International Journal of Computer Theory and Engineering, Vol. 2, No. 4, August, 2010
1793-8201

SlimSS-tree: A New Tree Combined SS-tree With Slim-down Algorithm

Lifang Yang, Xianglin Huang, Rui Lv and Hui Lv

***Abstract*—Nowadays, the appearance of a great number of multimedia data brings the need for more effective methods to manage these data. The multimedia retrieval systems always index these data based on feature vectors, and the index structures such as the R-tree family are used to manage them more efficiently. Slim-down algorithm is used in Slim-tree, and it can improve the disk access number for range queries in average 10%~20% for vector datasets. In this paper, we use Slim-down algorithm in both SS-tree and R-tree index structures, and propose a new structure: SlimSS-tree. Experiment results show that compared with SS-tree, the disk access number of SlimSS-tree for k nearest neighbor search improves by 20%~30% in average for vector datasets of 32 dimensions, and the SlimSS-tree can provide fast query performance as well. But the Slim-down algorithm is not suitable for reducing the intersection of rectangle regions as it does to the sphere regions. It's not efficient in rectangle node management.**

***Index Terms*—index structure; SlimSS-tree; R-tree; SS-tree; Slim-down; Reinsertion**

I. Introduction

Recently, the number of multimedia data such as video, voice, image, text, and numerical data has drastically increased. This results the development of multimedia retrieval systems to manage them. In order to satisfy the information needs of users, it is very important for retrieval system to know which portions of the database are relevant to user's requests. In particular, there is an urgent need of indexing techniques to support similarity queries.

In this paper, we focus on the R-tree[1] family. The most successful variant of R-tree seems to be R*-tree[2] which imports the idea of 'Forced-Reinsert' by deleting some rectangles from the overflowing node, and Reinserting them into the tree. The Reinsertion algorithm improves the query performance, and it was used in some other R-tree variants, such as SS-tree.

Slim-down algorithm is used in Slim-tree[3]. It could diminish the number of objects that fall within the intersection of two regions in the same level, sometimes even decrease the number of nodes in the tree. Thus it can produce a 'tighter' tree and improve the search performance evidently. It can improve the number of disk accesses for range queries in average 10%~20% for vector datasets. Especially for datasets with bigger bloat-factors, the average improvement goes to 25%~35%. In this paper, we use Slim-down algorithm in SS-tree index structure and propose a new structure: SlimSS-tree. We also use the Slim-down algorithm in R-tree index structure to test whether it's efficient in managing rectangle nodes, and compare it with Reinsertion algorithm.

Lifang Yang, Xianglin Huang, Rui Lv and Hui Lv are withComputer School, Communication University of China,Beijing, China(email: huangxl@cuc.edu.cn).

The rest of this paper is organized as follows. Section 2 gives a brief introduction to the R-tree family methods. In section 3 we describe the index structure of SlimSS-tree using advanced Slim-down algorithm combined Slim-down with Reinsertion. Then the experiment results about the performance of SlimSS-tree, SS-tree and R-tree with Reinsertion and Slim-down algorithm is presented in section 4. Finally, we conclude this paper in section 5.

II. R-tree Family of Index Structure

The R-tree is a completely dynamic index structure, proposed by Guttman in 1984[1]. It is a height-balanced tree similar to B-tree for multi-dimensional spatial objects. In order to maintain its balanced structure, it allows region overlapping among sibling nodes, which is its major drawback. Later, lots of work has done to avoid the performance degradation caused by region overlapping. And some new index structures, such as R^+-tree, R*-tree, X-tree, SS-tree, SR-tree and A-tree are generated. Most of them have the similar structure and properties as R-tree.

R^+-tree is proposed by Sellis etc. in 1987[4]. The great different between R-tree and R^+-tree is that the R^+-tree allows partitions to split rectangles, then zero overlap among intermediate node entries can be achieved. In the process of splitting node with rectangle *G*, it allows partitions to split rectangles which are in lower level of *G* into a collection of non-overlapping sub-rectangles, then zero overlap among the two new generated nodes can be achieved. Compare with R-tree, R^+-tree exhibit good search performance at the expense of some extra space, especially for point queries.

