RSR-tree: A Dynamic Multi-dimensional Index Structure

Aiguo Li, Chi Zhang
School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an, China
Email: liag@xust.edu.cn, chi_zh@163.com

Jiulong Zhang
School of Computer Science and Engineering, Xi'an University of Technology, Xi'an, China
Email: zhangjiulong@xaut.edu.cn

Zhenhai Zhang
School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an, China
Email: htty326@163.com

***Abstract*—A new dynamic multi-dimensional index structure called RSR-tree is proposed, in which splitting operations of leaf nodes and non-leaf nodes are performed separatively. RSR-tree retains the sequence ordering of index records in a leaf node of RS-tree and thus reduces the addressing time cost of disk access while reading data files. In addition, it integrates the characteristic of reducing the overlap between brother nodes of R-tree to reduce the query time. The accuracy test and parameter sensitivity test are done on different data sets and the experimental results show that RSR-tree is a dynamic multi-dimensional structure which can efficiently improve query performance and does not bring extra cost of creating index at the same time.**

***Index Terms*—index structure, dynamic index, data structure, balanced multiway search tree**

I. INTRODUCTION

Historical database in real-time database system is used to store and manage a mass of historical data. In case there is no index the great amount of multi-dimensional historical data with, data querying is quite slow. To tackle this problem, a number of multi-dimensional index structures have been proposed [1, 2]. After Guttman proposed the R-tree index structure [3] in 1984, many variations to R-tree and various optimization criterions [4,5] have been proposed over two decades such as R^+-tree [6], R*-tree [7], Hilbert R [8], RS-tree [9] and so on. Generally, R-tree and its variants are hierarchical data structure, and they are height-balanced tree using object definition technology [10].

The time cost T of query in the database system with multi-dimensional index mainly includes two parts: the time of searching the index file, T_i and the time of reading data from the data file, T_d. R-tree, R^+-tree and R*-tree mainly address how to reduce T_i and neglect the importance of T_d. But A.G. Li etc point out that T_d is the most important factors in the whole query process, and the query time can be improved by reducing T_d. Therefore, ordering of nodes is proposed to reduce T_d in the RS-tree structure. The query experiments with RS-tree show that T_d accounts for 90 percent of the total query time in VegeBam2.0 a database system developed by us, and T_d is directly proportional to the number of scanning index records C_i, i,e., the number of disk accesses will be increased when C_i increases, causing T_d to increase. The RS-tree structure provides that storage locations of index records are sequential in leaf nodes, which reduces T_d and improves the performance of searching in index tree. So reducing T_d means reducing C_i. Hence the new index structure RSR-tree based on the improvement of R-tree and RS-tree is proposed. RSR-tree keeps ordered the storage locations of index records of leaf node in database file for reducing the addressing cost of disk accesses in reading data files, and in non-leaf nodes, it reduces the overlap between brother nodes to reduce the number of search paths in searching index tree, so it can reduce C_i and T_d to get the ultimate pursue of reducing query time T.

This work was Supported by Key Scientific and Technological project of Shaanxi Province, China (No. 2008K01-58 and No. 2009K01-56)

II. RSR-TREE INDEX STRUCTURE

The descriptions of the RSR-tree index structure are defined as following.

Definition 1 [4]: For object A and object B, if the position of object A is before object B and object B follows close behind object A, then this relation is represented as A<B.

Definition 2: For object A and object B, object A has k entries as aC that: $aC_1<aC_2<\ldots<aC_k$, object B has h entries as bC that: $bC_1<bC_2<\ldots<bC_h$, if $aC<bC$, it means the first entry of object B bC_1 follows close behind the last entry of object A aC_k ($aC_k<bC_1$), then this relation is represented as A<<B.

The same as RS-tree, RSR-tree is a height-balanced tree. It consists of leaf nodes and non-leaf nodes. The data object region of leaf node stores the information of index data, and the data object region of non-leaf node stores child nodes.

doi:10.4304/jcp.6.12.2552-2558

A leaf node is shown as following

$$(I, T)$$

where I is an n-dimensional minimum bounding rectangle of the data object containing all index records in a leaf node. Its form is

$$I = (I_1, I_2, \ldots, I_n)$$

where, n is the number of dimension in the multi-dimensional space, and I_i is a closed interval $[a, b]$ describing I along dimension i, where a is the minimum and b is the maximum.

