajectory through connecting the front and back segments from the same moving object.

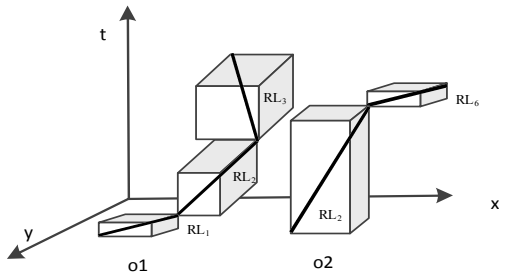

Figure 2. Minimum bounding box(MBB) of leaf nodes

TABLE I. FIELD DESCRIPTION TO SPATIO-TEMPORAL INDEX TABLE IN MONGODB.

Field Name	*Field Description*
oid	The unique identification of leaf nodes (Globally Unique Identifier).
binData	Binary package for node entries' information.
Object_id	If its value is valid, it refers to leaf node and its value is an object identification; else it refers to nonleaf node.
T_start	Start time of node.

Aiming to the trajectory-based model, this spatio-temporal R-tree structure is named *R1* to maintain data items of hierarchical tree structure for retrieving segments, but it still remains the problem of frequent updates and lack of memory for large amounts of nodes. Therefore, we also employ the hash structure to deal with these requirements. The hash table in this paper is divided into two kinds: One caches leaf nodes but out of spatio-temporal R-tree and the other one caches in-memory nodes which are loaded form external memory. In the former hash table, the leaf node is not inserted into the *R1* tree right now until the leaf node is full or out-dated. Therefore, *R1* supports bulk update operations. This bulk operations improves efficiency of dynamic updates for the *R1* tree; The latter hash table avoids frequent interactions between memory and external memory, and reduces the memory proportion of the *R1* tree, this hash utilizes partial persistency techniques. Mixing with *R1*, two hash tables and B-tree index of MongoDB constructs this real-time trajectory indexing structure.

R1's nonleaf node entry is of the form *(oid, mbb,* t_{start}*,* t_{end}*, children [1...n])*, *mbb* refers to spatial minimum bounding box of *R1* nodes, t_{start} and t_{end} are start time and end time of *R1* nodes, the element of *children* points to the nonleaf node's child. The entry from the leaf node is of the form *(oid, o, mbb,* t_{start}*,* t_{end}*, tuple[1...n])*, *tuple* refers to the element of a trajectory segment, *o* refers to the moving object's unique identifier. Two hash tables' summary of in-memory structure is below:

This hash table is expressed as *map(o, RL)*, which is built in the hash of the secondary index structure between the moving object (*o*) and the latest leaf node (*RL*), and it maps a moving object to the latest leaf node. When a new trajectory point is inserted into real-time trajectory index, we will place the point to the position in the hash depending on the moving object of the point.. If we find a data item in *map(o, RL)*, we will place the point into this data item, and it implements lazy updates for bulk update operations, else we will create a new data item in *map(o, RL)* mapping the moving object of the new point to an empty leaf node, and append this point as the first tuple into the new node. As figure 3 shows, the trajectory points should be first cached into the latest leaf node of *map(o, RL)*, and the node won't immediately be inserted into *R1* only if the leaf node is full or out-dated. Subsequently, after the latest leaf node is inserted into *R1,* it will be emptied to load new points. Since there are not frequently updates in *R1*, it will improve the efficiency of dynamically updates.