R*-tree is presented by Beckmann and Kriegel in 1990[2]. In this structure, the idea 'Forced-Reinsert' is proposed. That is when a node is overflowed, it checks whether this given level has executed Reinsertion before. If Reinsertion isn't executed before, it chooses some farthest entries of this node, deletes them and Reinserts them into the tree. Otherwise, it executes the split algorithm. Also the factors of area, margin and region overlapping etc. are taken into consideration for node split. Due to the concept of 'Forced-Reinsert', the storage utilization is higher and the R*-tree clearly outperforms other R-tree variants proposed before.

X-tree is proposed by Berchtold, Keim and kriegel in 1996[5]. The X-tree uses a split algorithm minimizing overlap and new concept of Supernode to reduce the overlap of bounding rectangles caused by increase of dimension. It is based on the R*-tree. The mainly difference between these two structures is that when a node needs to be split, the R*-tree always split the node according to the factors of area, margin and region overlapping etc.. But for the X-tree, if the

split for minimizing overlap can't be found, the node remains unsplit and a Supernode is generated. Therefore, the overlap is reduced. In low dimensional space, the X-tree has similar performance as R*-tree. But in high dimensional space, for the number of Supernodes increases, the X-tree outperforms the R*-tree significantly.

David, Ramesh etc. proposed SS-tree in 1996[6]. The SS-tree uses super sphere to represent the nodes and objects replacing the super rectangles used in R-tree, R*-tree etc.. The result of their tests shows that on higher dimensional data (> 5D) SS-tree provides faster query performance than the R*-tree.

SR-tree is presented by Katayama and Satoh in 1997[7]. In paper [7], they have proved that bounding spheres occupy much larger volume than bounding rectangles, thus the search efficiency is reduced. In order to overcome this drawback, a region of the intersection of a bounding sphere and a bounding rectangle is proposed in [7] to represents the nodes. Their test result shows that the SR-tree outperforms both the SS-tree and R*-tree.

Sakurai, Yoshikawa etc. proposed the A-tree in 2000[8]. The basic idea of A-tree is the use of Virtual Bounding Rectangles (VBRs), which contain and approximate MBRs (Minimum Bounding Rectangles) and data objects. Because of the compact VBRs, each node can install large number of entries, which means the fanout of nodes become large, thus lead to fast search. The test results using both synthetic and real data sets show that the A-tree outperforms the SR-tree in all range of dimensionality up to 64.

III. SLIMSS-TREE INDEX STRUCTURE

A. The structure of SlimSS-tree

SlimSS-tree is proposed based on the SS-tree, which uses the super spheres to manage the nodes instead of the super rectangles. The SlimSS-tree is a completely dynamic index structure as R-tree, and has similar structure and properties as R-tree. Let M and m ≤ M/2 be the maximum number and minimum number of entries in one node respectively, and each node(not include the date node) in SlimSS-tree contains entries with the form of SSTreeElem structure, as shown in figure 1. The SlimSS-tree satisfies following properties:

(1) Every node must contain at least *m* and at most *M* entries, unless it is the root node.
(2) The root should contain at least two entries, unless it is a leaf node.
(3) For each entry representing by SSTreeElem structure as figure 1, *childptr* represents the pointer of this entry to its child node; *update_count* is used for the entry to be periodically recalculated when its child node changed; *radius* is the radius of the enclosing sphere of its child node; and *centroid[Dim]* is the mean value of all its child entries' centroids.
(4) All leaves appear on the same level.