Each data record in the database file has a unique identifier. T is the tuple that saves the identifiers of data records in the database file which are associated with all the index records contained in a leaf node. The form of T is

$$T = (T_1, T_2, \ldots, T_k)$$

where k denotes the number of index records contained in a leaf node and T_i is the unique identifier of index record i, and there are $T_1<T_2<...<T_k$ in the database file.

A non-leaf node is described as the form

$$(I, NC)$$

where I is an n-dimensional minimum bounding rectangle of the spatial object covering all child nodes in a non-leaf node, which is just like a leaf node. NC is the tuple containing collection of child nodes in a non-leaf node, and it has the form

$$NC = (NC_1, NC_2, \ldots, NC_k)$$

where k refers to the number of child nodes contained in a node, and NC_i is a child node of a non-leaf node ($1\le i\le k$).

If there is $S\subseteq NC$, $S\neq\phi$, then, S is represented as sub-item set of a node. S is formed as follow

$$(I, SC)$$

where, I is an n-dimensional minimum bounding rectangle of the spatial object that contains all sub-items in S and its specific representation is the same as that in a leaf node. SC is the collection of all sub-items in S, and its form is

$$SC = (SC_1, SC_2, \ldots, SC_t)$$

where t denotes the number of sub-items contained in S, and $t=|S|$, and SC_i is a sub-item in S ($1\le i\le t$).

The structure of RSR-tree is displayed in Fig. 1. In Fig.1, N0 is the root of the tree, and N1~N3 are non-leaf nodes, and L1~L10 are leaf nodes. There are two characteristics in the RSR-tree as follows.

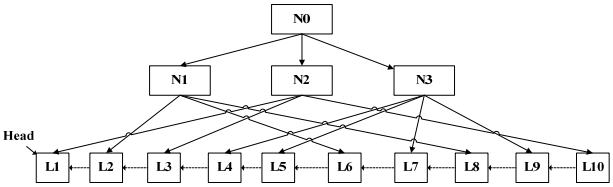


Figure 1. The structure of RSR-tree

a) Sequence ordering of index records in a leaf node

It can be seen that index records in a leaf node are sequential from the description of RSR-tree structure. According to the definition of the leaf node structure, there is $L.T_1<L.T_2<\ldots<L.T_k$ in a leaf node L.

b) Ordering of leaf nodes

The leaf nodes of RSR-tree are in the form of ordered list, with the ordering of leaf nodes: there are $L_i<<L_j$ between leaf node L_i and L_j ($i<j$). In Fig. 1, though there is the relation L5<<L6 between the leaf nodes L5 and L6, there does not exist the relation N3<<N1 between their parent nodes N3 and N1.

Assume m is the minimum number of entries in a node, and M is the maximum number of entries. To sum up, an RSR-tree has the following properties:

- In every node, there are the relation between m and M as $2\le m\le M/2$.
- A root node has two child nodes at least unless it is a leaf node.
- A non-leaf node has k child nodes and $m\le k\le M$, unless it is also a root node.
- A leaf node has k index records, and there is $m\le k\le M$ unless it is also a root node.
- For each non-leaf node (I, NC), I is the minimum bounding rectangle that must covers all the minimum bounding rectangles of all child nodes in the non-leaf node.
- For each leaf node (I, T), I is the minimum bounding rectangle that must covers all the minimum bounding rectangles of all index records in the leaf node.
- The minimum bounding rectangle I of each index record in a leaf node must contains the n-dimensional data object of the index record.
- The leaf node has sequence ordering of index records in a leaf node, and the index records of leaf node are arranged according as the storage locations of records in data file increase.
- There is the ordering of leaf nodes, and the form of the leaf layer is the sequential list.
- All leaf nodes are on the same level.

III. ALGORITHM DESCRIPTION OF RSR-TREE

In the RSR-tree algorithm, the insert algorithm adds data records by the principle of records ordering. Similar to *Distribute* algorithm of RS-tree, the split algorithm of the leaf node divides a leaf node by the sequence ordering of index records in the leaf node. The split algorithm of non-leaf node is similar to *PickSeeds* and *PickNext* process of R-tree in that neither of them considers the sequence ordering of nodes. Thus it ensures the sequence ordering of index records in leaf node and a reduction of overlap between non-leaf nodes.