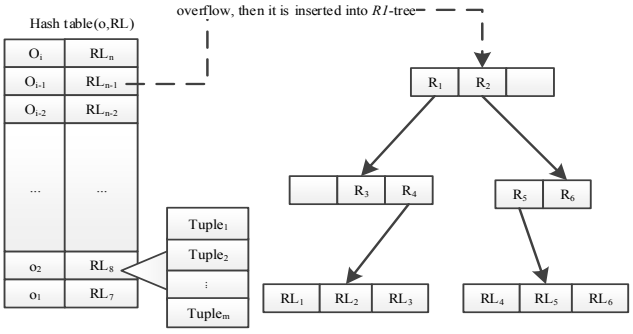

Figure 3. Inserting node after overflow in *map(o, RL)*

Another hash table is the form of *map(oid, R)*, which is built in the secondary hash index structure mapping the unique identifier (*oid*) to an in-memory node. We set that all of in-memory nodes should be mapped to hash table - *map(oid, R)*. The purpose of *map(oid, R)* is to cache in-memory nodes which have been recovered from real-time index table in MongoDB and to preserve active nodes in partial *R1*, it utilizes partial persistency techniques. As figure 4 shows, when obtaining the *oid* of a node, first we should know that if we can find the *oid* in *map(oid, R)* or not, if not we will recover the in-memory node corresponding to the *oid* from real-time index table in MongoDB and insert the *oid* and node into *map(oid, R)*. When processing trajectory-based queries or insertion in real-time index, we can load in-memory nodes entries for partly of *R1* to keep nodes active, it reduces the loss of in-memory resources in system as much as possible and avoids frequent operation related to external memory.

Both hash tables contribute to the efficiency of dynamically updates and the loss of in-memory resources, and are the essential structure of real-time trajectory index. This is important features in this paper.

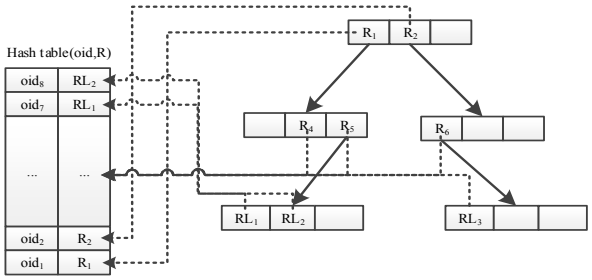

Figure 4. Nodes cache of *map(oid, R)*

III. OPERATION ALGORITHM

A. Inserting data

There are two operations - inserting points and inserting leaf nodes in this paper. Inserting points applies to *map(o, RL)*, which indicates that we should cache the new trajectory points into a node from the *map(o, RL)* depending on its moving object. But if the latest leaf node overflows or out-dated, we should do the operation - inserting leaf nodes. As known above, the insertion of *R1* is to insert the latest leaf node as the data item into *R1*, which is different from the insertion of other R-tree variants, and it improves the efficiency of insertion in *R1*.

Physical deletion does not exist in *R1* due to irreversible time. Therefore, we will not involve deletion. This section mainly introduces the interactive insertion between *R1* tree and the latest leaf node of *map(o, RL)*, and the algorithms of the search and inserting leaf nodes have been mentioned in references [8] [9]. As figure 5 shows, when inserting an entry *(T)*, we will call *insertToIndex* function to insert this new point. *N1* refers to the latest leaf node entry in *map(o, RL)*, *R1* refers to spatio-temporal R-tree structure. *insertLeafToIndex*'s function is to insert the whole leaf node into *R1* tree.

Algorithm insertion of trajectory index *R1*: *insertToIndex(T)*

Algorithm for entry:
  if *find the latest leaf node N1 of object T.o* in *map(o, RL)* ***then***
    if *N1 is not full* ***then***
      insert entry T into node N1; ***return****;*
    else
      insertLeafToIndex(N1): insert N1 into R1;
      N1(NULL): remove all of tuples from the leaf node N1;
      insert entry T into node N1;
    endif ;
  else
    create the new leaf node corresponding to T.o,and insert it into map(o, RL);

Figure 5. Insertion function *insertToIndex(T)*

B. Query operations

According to queries of past, recent queries for related research include: (1) spatio-temporal range queries, which search all the entries in a spatio-temporal interval; (2) time stamp queries, which search the current version objects in a space interval at a time stamp; (3) nearest neighbor queries; (4) trajectory-based queries. This paper mainly introduces two query algorithms, one is ordinary *R1* traversal algorithm, and the other one is trajectory-based queries algorithm. The former is based on *R1* tree, and the latter is based on B-tree of MongoDB to retrieve leaf nodes of *R1* tree.