In our design, The SlimSS-tree mainly consists of intermediate nodes, leaf nodes and date nodes. The date nodes are used to store the date objects (Here they are feature vectors), and each intermediate node or leaf node is composed by the super spheres which completely enclose all super spheres of its lower level nodes, as shown in figure 2. Figure 2 shows one example of SlimSS-tree structure. In figure 2(a), each point stands for one feature vector and each super sphere stands for one entry of a node(not include the date node). The index structure showed in figure 2(b) is the corresponding SlimSS-tree structure for figure 2(a). Here m=2 and M=4.

```
structure SSTreeElem {
{
    BYTE *childptr;        // Child pointer
    int update_count;   // Refresh value
    float radius;           // The radius of   enclosing sphere
    float centroid[ Dim ];    // The mean value of its child entries' centroids.
};
```

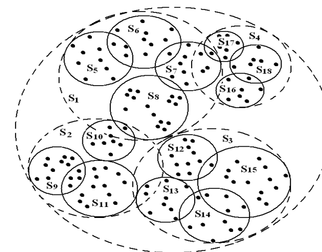
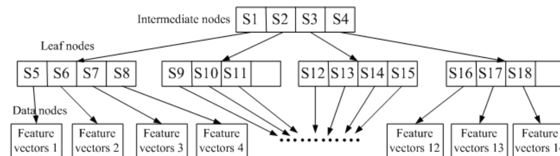


Figure 1. The SSTreeElem structure

(a) The data set of feature vectors



(b) The SlimSS-tree structure of (a)

Figure 2. The example of SlimSS-tree structure

B. Insertion Algorithm

The insertion algorithm of SlimSS-tree is similar to the insertion algorithm of R-tree. There are three main differences:

(1) The split algorithm used here simply finds the dimension with the highest variance, and then chooses the split location to minimize the sum of the variances on each side of the split[6].
(2) When a node N is splited into two new node N_1 and N_2, the original node N will be replaced with the new generated node N_1 or N_2 whose centroid is nearer to N's centroid.
(3) In each node(not include the date node) of SlimSS-tree, there is an *update_count* parameter for each entry. Only when the *update_count* parameter of an entry is multiple of UPDATE (A number can be set), the *centroid* and *radius* parameters of this entry and its forefather entries need to be adjusted. The influence of parameter UPDATE to the performance of SlimSS-tree will be given in section 4.

Insertion Algorithm: Insert one feature vector v to the SlimSS-tree with root r.
BEGIN
  Set N to be the root r.
  WHILE (N is not a leaf node)
    Find the entry n_p whose *centroid* is nearest to v in N.
    Set N to be the child node of entry n_p.
  END WHILE //N is a leaf node
  Find the entry n_p whose *centroid* is nearest to v in N.

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG

Set N to be the child node of entry n_p. Then N is a date node.
IF (N is not full)
Insert v to N.
The entry number of N increases one.
n_p.*update_count* increases one.
IF (n_p.*update_count* % UPDATE = 0)
Adjust the *centroid* and *radius* parameters of n_p and its forefather entries.
ELSE // N is full
Split date node N into two new date node N_1 and N_2.
IF (the centroid of N_1 is nearer to the centroid of N than N_2)
Replace node N with N_1. And adjust the *centroid* and *radius* parameters of its father entry n_p and n_p's forefather entries.
Insert node N_2 to the tree.
ELSE
Replace node N with N_2. And adjust the *centroid* and *radius* parameters of its father entry n_p and n_p's forefather entries.
Insert node N_1 to the tree.
END

The insertion process of inserting one node N to the tree is similar to the feature vector insertion process. In the node insertion process, the entry whose *centroid* is nearest to the centroid of the inserted node N will be chosen for inserting the node. And N will be inserted to the level with the same level flag as N. After the insertion of all feature vectors, the *centroid* and *radius* parameters of all entries in SlimSS-tree need to be adjusted for the reason of the *update_count* parameter.

C. Advanced Slim-down Algorithm

After all feature vectors are inserted to the SlimSS-tree and the centroid and radius parameters of all entries has been adjusted, the advanced Slim-down algorithm which combins Slim-down with Reinsertion will be used to post-process the tree to make it more tighter. Figure 3 shows how slim-down algorithm works on one node. In figure 3, entry c is the farthest entry of node i, and it is also included in node j. The Slim-down algorithm moves it from node i to node j. So the radius of node i is reduced, thus the intersection region between node i and node j is reduced as well.