A. Insert Algorithm.

The insert algorithm of RSR-tree adds a new index record into the suitable position of RSR-tree. The operations of the *Insertion* are: finding the appropriate leaf node L by scanning the RSR-tree into which the new index record E is added. The description of *Insertion* is as follows:

Input: A RSR-tree root node R and an index record E

Output: The new RSR-tree

Method: Find the position where E should be inserted, and add it to the appropriate leaf node

I1 [Find the appropriate leaf node] Call *Choose*(R, E) to select the appropriate leaf node L where the new

index record E is placed

I2 [Add the new index record to the leaf node] Add the new index record E into the selected appropriate leaf node L, then sort ascendingly the tuple T of the leaf node L by

I3 [Initialization] Set N to be the parent node of L

I4 [Update the index tree] Recalculate the minimum bounding rectangle I of node N, which covers all the minimum bounding rectangles of all child nodes contained in node N, then assign the parent node of N to N

I5 [Judge the end condition] if N is null, turn to I6. Otherwise turn to I4

I6 [Split the tree] Call *Split*(N) and if the number of child nodes in N is greater than M, return a new adjusted RSR-tree, otherwise return the RSR-tree that added a new index record

The main idea of algorithm *Choose* is to select the appropriate inserting leaf node for index record E by a top-down search starting from the root. The form of leaf nodes is a list, ordered by the storage location of records from small to large.

Input: A RSR-tree root node R and an index record E

Output: The appropriate leaf node L

Method: Find the appropriate leaf node in which to place E

C1 [Initialization] Set N=R

C2 [Judge] If N is a leaf node, turn to C4. Otherwise continue

C3 [Down search] Set N to the left child node of N, then turn to C2

C4 [Find the head of leaf nodes] If N is the head of leaf nodes, turn to C5. Otherwise, set N to the left brother node of N, then repeat C4

C5 [Get the right brother node of the leaf node] Get M that is the right brother node of N

C6 [Search the appropriate leaf node] If $N.T_1<E.id<N.T_k$, return N. If $N.T_k<E.id<M.T_1$, calculate $DIS_1=E.id-N.T_k$ and $DIS_2= M.T_1-E.id$, if $DIS_1> DIS_2$, return M, otherwise return N. If $E.id\geq M.T_1$, assign N to M, then turn to C5

B. Split Algorithm.

The split algorithm of RSR-tree splits a leaf node containing M+1 entries into two new leaf nodes in order to ensure that the number of entries in each leaf node is between m and M, then each node is adjusted from the leaf node to top in order to satisfy the constraints of RSR-tree.

The *Split* provides a bottom-up operation of adjusting each node from the node N in order to ensure that the number of entries in each node is between m and M in the RSR-tree. As the node types are different, the algorithm used to adjust is different.

Input: A node N in a RSR-tree

Output: A new adjusted RSR-tree

Method: Adjust each node from node N to top, to ensure that the number of entries in each node satisfies the properties of RSR-tree

S1 [Initialization] Assign node N to node L

S2 [Judge the node type] If L is a leaf node, turn to S3, otherwise turn to S4

S3 [Split a leaf node] Create a new node LR, and invoke *Distribute*(L) for redistributing from the child nodes in L to node L and LR

S4 [Search the two farthest sub-item sets in a non-leaf node] Invoke *PickSeeds*(L) to choose the two farthest entries in L as LC_1 and LC_2, and get three sub-item sets L_1、L_2 and LC, where LC is the remaining sub-item set of L without LC_1 and LC_2

S5 [Judge the end condition of distribution] If $|LC|$=0, or $|L_1|\geq M-m+1$, or $|L_2|\geq M-m+1$, turn to S7

S6 [Distribute an entry in the remaining sub-item set] Invoke *PickNext*(L_1, L_2, LC) to assign an entry from LC to L_1 or L_2, then go to S5

S7 [Split the non-leaf node] If $|L_1|<M-m+1$, all the remaining sub-items in LC are distributed to L_1. If $|L_2|<M-m+1$, all the remaining sub-items in LC are distributed to L_2. Create a new node LR, and assign L_1 to L and assign L_2 to LR

S8 [Grow the tree up] If the parent node of L is not null, assign the parent node of L to L. If L is the root node, create a new node R as the root node, and set the child nodes of R as L and LR, and assign R to L at last

S9 [Judge the end condition] If the number of entries in L is less or equal to M, the algorithm ends and returns. Otherwise turn to S2

The *Distribute* process provides operations as follows: According to the ordering of the tuple T in leaf node L and the principle of minimum overlap, the M+1 child nodes in leaf node L are divided into two groups to ensure that the number of child nodes of leaf node is between m and M.