As the ordinary operation of queries in MongoDB, we get the next result through *Next* function. The path of this query is preserved in a global variable of stack *S* where we can remember the visited nodes in *R1*. The data item in stack *S* is the form of *(R-n)*, *R* refers to the node of *R1*, and *n* refers to the *n*-th child in the node *R* when walk in *R1*'s node. For example, as figure 6 shows below, we use stack *S1* which is expressed by *(Root-1, R_1-3, R_6-3, RL_3-m)*, *Root* refers to the root node in *R1*, When calling *Next* to obtain next result, since all the tuples in RL_3 have been walked through, we pop the data item RL_3-*m* from stack *S1*. At the same time, all of children of R_6 and R_1 have been walked through, then the data items corresponding to R_6 and R_1 should be popped from stack *S1*. We walk the first child of RL_4 from the child R_7 of R_2, subsequently, the first tuple as the child of node RL_4 is obtained. Conclusively, the final result is showed as stack *S2 (Root-2, R_2-1, R_7-1, RL_4-1)* in figure 6 and figure 7. However, the above-mentioned context happening is that all the accessing nodes' boxes overlap the search range, if the box do not overlap the range, we should access the next sibling node of the current node to compare, and need not access the children of the current node.

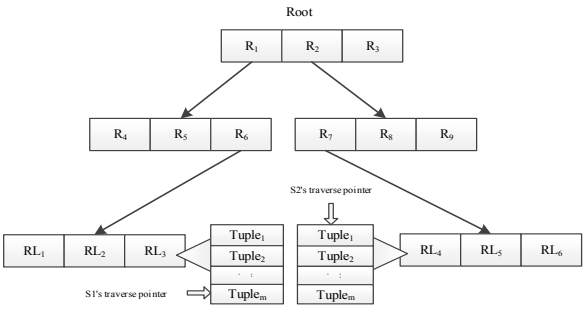

Figure 6. The stack in traverse query.

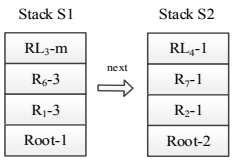

Figure 7. Stack *S1* and stack *S2*.

Aiming at searching the trajectory in a spatio-temporal range, we propose an another query algorithm for the efficient search. In figure 1, we find documents in the search range through B-tree which builds index on attributes *(Object_id, T_start)* from index table, and recover documents to in-memory nodes in *R1* tree which will be sorted by time and preserved in an array. Conclusively, we obtain tuples which come from eligible nodes from the array.

However, the results of these two query algorithms are not final results that we want, because some latest leaf nodes are still cached in *map(o, RL)*. In figure 3, after traversing the R-tree and B-tree structure, we also need to retrieve the cached leaf nodes in *map(o, RL)*. Although this costs extra time to search results in *map(o, RL),* however, this is good for the query of the spatio-temporal data of the latest updates.

IV. QUERY PERFORMANCE ANALYSIS

Since SETI is already thoroughly embedded in MongoDB's structure and this method has better performance compared to some other index structures, we only present the experimental results concerning our index and compare our index to SETI.

A. Experimental Settings

The experiments are carried out on a PC with 2.80GHz Intel(R) Core(TM)2 Duo CPU, Windows 7 platform, and 2.00 GB RAM. In real-time trajectory index, every nonleaf node contains between 16 and 40 entries, and every leaf node contains 80 tuples. Since SETI is embedded in MongoDB, we can't obtain its concrete node structure.

We use the real data which has been collected by Naval Academy Research Lab, France for research purposes. On the basis of the real data, we also extend the number 5 million to 70 million by time dimension.

B. Experimental Results

Our indexing scheme is mainly compared and evaluated in cost time of window queries. We compare the performance between real-time trajectory index and SETI which is embedded in MongoDB. In the following experiments, we will use the variables of space percentage and time to prove the efficiency in our method, and both conditions in windows queries are linear growth. Space percentage range is between in (5%, 10%, 15%), the time range is between in (5 days, 10 days, 15 days, 20 days, 25 days, 30 days).