Advanced Slim-down algorithm: Using the Slim-down algorithm combined with Reinsertion to post-process the SlimSS-tree nodes in date Node level.
BEGIN
FOR (each node i in date node level of the SlimSS-tree)
WHILE (node i has entries not less than m)
Find the farthest entry c from the centroid of i.
Find a sibling node j of i, which also covers c.
IF (j exists and it is not full)
Remove c from i to j. The entry number of i decrease one, and the entry number of j increase one.
Adjust the *centroid* and *radius* parameters of node i's, j's father entries and their forefather entries.
ELSE
Run out of the loop of **WHILE**
END WHILE //stop Slim-down node i
IF (the number of entries in node i is less than m)
Delete node i from its parent node, the entry number of its parent node decreases one, and adjust the *centroid* and *radius* parameters of its forefather entries.
FOR (each entry in node i)
Push it to the Reinsertion stack.
FOR (from leaf level to the root level)
FOR (each node i in this level)
IF (the entry number of node i is smaller than m)
Delete node i from its parent node, the entry number of its parent node decreases one, and adjust the *centroid* and *radius* parameters of its forefather entries.
FOR (each entry in node i)
Reinsert it to the tree directly.
FOR (every entry in the Reinsertion stack)
Reinsert it to the tree.
Adjust the *centroid* and *radius* parameters of all entries in SlimSS-tree.
END

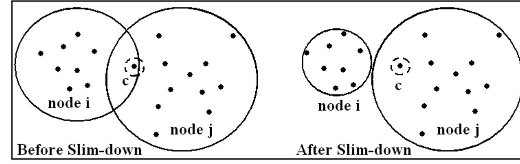


Figure 3. How the Slim-down algorithm works

D. Nearest Neighbor Queries

Give a domain of feature value D — the indexed feature vectors, a query vector Q∈D, and an integer k≥1, the KNN query NN(Q, k) selects the k indexed feature vectors which have the shortest distance from Q.

MINDIST algorithm[9] is one of the most popular nearest neighbor search algorithms. The MINDIST (Minimum Distance) of a point $Q(q_1, q_2,\ldots, q_d)$ in Euclidean space E(d) from a sphere $S(O, r)$ in the same space is defined as:

$$MINDIST(Q,S)=\begin{cases}0, & if\ d(Q,O)\le r\\ d(Q,O)-r, & others\end{cases} \tag{1}$$

Where, $O(o_1, o_2,\ldots, o_d)$ is the centroid of the sphere S, r is the radius of S, and

$$d(Q,O)=\left(\sum_{i=1}^{d}\left|q_i-o_i\right|^2\right)^{1/2} \tag{2}$$

Let *KNNlist* to be the current k indexed feature vectors nearest to query Q, and *KNNdist* to be the distances of the k indexed feature vectors to Q, d_{max} to be the kth shortest distance from the indexed feature vectors to Q, and *entrylist* to be the list for entries which may contain the NN(Q, k).

KNN Query Algorithm: Find the k indexed feature vectors which have the shortest distance from query vector Q in SlimSS-tree.
BEGIN
Clear the *KNNlist* to be NULL, set all the elements of *KNNdist* to be MAX, initiate d_{max} to be MAX, and use entries in the root of SlimSS-tree to initiate the *entrylist*. Then calculate the MINDISTs of Q from every entry in the *entrylist*, and sort the entries according to the MINDISTs in increasing order.
WHILE (the *entrylist* is not empty)
IF (the first entry in *entrylist* is one entry of a leaf node)
FOR (every indexed feature vector O_j in this entry)
Calculate the distance of Q and O_j: $dist_j$
IF ($dist_j < d_{max}$) THEN
Delete the last element in *KNNdist* and *KNNlist*. Insert $dist_j$ to *KNNdist* in increasing order. Insert O_j to *KNNlist* in the order as $dist_j$ in *KNNdist*. Let d_{max} to be the last element in *KNNdist*.