Input: A leaf node L

Output: Two new leaf nodes: L and LR

Method: Distribute the entries from L to L and LR according to certain rules

D1 [Initialization] Set i=m, $Index$=i, calculate the minimum bounding rectangle I_1 that covers the first i entries of leaf node L and I_2 analogously for the last M+1-i entries of L, then set $I_{min}=I_1\cap I_2$

D2 [Search the minimum overlap]Set i=i+1, then calculate the minimum bounding rectangle I_1 that covers the first i entries of leaf node L and I_2 analogously for the last M+1-i entries of L. If $I_{min}>(I_1\cap I_2)$, set $I_{min}=I_1\cap I_2$ and $Index$=i

D3 [Judge the end condition] If $i<M$+1-m, turn to D2

D4 [Distribute sub-items] Assign the first $Index$ entries of L to L and assign the remaining entries of L to LR by the ordering of tuple T

The main operations of algorithm *PickSeeds* are as follows: Select two farthest entries NC_1 and NC_2 from the M+1 entries of non-leaf node N, then add NC_1 and NC_2 respectively into the sub-item sets N_1 and N_2, and return the sub-item set NC that contains the remaining entries of N without NC_1 and NC_2. In the algorithm, $P(I)$ is used to calculate the perimeter of the minimum bounding rectangle I.

Input: A non-leaf node N

Output: The sub-item sets N_1、N_2 and NC

Method: Choose the two farthest entries and add them respectively into the appropriate sub-item set

PS1 [Initialization] Let i=0, j=i+1, NC_1=NC_i, NC_2=NC_j, then calculate respectively the minimum bounding rectangle I_1 for NC_1 and I_2 for NC_2, and calculate the distance between NC_1 and NC_2: D_{max}=$P(I_1+I_2)$-$P(I_1)$-$P(I_2)$

PS2 [Judge the end condition] If i=M+1, turn to PS4

PS3 [Calculate the distance between two entries] If j<M+1, calculate respectively the minimum bounding rectangle I_1 covering the i-th child node of N and I_2 analogously for the j-th child node of N, and the distance between the i-th child node and the j-th child node in N is that: DIS= $P(I_1+I_2)$-$P(I_1)$-$P(I_2)$. If D_{max}<DIS, set D_{max}= DIS, NC_1=NC_i, NC_2=NC_j, j=j+1, then turn to PS3. Otherwise, set i=i+1, j=j+1, then turn to PS2

PS4 [Save the two farthest entries] NC_1 and NC_2 are added respectively into sub-item sets N_1 and N_2, and remove NC_1 and NC_2 from the sub-item set NC of N

Algorithm *PickNext* chooses one remaining entry in the sub-item set NC for classification in sub-item set N_1 or N_2.

Input: The sub-item sets NC, N_1 and N_2

Output: New sub-item sets NC, N_1 and N_2

Method: Select a remaining entry for classification in the nearest group

PN1 [Initialization] Let i=0, $Index$=i, $maxPre$=D_1=D_2=0, then calculate respectively the minimum bounding rectangle N_1I of N_1 and N_2I analogously of N_2

PN2 [Calculate the distance between the node and each sub-item set] Calculate respectively the minimum bounding rectangle I_1 covering the sub-item set N_1 and the entry NC_i and I_2 analogously for N_1 and NC_i , and the distance NC_i to N_1 and the distance NC_i to N_2 are respectively d_1=$P(I_1)$-$P(N_1)$-$P(NC_i)$, d_2=$P(I_2)$-$P(N_2)$-$P(NC_i)$. Then calculate the difference between d_1 and d_2: pre=$|d_1-d_2|$. If pre>$maxPre$, Let $maxPre$=pre , $Index$=i , D_1=d_1, D_2=d_2

PN3 [Judge the end condition] Set i=i+1. If i<$|NC|$, turn to PN2

PN4 [Distribute a entry] If D_1<D_2, assign NC_{Index} the $Index$-th entry of NC to N_1, otherwise assign NC_{Index} to N_2. Then remove NC_{Index} from NC

In the process of querying index tree, each node is searched with a top-down search starting from the root node, comparing the minimum boundary rectangle of each node with the one of query record until leaf node, then the content of index records in leaf node are read in order from the data files. If the query record is not searched after scanning leaf node, the scanning operation is repeated by recalling the parent node level, until the query record is found or unfound. In the query process, suppose the number of leaf nodes need to be read is m, and the number of index records in each leaf node is n_i, The time spent in reading the content of a record in data file is T_r, and the time of locating the position of record in data file is T_s, then the time spent in reading the index records is $T_d = \sum_i^m n_i \times (T_s + T_r)$. Because the location of index records in a leaf node in data file is continuous, T_s is negligible in the database with mass data, so there is $T_d = \sum_i^m n_i \times T_r$. In Fig. 2, non-leaf node A is divided into two nodes B and C. Using the split algorithm of R-tree, the algorithm of non-leaf node in RSR-tree divides the group of entries into the two farthest groups by the principle of the farthest distance, as shown in Fig. 2.a). In Fig. 2.b), the group entries are distributed by principle of ordering of nodes during the split process using the split algorithm of RS-tree. The split condition in R-tree is identical with the condition in RSR-tree in Fig. 2.c). The shadow is the overlap between nodes B and C. It can be seen that the overlap between B and C divided using the split algorithm of RSR-tree is smaller than that of RS-tree, and the number of query path in query process is smaller so that m is smaller. Therefore T_d is reduced and the query speed in RSR-tree is improved compared to RS-tree.