We test ten times in every experiment, and remove the highest cost time and the lowest cost time. At last we obtain the mean of this values for the result.

Since the efficiency of query performance can be judged by the cost of query time which is took from the query process. Then, the shorter the cost of query time, the higher the performance of query. In the other side, the longer the cost of query time, the lower the query performance. In the following figures, in order to cut the number of words for less cost of space, trajectory index mean real-time trajectory indexing method and MongoDB index means SETI method.

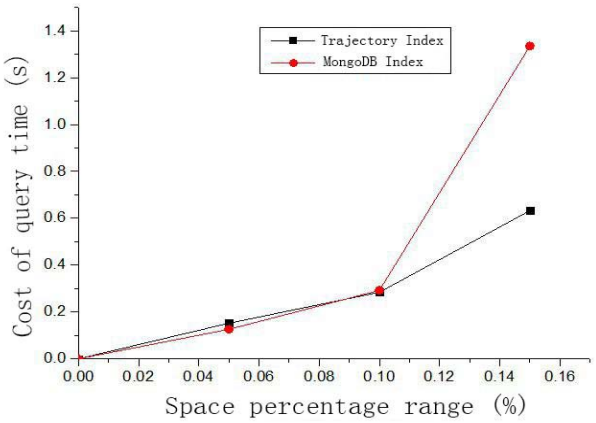

Figure 8. Comparison of query time for 5 million data in space range

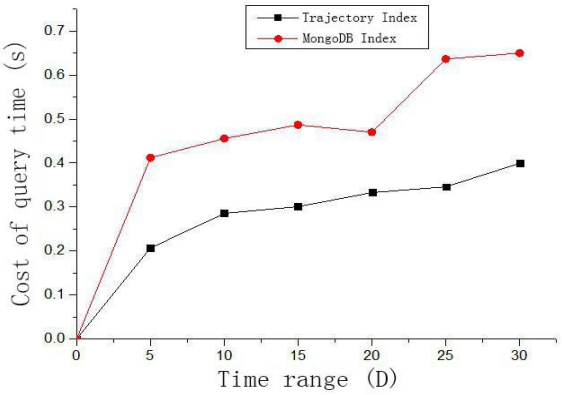

Figure 9. Comparison of query time for 5 million data in time range

As the curves trend in figure 8 and figure 9, when we use a data sample with 5 million trajectory points and the space percentage between in [10%, 15%], the query performance of the real-time trajectory index is higher than SETI. It shows the larger windows range, the higher performance of windows queries in real-time trajectory index.

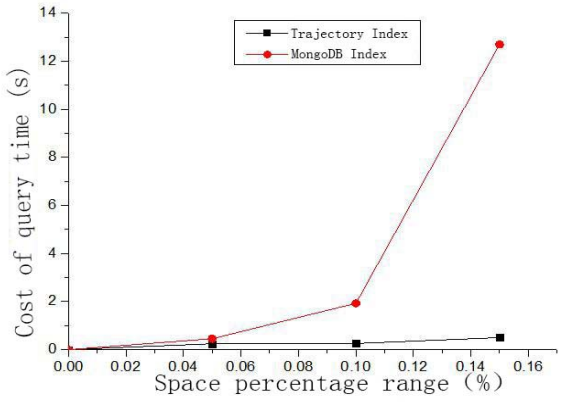

Figure 10. Comparison of query time for 70 million data in space range

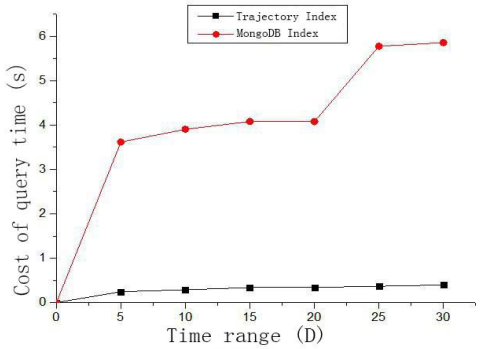

Figure 11. Comparison of query time for 70 million data in time range.