ELSE IF (the first entry in *entrylist* is the entry of an intermediate node)
In *entrylist*, stead the first entry with the entries of its lower level. Calculate the MINDISTs of *Q* from every entry in the *entrylist*, and sort the entries according to the MINDISTs in increasing order. Delete the entries whose MINDISTs are larger than d_{max} from the *entrylist*.
END WHILE
The indexed feature vectors in *KNNlist* are the NN(*Q*, *k*).
The distances in *KNNdist* are the *k* shortest distance to query *Q*.
END

IV. Experiment Result

In this section, we provide experiment results of the performance of the SlimSS-tree with different UPDATE parameter, and compare the SlimSS-tree with the SS-tree index structures (include SS-tree and SS-tree without Reinsertion) and R-tree index structures (include R-tree, R-tree with Reinsertion and R-tree with Slim-down). We implemented these trees in VC6.0 under Windows on a Dell OPTIPLEX 360 computer with Intel Core2 Duo CPU E7400 2.8GHz and RAM 3.0GB. The datasets used in our test are loaded from the website http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html. These datasets contain four sets of image features: the color histogram, color histogram layout, color moments, and co-occurence texture, extracted from a Corel image collection of 68040 pictures. The page size of the index structure node is 4096 Bytes in our test. In paper [2], it has proved that *m*=40%**M* yields the best performance. So *m*=40%**M* is used in all kinds of nodes in our design. In the following figures, we report the average number of disk accesses obtained from 60 points KNN queries, and all of the query points are randomly extracted from the 68040 feature vectors for each dataset.

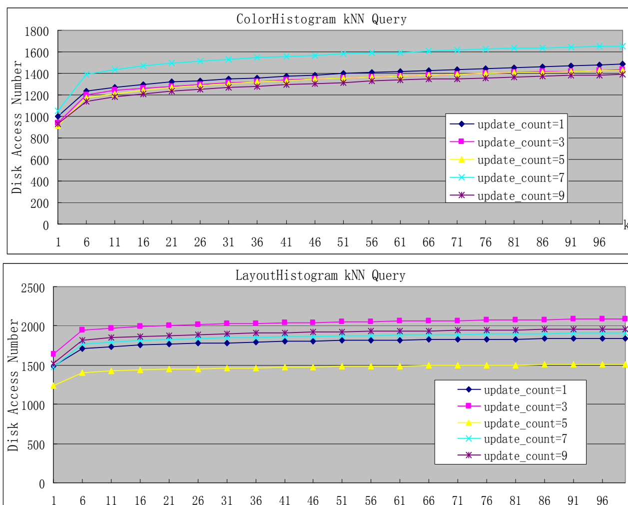
A. The performance of SlimSS-tree with different UPDATE parameter

The most important characteristics of these datasets and the execution time for building SlimSS-tree and query with different UPDATE parameter is shown in table 1. Here the execution time of query is the total execution time for 60 point *k*=50 nearest neighbor Queries. Figure 4 shows the performances of SlimSS-tree with different UPDATE parameter for KNN query, where *k* ranges from 1 to 100 with step 5. The execution time for adjusting tree is reduced with the increase of UPDATE parameter. As a result, the execution time for building SlimSS-tree is reduced. At the same time, the *centroid* and *radius* parameter of each entry is not very accurate, which may influence the execution time for query (as shown in table 1) and the performance of SlimSS-tree (as shown in figure 4). We can see from table 1 and figure 4 that when UPDATE=5, the SlimSS-tree achieves the best performance. So the SlimSS-tree, SS-tree index structures are all built in the condition UPDATE=5 next section.

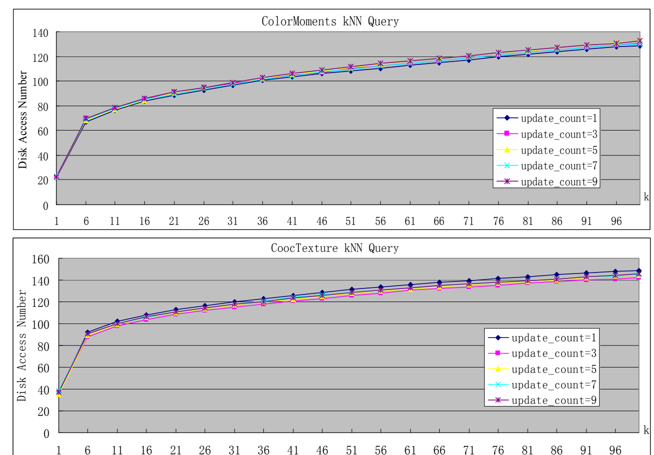
TABLE I. Characteristics of Feature Datasets and Execution Time for Building SlimSS-Tree and Query