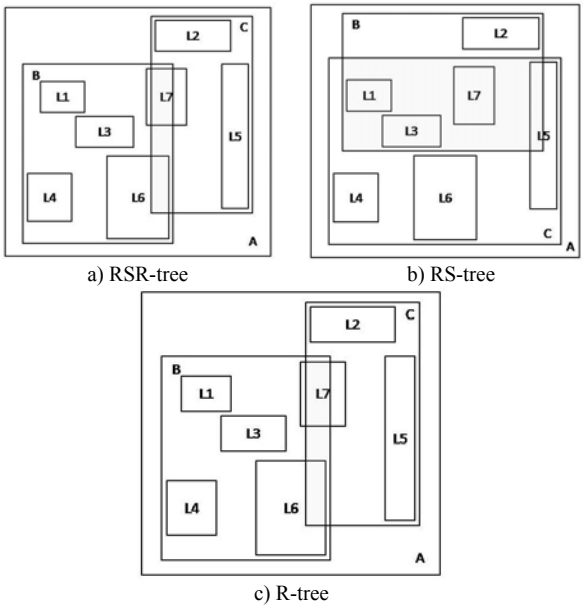


Figure 2. The condition of splitting node in three index trees

IV. EXPERIMENTS

The hardware in these experiments is a personal computer, whose CPU is Intel(R) Core(TM) 2 Duo CPU T8300 with core speed at 2.4GHz and 2GB memory. The operating system of the computer is Windows Vista Ultimate. The programs are written in standard C++ language and the compiler is Microsoft Visual Studio 2008 SP1. The experiments include two parts, namely the accuracy test and parameter sensitivity test.

Before the experiments, three definitions are given as the evaluation criterion.

Definition 3: In the writing data process *W*, the time of writing *N* data records is T_{wi}, and *N* data records are repeated *M* times, then the average time of writing data is

$$T_{wav} = \frac{1}{M}\sum_{i=1}^{M}\frac{T_{wi}}{N} \quad (1)$$

Definition 4: In a querying data process *Q*, the number of data records queried is N_i, and the number of scanning records is S_i, and the query process is repeated *M* times, then the average number of scanning record is

$$Q_{sav} = \frac{1}{M}\sum_{i=1}^{M}\frac{S_i}{N_i} \quad (2)$$

Definition 5: In a querying data process *Q*, the total time of querying N_i data records is T_{qi}, and the query process is repeated *M* times, then the average query time is

$$T_{qav} = \frac{1}{M}\sum_{i=1}^{M}\frac{T_{qi}}{N_i} \quad (3)$$

A. Test Data.

Two different data sets are used in the experiments, Intel data set and Bio_Train data set. Intel data set is collected from 54 sensors of Intel Berkeley Research lab, and Bio_train data set is obtained from KDDCUP2004.

History database system VegeBam2.0 is used throughout the experimental. *N* attributes in the data set are selected as the experimental data, and converted into FloatStream a data type of VegeBam2.0, and imported into the database. ID, a new attribute field, is added for the unique identifier of each record, and its data type is Int. Thus, every record in the database contains two fields—the first field is ID and the second field is experimental data. The 6 attributes of Intel data set are selected as the experimental data and import the data in accordance with the above-mentioned format into the database as the Intel recordset, the first column of which is linear, and the second column of which is always 1 and the remaining columns data value are fluctuating with little change. The first 74 attributes of Bio data set are selected as Bio recordset, which are imported into the database according to the above-mentioned form. The value of each column data in Bio recordset changes in fluctuating, 7 columns of which change in the larger beyond 50,000 and 6 columns of which change in the little within 10. The information of experimental recordsets is shown in TABLE I.