According to the comparison of the cost of query time between 5 million and 70 million spatio-temporal data in figure 8 and 10 or figure 9 and 11, we can learn that the larger number of spatio-temporal data, the higher performance of the real-time trajectory index in comparison to SETI.

V. CONCLUSION

Spatio-temporal index is an important part of spatio-temporal data management approach, which provides an efficient query method for large-scale spatio-temporal data based on the location, shape, time-related parameters. In the first, we briefly present the features of MongoDB, then introduce data structure and operation algorithm of real-time trajectory index: insertion and query operation. At last, we provide the experimental results for comparing real-time trajectory index to SETI, and it shows the real-time trajectory indexing method has efficient updates and queries.

Real-time trajectory indexing method based on MongoDB meets the demand of spatio-temporal queries in GIS system, improves the efficient performance of queries, and effectively solves the problem of inefficient trajectory-based queries in MongoDB, it has an important significance for spatio-temporal database. However, with the widespread use of multithreading, when multiple users simultaneously do queries or insertion to the same index structure, the results will be inaccurate and the node resource of trajectory index structure will be preempted. At last it causes confusion in *R1* tree. Therefore, we need the concurrency control to deal with the problem [10]. When the system where real-time trajectory index builds happens to a sudden collapse, and loses the current information of *R1* tree, it will be difficult to recover *R1* tree from external memory, and we need recovery mechanism [11] to solve the collapse problem. Therefore, our follow-up work is mainly to focus on concurrency control and recovery mechanisms.

REFERENCES

[1] Guttman, Antonin. R-trees: A dynamic index structure for spatial searching. Vol. 14. No. 2. ACM, 1984.

[2] Theoderidis, Y., Michalis Vazirgiannis, and Timos Sellis. "Spatio-temporal indexing for large multimedia applications." In Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on, pp. 441-448. IEEE, 1996.

[3] Chakka, V. Prasad, Adam C. Everspaugh, and Jignesh M. Patel. "Indexing large trajectory data sets with SETI." Ann Arbor 1001 (2003): 48109-2122.

[4] Pfoser, Dieter, Christian S. Jensen, and Yannis Theodoridis. "Novel approaches to the indexing of moving object trajectories." In Proceedings of VLDB, pp. 395-406. 2000.

[5] Mokbel, Mohamed F., Thanaa M. Ghanem, and Walid G. Aref. "Spatio-temporal access methods." IEEE Data Eng. Bull. 26, no. 2 (2003): 40-49.

[6] Nguyen-Dinh, Long-Van, Walid G. Aref, and Mohamed Mokbel. "Spatio-temporal access methods: Part 2 (2003-2010)." (2010).

[7] Tao, Yufei, and Dimitris Papadias. "The mv3r-tree: A spatio-temporal access method for timestamp and interval queries." (2001).

[8] Gong Jun, Ke Sheng-nan and Bao Shu-ming. "Brand-new node-choosing algorithm of R-tree spatial index." In Application Research of Computers, 2008, 25(10) , pp. 2946-2948+2955.

[9] Gong Jun, Zhu Qing, Zhang Yeting. "An Efficient 3D R-tree Extension Method Concerned with Levels of Detail." Acta Geodaetica et Cartographica Sinica, 2011, 40(2), pp. 249-255.

[10] Kornacker, Marcel, and Douglas Banks. "High-concurrency locking in R-trees." In VLDB, vol. 95, pp. 134-145. 1995.

[11] Kornacker, Marcel, C. Mohan, and Joseph M. Hellerstein. "Concurrency and recovery in generalized search trees." ACM SIGMOD Record. Vol. 26. No. 2. ACM, 1997.

[12] Wang Ping-Gen, Zho Jiao-Gen. "A Hybrid Indexing Mechanism for Spatial-Temporal Databases." Computer Science, 2007, 34(9), pp. 103-106.