Feature datasets	Dimension	UPDATE =1		UPDATE =3		UPDATE =5		UPDATE =7		UPDATE =9	
		Build	query	Build	query	Build	query	Build	query	Build	query
ColorHistogram dataset	32	4.23s	9.07s	2.70s	7.67s	3.59s	8.79s	2.29s	11.26s	2.20s	7.53s
LayoutHistogram dataset	32	3.90s	12.43s	2.53s	9.73s	2.21s	6.18s	2.10s	7.70s	2.11s	6.64s
ColorMoments dataset	9	2.25s	5.32s	1.29s	1.23s	1.11s	1.21s	1.01s	1.34s	0.96s	1.06s
Co-ocTexture dataset	16	2.73s	6.51s	1.71s	1.31s	1.51s	1.21s	1.43s	1.11s	1.42s	1.48s

Notes: The query time is the total execution time for 60 query vectors for *k*=50 nearest neighbor queries.



(a) Experiment result for ColorHistogram dataset
(b) Experiment result for LayoutHistogram dataset



(c) Experiment results for ColorMoments dataset
(d) Experiment result for Co-ocTexture dataset

Figure 4. Performance of SlimSS-tree with different UPDATE parameter

IACSIT International Association of Computer Science and Information Technology WWW.IACSIT.ORG

TABLE II. CHARACTERISTICS OF THE BUILT TREES

Feature datasets		SS-tree without Reinsertion	R-tree	SS-tree	R-tree with Reinsertion	SlimSS-tree	R-tree with Slim-down
ColorHistogram dataset	*H*	3	4	3	4	3	4
	I	175	322	154	290	100	322
	V	3235	3089	3052	2832	1905	3089
LayoutHistogram dataset	*H*	3	4	3	4	3	4
	I	157	341	134	321	96	341
	V	2971	3066	2786	2848	1854	3061
ColorMoments dataset	*H*	2	2	2	2	2	2
	I	17	31	14	27	16	31
	V	941	935	884	852	924	935
Co-ocTexture dataset	*H*	2	3	2	3	2	3
	I	44	85	40	74	41	85
	V	1617	1591	1533	1425	1568	1591

Notes: *H* represents the height of the tree, *V* represents the data node number of the tree, *I* represents the number of intermediate nodes and leaf nodes.

TABLE III. EXECUTION TIME FOR BUILDING TREES AND QUERY

Feature datasets	R-tree		SS-tree without Reinsertion		R-tree with Reinsertion		SS-tree		R-tree with Slim-down		SlimSS-tree	
	Build	query	Build	query	Build	query	Build	query	Build	query	Build	query
ColorHistogram dataset	4.50s	87.14s	1.50s	26.06s	6.68s	73.79s	2.15s	20.00s	4.53s	88.00s	2.42s	7.20s
LayoutHistogram dataset	4.37s	188.32s	1.89s	9.34s	6.28s	189.73s	1.95s	9.04s	4.42s	185.04s	2.26s	2.82s
ColorMoments dataset	3.92s	0.39s	1.31s	0.31s	4.51s	0.37s	1.48s	0.28s	4.21s	0.39s	1.10s	0.31s
Co-ocTexture dataset	3.98s	0.79s	1.62s	0.26s	5.11s	0.54s	1.60s	0.23s	4.03s	0.76s	1.56s	0.23s

Notes: The query time is the total execution time for 60 query vectors for k=50 nearest neighbor queries. And the execution time of building SlimSS-tree, SS-trees is test in the condition UPDATE=5.