TABLE I. THE INFORMATION OF EXPERIMENTAL RECORDSETS

Attribute of Recordset	Name of Recordset	
	Intel Recordset	*Bio Recordset*
Dimension of FloatStream	6	74
Number of records	30,000	30,000

During writing data experiment, firstly, writing data program is run to create database with index for 30,000 data records, and note down T_{wi} the writing time, then the above process is repeated *M* times, finally T_{wav} the average time of writing data is calculated as the final experimental result. During query experiment, at first, *N* records are selected in random as the query records, then query program is run with the second field value of each query record as parameter whose data type is FloatStream and is repeated *M* times in the database with index, noting down S_i the scanning records number and T_{qi} the total query time of querying *N* records in each test, finally, the average number of scanning records Q_{sav} and the average query time T_{qav} are calculated as the final experimental results. During query experiment, it returns as long as the second field value of a record is the same as one of the query record.

B. Accuracy Test.

The accuracy test does the writing data experiment and the query experiment on RSR-tree index, R-tree index and RS-tree index in the different recordsets, and compares the T_{wav}, Q_{sav} and T_{qav}. The minimum number of entries in a node *m* is 10 and the maximum number of entries in one *M* is 30 in the created RSR-tree, R-tree and RS-tree.

a) Writing data experiment

The comparison tests of writing data, T_{wav} of R-tree index, RS-tree index and the RSR-tree index in Intel recordset or Bio recordset are obtained. The steps of writing data experiment are executed as follows:

Step 1: Choose the different index, and create the database with RSR-tree or the one with RS-tree or the one with R-tree index

Step 2: Insert the first 30,000 records of Intel recordset or Bio recordset into the database with RSR-tree or the one with RS-tree or the one with R-tree, and note down the time of each writing process T_{wi}. Repeat the process 3 times, finally calculate T_{wav}.

b) Query experiment

The test of query compares Q_{sav} and T_{qav} for R-tree index, RS-tree index and RSR-tree index in Intel recordset or Bio recordset. In the three types of index tree, the optimization criterion is that the perimeter of minimum bounding rectangle should be minimized. The steps of query experiment are as follows:

Step 1: Run the query test setting program and selects *N* records in random as the query records from Intel recordset or Bio recordset

Step 2: Invoke the *Search* method of RSR-tree (or the *Search* method of RS-tree or the *Search* method of R-tree) with the parameters that is the value of second field in the *N* query records, and note down the scanning records number S_i and the total query time T_{qi} in each query process

Step 3: Repeat the step 2 three times, then calculate the Q_{sav} and T_{qav}

Here are the experimental results of writing data experiment and query experiment in Intel recordset and Bio recordset.

TABLE II. THE AVERAGE TIME OF WRITING DATA OF THREE INDEXES IN DIFFERENT RECORDSETS

Number of writing records	T_{wav} (ms)					
	Intel recordset			Bio recordset		
	R-tree	*RS-tree*	*RSR-tree*	*R-tree*	*RS-tree*	*RSR-tree*
30,000	1.13	0.60	0.93	1.27	1.00	1.10

TABLE III. THE AVERAGE NUMBER OF SCANNING RECORD OF THREE INDEXES IN DIFFERENT RECORDSETS

Number of query records	Q_{sav}					
	Intel recordset			Bio recordset		
	R-tree	*RS-tree*	*RSR-tree*	*R-tree*	*RS-tree*	*RSR-tree*
100	19	303	18	6,235	4,444	3,951

TABLE IV. THE AVERAGE QUERY TIME OF THREE INDEXES IN DIFFERENT RECORDSETS

Number of query records	T_{qav} (ms)					
	Intel recordset			Bio recordset		
	R-tree	*RS-tree*	*RSR-tree*	*R-tree*	*RS-tree*	*RSR-tree*
100	0.59	7.07	0.57	144.23	103.12	92.91

It can be seen in TABLE II that T_{wav} of R-tree is the longest, and it is 89% higher than T_{wav} of RS-tree, and T_{wav} of R-tree is 56% higher than T_{wav} of RS-tree, during the writing data process with the three indexes in the experiment with Intel recordset. T_{wav} of R-tree is the longest during the writing data process with the three indexes in the experiment with Bio recordset, and it increases by 27% compared with T_{wav} of RS-tree index, and T_{wav} of RSR-tree increases by 10% compared with T_{wav} of RS-tree.

From TABLE III and IV it can be seen that Q_{sav} and T_{qav} are directly proportional during the query process. In Intel recordset, Q_{sav} of RSR-tree reduces by 94% compared to Q_{sav} of RS-tree, and it reduces by 5% compared to Q_{sav} of R-tree. T_{qav} of RSR-tree raises about 12 times compared to T_{qav} of RS-tree, and it is 4% higher than T_{qav} of R-tree. In the query process in Bio recordset, Q_{sav} of RSR-tree is 11% less than Q_{sav} of RS-tree, and it is 37% less than Q_{sav} of R-tree. T_{qav} of RSR-tree rises by 11% compared to T_{qav} of RS-tree and raises 1.5 times compared to T_{qav} of R-tree. In other words, the query speed of RSR-tree improves compared to that of R-tree and that of RS-tree.

From the above two groups of experiments, it can be seen that the query speed with R-tree is the highest in Intel recordset and Bio recordset. The query speed with R-tree is close to the one with RSR-tree in Intel recordset, but in Bio recordset the query speed with R-tree is the lowest. The reason is that the characteristics of the two recordsets are different, i.e., the Intel recordset has low dimension and the Bio recordset has high dimension. The query performance with R-tree in high dimensional data is lower, while the stability of query performance of RSR-tree is better than R-tree's, and RSR-tree has a better speed in querying data than RS-tree. In the writing process, the speed of writing data with RSR-tree is slightly slower than the one with RS-tree, and it is faster than the one with R-tree. The combination property contained writing data and query data display that RSR-tree has the synthetic advantage compared to R-tree and RS-tree. It is an effective multi-dimension index structure.

C. Parameter Sensitivity Test.

First of all, the test program alters the parameters value of RSR-tree, and it creates corresponding databases using the Intel data set and Bio data set respectively. Then the program imports the first 30,000 records of Intel data set and Bio data set, and does the writing data experiment and query experiment respectively. The main parameters in RSR-tree index are *m* that is the minimum number of entries in a node and *M* that is the maximum number of entries in a node.

a) Writing data experiment

The aim of writing data experiment is to compare T_{wav} of RSR-tree with the different value of *m* and *M*. The steps of the experiment are as follow.

Step 1: Set the value of *m* and *M*, and create the database with RSR-tree index

Step 2: Write the first 30,000 records of Intel data set or Bio data set in the database with RSR-tree index, and note down the time of executing writing data process T_{wi}. Repeat the process 3 times and calculate T_{wav}

b) Query experiment

During query experiment, the test program is run to do the query test in the Intel recordsets that have been created, and note down the Q_{sav} and T_{qav} of RSR-tree index with the different value of *m* and *M*. The experiment program performs as follow.

Step 1: Run the query test setting program and selects *N* records randomly from Intel recordset or Bio recordset as the query records from Intel recordset

Step 2: Invoke the *Search* method of RSR-tree with the parameters that is the value of second field in the N query records, and note down the scanning records number S_i and the total query time T_{qi} in each query process

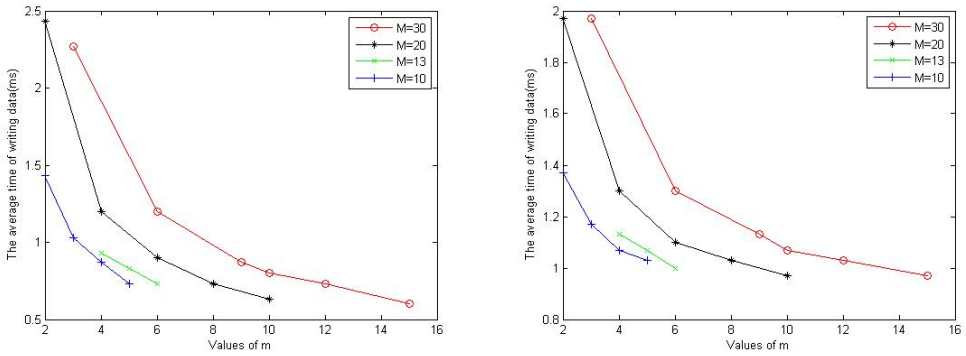
Step 3: Repeat the step 2 three times, then calculate the Q_{sav} and T_{qav}

The test program is run with respectively *M*=30, 20, 13, 10 and *m* = 10%, 20%, 30%, 40%, 50% of *M*. The several values of parameters are shown in TABLE V, and the symbol '-' expresses no value.