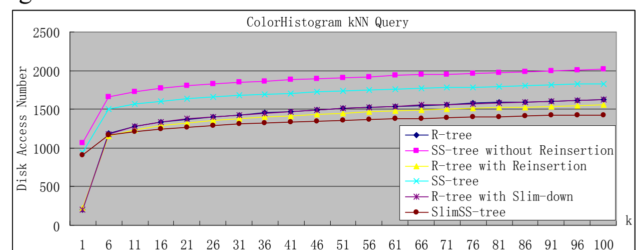
B. The performance of SlimSS-tree, SS-tree index structures and R-tree index structures

In this paper, the quadratic split algorithm is used in our R-tree index structures, and the R-tree with Slim-down is generated by executing the Slim-down algorithm in the date node level of R-tree. We don't provide details since paper [1] gives the description about R-tree index structure and the introductions about the Reinsertion algorithm and Slim-down algorithm can be found in paper [2] and paper [3] respectively.

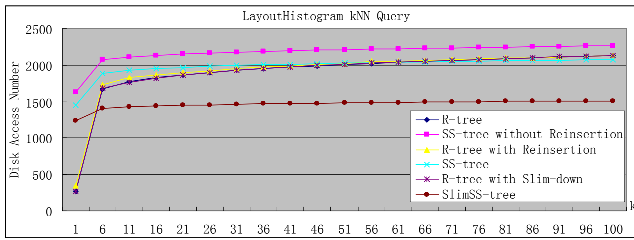
Table 2 shows the most important characteristics of the built SlimSS-tree, SS-trees and R-trees. In table 2, it is evident that both the node number of the R-tree and SS-tree decreases for the use of Reinsertion algorithm, and the node number of SlimSS-tree reduces obviously in high dimensional space for using the advanced Slim-down algorithm. But Slim-down algorithm is not useful for reducing the node number of R-tree structure as it does to SlimSS-tree. In table 3, it is evident that both the execution time for building SS-tree structures and query in SS-tree structures is less than the execution time for building R-tree structures and query in R-tree structures. And the execution time for query in SlimSS-tree is the least for dimension 32. Here the execution time of query is the total execution time for 60 points k=50 nearest neighbor Queries.

We can see from figure 5 that the search performances of R-trees, SS-trees and SlimSS-tree are degraded clearly with the increase of dimensionality (as figure 5(a), 5(b), 5(c) and 5(d)). In high dimensional space when the nearest neighbor number k>6, the SlimSS-tree proposed in this paper outperforms the other five index structures, especially outperforms the SS-tree and SS-tree without Reinsertion (as figure 5(a), 5(b)). Compared with the SS-tree, the disk access number of SlimSS-tree improves by 20%~30% in average in high dimensional space. The main reason is that as the increase of dimensionality, the overlap regions between nodes increases and the advanced Slim-down algorithm can effectively reduce the intersection regions between spheres very well, thus improve the search performance of SlimSS-tree obviously. In the low dimensional space, it is evident that the SlimSS-tree and SS-tree sturctures outperform the R-tree structures (as figure 5(c), 5(d)). As shown in figure 5, when the dimension of feature vectors is low or the number of Nearest Neighbors is small, the SlimSS-tree can almost has the same performance as SS-tree structures as well.

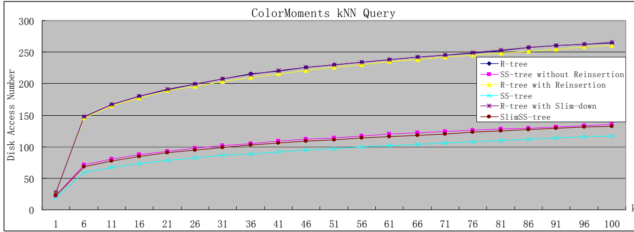
It is clear in figure 5 that the Reinsertion algorithm can improve the search performance of SS-tree and R-tree structures. But the R-tree with Slim-down algorithm almost has the same performance as the original R-tree. The Slim-down algorithm can't improve the search performance of R-tree as it does to the SlimSS-tree, mainly for the reason that most intersection regions of MBRs of R-tree are not in the edge parts, thus the farthest feature vector of one data node is not contained in its sibling nodes and the execution of Slim-down stopped. So it can't decrease the intersection regions for rectangle nodes. Table 4 gives the successful times and failed times for the executing of Slim-down algorithm in R-tree.