TABLE V. VALUES OF M AND M IN RSR-TREE

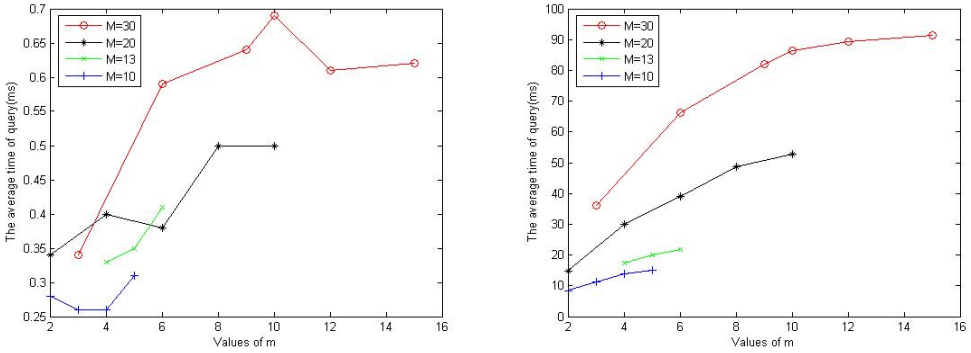
Value of *M*	Value of *m*					
	M×0.1	*M*×0.2	*M*×0.3	Default	*M*×0.4	*M*×0.5
30	3	6	9	10	12	15
20	2	4	6	-	8	10
13	-	-	4	-	5	6
10	-	2	3	-	4	5

Fig. 3 and 4 display the results of writing data and query experiment of RSR-tree with different values of *m* and *M* in Intel recordset and Bio recordset. From Fig. 3 it can be seen that, when *M* is certain, the bigger *m* is, the faster the speed of writing data with RSR-tree is. It displays in Fig. 4 that, when there is a certain value of *M* in query data experiments, the query time of RSR-tree is



a) Intel recordset b) Bio recordset

Figure 3. The average time of writing data of RSR-tree with different m and M



a) Intel recordset b) Bio recordset

Figure 4. The average query time of RSR-tree with different m and M

on the raise with the increasing of *m*. Thus, the value of *m* and *M* can affect the query performance of RSR-tree.

V. CONCLUSIONS

In the index structure based on RS-tree, reducing the number of scanning record is an efficient way to promote the querying speed in index, without increasing the cost of writing data when the speed of querying in index is enhanced. On above view, RSR-tree is promoted through combining the characteristic of R-tree and RS-tree. The key difference between RSR-tree and R-tree is that there is ordering of index records in leaf nodes and ordering of leaf nodes in RSR-tree, while ordering of non-leaf nodes is unnecessary; however RS-tree requires leaf and non-leaf nodes be both ordered. Through precision and parameter sensitivity experiments, the result shows that during the multi-dimensional data query process, RSR-tree is a multi-dimensional structure which can efficiently improve query performance and it does not bring extra cost of creating index at the same time.

REFERENCES

[1] V. Gaede, O. Gunther. "Multidimensional access methods," ACM Computing Surveys, vol. 30, pp. 170-231, 1998.

[2] C. Böhm, S. Berchtold, D.A. Keim. "Searching in highdimensional spaces: Index structures for improving the performance of multimedia databases," ACM Computing Surveys, vol. 33, pp. 322-373, 2001.

[3] A. Guttman. "R-TREES: A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING," Proc ACM SIGMOD Int Conf on Management of Data, pp. 47-57, 1984.

[4] T. Sellis, N. Roussopoulos, C. Faloutsos. "THE R+-TREE: A DYNAMIC INDEX FOR MULTI-DIMENSIONAL OBJECTS," Proc of the 13th VLDB, pp. 507-518, 1987.

[5] N. Beckmann, H.P. Kriegel. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangle," Proc ACM SIGMOD Int Conf on Management of Data, pp.322-331, 1990.

[6] I. Kamel, C. Faloutsos. "Hilbert R-tree: An improved R-tree using fractals," Proc of the 20th VLDB, pp.500-509, 1994.

[7] A.G. Li, Z.H. Zhang. "RS-tree: A dynamic index structure," Proc of Int Conf on Information Engineering and Computer Science, pp. 1112-1115, 2009.

[8] Y. Garcia, M. Lopez, S. Leutenegger. "On Optimal Node Splitting for R-trees," Proc of the 24th VLDB, pp. 334-344, 1998.

[9] Y.J. Fu, J.C. Teng, S.R. Subramanya. "Node Splitting Algorithms in Tree-Structured High-Dimensional Indexes for Similarity Search," Proc of the 2002 ACM symposium on Applied computing, pp. 766-770, 2002.

[10] H.K. Ahn, N. Mamoulis, H.M. Wong. "A Survey on Multidimensional Access Methods," UU-CS, Utrecht, The Net herlands: Technical Report 2001-14, 2001.