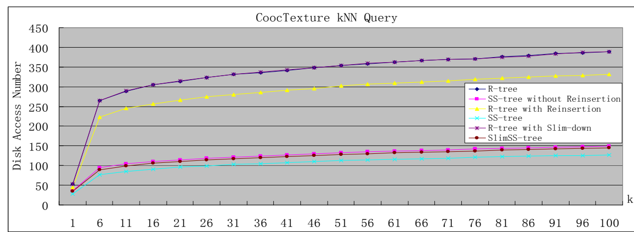
(a) Experiment result for ColorHistogram dataset



(b) Experiment result for LayoutHistogram dataset



(c) Experiment result for ColorMoments dataset



(d) Experiment result for Co-ocTexture dataset

Figure 5. The performance of R-trees, SS-trees and SlimSS-tree

TABLE IV. THE EXECUTING SITUATION FOR SLIM-DOWN IN R-TREE

Feature datasets	Successful Times	Failed Times
ColorHistogram datasets	27	55
LayoutHistogram datasets	22	9
ColorMoments datasets	8	3
Co-ocTexture datasets	0	2

Notes; Successful Times represents the Slim-down algorithm successfully remove the farthest entry in one node to its sibling node in R-tree. Failed Times represents the Slim-down algorithm failed to execute Slim-down, for its sibling node is full.

V. CONCLUSION

In this paper, we use the Slim-down algorithm combined with Reinsertion in SS-tree, and propose a new index structure: SlimSS-tree. In the high dimensional space, the SlimSS-tree outperforms the SS-tree obviously, while in low dimensional space it almost has the same performance as the SS-tree. The main reason is that the overlapping regions between nodes increases as the increase of dimensionality of feature vectors, and the Slim-down algorithm can effectively reduce the intersection regions between spheres, thus improve the search performance of SlimSS-tree obviously. We also compare our proposed SlimSS-tree, SS-tree, with the R-tree structures. Experiment results show that the execution time for building SlimSS-tree, the SS-tree structures and query in them is evidently less than the R-tree structures. We can also see that the Slim-down algorithm can not improve the performance of R-tree clearly as it does to the SlimSS-tree. It is not suitable for reducing the intersection of rectangle regions, mainly for the reason that most intersection regions of MBRs are not in the edge portions of MBRs.

ACKNOWLEDGMENT

This paper is supported by the program for New Century Excellent Talents in University (NCET-07-0768) and Project 211 of China.

REFERENCES

[1] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47-57.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. ACM SIGMOD, 1990, pp. 322-331.

[3] Caetano Traina Jr., Agma Traina, Bernhard Seeger, Christos Faloutsos. Slim-trees: High Performance Metric Trees Minimizing Overlap Between Nodes. Proc. of the 7th Int. Conf. on Extending Database Technology: Advances in Database Technology, 2000, pp. 51-65.

[4] Timos Sellis, Nick Roussopoulos and Christos Faloutsos. The R+-tree: A Dynamic index for Multi-dimensional Objects. Proc. of the 13th VLDB Conf., Brighton, 1987, pp. 507-518

[5] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel. The X-tree: An Index Structure for High-dimensional Data. Proc. of the 22nd VLDB Conf. Mumbai (Bombay), India, 1996, pp. 28-39

[6] David A. White, Ramesh Jain. Similarity Indexing with the SS-tree. Proc. of the 12th Int. Conf. on Data Engineering, 1996, pp. 516-523.

[7] Norio Katayama, Shin'ichi Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 1-12.

[8] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, Haruhiko Kojima. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. Proc. of the 26th VLDB Conf., 2000, pp.

[9] Nick Roussopoulos, Stephen Kelley, Frederic Vincent. Nearest Neighbor Queries. Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995, pp. 71-79.

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG