SC-Tree: An Efficient Structure for High-Dimensional Data Indexing

Ben Wang and John Q. Gan

Department of Computer Science, University of Essex,
Colchester CO4 3SQ, UK
{bwangm, jqgan}@essex.ac.uk

Abstract. In content-based information retrieval (CBIR) of multimedia data, high-dimensional data indexing and query is a challenging problem due to the inherent high dimensionality of multimedia data. As a data-based method, metric distance based high-dimensional data indexing has recently emerged as an attractive method because of its ability of making use of the properties of metric spaces to improve the efficiency and effectiveness of data indexing. M-tree is one of the most efficient indexing structures for searching data from metric space, and it is a paged, balanced, and dynamic tree that organizes data objects in an arbitrary metric space with fixed sizes for all its nodes. However, inherent disadvantages are veiled in the M-tree and its variants, which prevent them from further improvement of their indexing and query efficiency. To avoid these disadvantages, this paper proposes a sorted clue tree (SC-tree), which essentially modifies the nodes, entries, indexing algorithm, and query algorithm of the M-tree but reserves its advantages. Experimental results and complexity analyses have shown that the SC-tree is much more efficient than the M-tree with respect to the query time and indexing time without sacrificing its query accuracy.

1 Introduction

Efficient access is essential for content-based information retrieval of large multimedia databases because multimedia data are usually characterized by high-dimensional features which bring about the curse of dimensionality problem in similarity searching operation.

There are two general categories of partitioning methods for data indexing: space-based partitioning and data-based partitioning. Space-based partitioning [12] [14][15][16][17] is also called grid-based partitioning, which partitions each dimension of the space into intervals and thus the whole space into grids. Although this is a simple partitioning method, the number of grids increases exponentially with the space dimension, resulting in the curse of dimensionality problem. Data-based partitioning [3][4][7] can also be called prototype-based partitioning, clustering-based partitioning, or distance-based partitioning. The number of partitions in data-based partitioning depends on the data distribution, e.g., the number of clusters, which is not directly related to the space dimension. Therefore, data-based partitioning does not

D. Bell and J. Hong (Eds.): BNCOD 2006, LNCS 4042, pp. 164 – 176, 2006.

have the curse of dimensionality problem and has found wide applications in high-dimensional data indexing.

As a data-based method, metric distance based high-dimensional data indexing has recently emerged as an attractive method because it is able to make use of the properties of metric spaces to improve the efficiency and effectiveness of data indexing [3][4][19]. Typical metric distance based indexing structures include vantage point tree (VP-tree) [18], multiple vantage points tree (MVP-tree) [5], geometric near-neighbour access tree (GNAT) [6], and paged metric tree (M-tree) series [2][7]. VP-tree partitions a data set according to distances that the objects have with respect to a vantage point, and then utilizes the triangle inequality to filter data objects to reduce the similarity search cost. However, due to its small fan-out, VP-tree structure is very deep, thus a search operation is time-consuming. MVP-tree employs multiple vantage points, and exploits pre-computed distances to reduce the number of distance computations during query process, but it is static and cannot be incrementally updated. GNAT captures the geometry of a data set by hierarchically breaking it down into regions. Long preprocessing time is the main disadvantage of GNAT. M-tree [7] is a paged, balanced, and dynamic tree that organizes data objects in an arbitrary metric space with fixed sizes for all its nodes. Since metric spaces strictly cover vector spaces, M-tree has a far more general applicability than multi-dimensional access methods, such as R-tree [13] and its variants [10]. For instance, a set of strings can be compared and organized in the M-tree according to edit distance which is defined as the minimal number of character changes needed to transform one string into another. In recent years, four important improvements have been made to the original M-tree: complex similarity search, approximate search, cost models, and user-defined distances. Complex similarity search handles several features, such as color, shape, or texture [8]. Approximate searching introduces PAC-NN (probably approximately correct nearest neighbor) queries, where error bound and accurate ratio can be tuned during a query period to trade query accuracy for query time [9]. Cost models concern the distance distribution of objects and predict both I/O and CPU query costs [11]. The user-defined distance approach develops a QIC-M-tree (QIC stands for query, index, comparison distances), which involves several distinct metrics at the same time [10]. However, three inherent disadvantages are veiled in the M-tree and its variants, which prevent the M-tree and its invariants from further improvement of their indexing and query efficiency. To avoid these disadvantages, this paper proposes a sorted clue tree (SC-tree), which essentially modifies the nodes, entries, indexing algorithm, and query algorithm of the M-tree but reserves its advantages.

In this paper, the M-Tree indexing structure is briefly introduced in section 2. The SC-tree is proposed in section 3. Experimental results and analyses are given in section 4. And conclusion is given in section 5.

2 M-Tree

M-tree is an efficient indexing structure for searching data from metric space [7]. A generic metric space is a pair, M = (U, d), where U is a domain of feature vectors and

d is a distance function with the following postulates: symmetry, positivity, and triangle inequality:

$$d(x, y) = d(y, x). \text{ (Symmetry)} \tag{1}$$

$$d(x, y) \geq 0 \text{ and } d(x, y) = 0 \text{ iff } x = y. \text{ (Positivity)} \tag{2}$$

$$d(x, y) + d(y, z) \geq d(x, z). \text{ (Triangle inequality)} \tag{3}$$

For the sake of self-containment, this section briefly describes the indexing structure and indexing and query algorithms of the M-tree.

2.1 Indexing Structure

M-tree indexing structure is constructed by hierarchical nodes. Each node consists of a fixed number of entries. There are two types of nodes: internal nodes and leaf nodes, corresponding to two types of entries. An internal entry, stored in internal nodes, contains a routing object, covering radius, a pointer to its sub-tree (a node at the next level), and the distance between the routing object and its parent. The routing object is defined as the representative centroid of objects in the sub-tree, and the covering radius is the farthest distance between any objects in the sub-tree and the routing object. However, in a leaf entry, an object identifier, its feature vector, and the distance between the object and its parent are recorded.

The formal definitions of the M-tree node, leaf entry, and internal entry are as follows: An M-tree node has a fixed number of entries, defined as $entries_i$, $i < numOfEntries$. A leaf entry has the format of $[O_i, oid(O_i), d(O_i, P(O_i))]$, where O_i is defined as the feature vectors of the routing object, $oid(O_i)$ as the object identifier, and $d(O_i, P(O_i))$ as the distance between O_i and its parent object $P(O_i)$. An internal entry has the format of $[O_r, r(N_r), ptr(N_r), d(O_r, P(O_r))]$, where O_r is defined as the routing object, $r(N_r)$ as the covering radius of sub-tree N_r, and $ptr(N_r)$ as the pointer to N_r. For each O_i' in the sub-tree rooted at N_r, it has the property $d(O_r, O_i') \leq r(N_r)$.

2.2 Indexing Algorithm

The indexing algorithm of the M-tree inserts data objects into its nodes one by one. The insert algorithm recursively locates the most suitable internal or leaf node to accommodate a new data object. The strategy to find the most suitable node is to minimize the enlargement of the covering radius of the entries at each level. If a node is full of entries, the split algorithm will be called to deal with the overflow situation. Regardless of the specific split policy, the semantics of covering radius has to be preserved after each splitting operation.

In general, the indexing algorithm of the M-tree follows a bottom-up approach. Initially, an empty root node is generated. The first leaf entry is generated by selecting an object from the data set and inserting it into the root node. A leaf entry is inserted into a node if the node is not full. Otherwise, the split algorithm partitions the node into two sub-trees and a new node is generated at the same time. From these sub-trees, two routing objects are chosen as the new internal entries, whose pointers point to the

sub-trees respectively. These two internal entries are then inserted into the new node. At this moment, the first M-tree with one root node and two sub-trees is formed. After that, the second leaf entry is generated by selecting another object form the data set and inserting it into the root node. The covering radius of the inserted entry in the root node should be updated. If the current node (the root node in this case) has sub-trees, the entry is recursively inserted into one of the sub-trees until a leaf node is reached. If the leaf node is full, the split algorithm has to be called. Otherwise, the entry is inserted into the leaf node. Following the above procedure, all the objects in the data set are inserted into the M-tree indexing structure.

2.3 *k*-NN Query Algorithm

The *k*-NN query algorithm retrieves *k* most similar objects with respect to a given query object *Q*. A priority queue *PR* and an array *NN* with *k* elements are utilized in the algorithm. The *PR* is a queue of pointers to active sub-trees where qualified objects can be found. A lower bound that records the distance between any object in the sub-trees and the query object *Q* is also kept in the *PR*, and the node with the minimal lower bound will be chosen. Since the pruning criterion of *k*-NN query algorithm is dynamic, the search radius is the distance between *Q* and its current *k*-th nearest neighbor. The order of the accessing nodes is crucial for high query performance. The query algorithm starts from the root node. It firstly locates active sub-trees of the root node and their lower bounds, and inserts them into the *PR*. After that, the query algorithm chooses one sub-tree from the *PR,* stores the node identifiers and the distances from the query *Q* in the array *NN*, and returns a *k*-NN value, d_k., which is used later as the search radius to remove sub-trees in the *PR* whose lower bounds exceed d_k. At the end of execution, the *k*-NN query results are stored in the array $NN[i] = [oid(O_i), d(O_i,Q)]$, $0 \le i < k$, where $oid(O_i)$ is the object identifier of the *i*-th nearest neighbor of *Q* and $d(O_i,Q)$ represents the distance of the *i*-th nearest neighbor from *Q* [10].

2.4 Advantages and Disadvantages

M-tree has the following major innovative properties [9]. It is a balanced and incremental updating indexing structure that is able to index data sets from generic metric spaces. It is also dynamic and scalable. The *k*-NN query can be performed on the M-tree, with query results ranked in terms of the distances with respect to a given query object. It is suitable for indexing high-dimensional data.

However, M-tree has three inherent disadvantages that largely limit its indexing and query efficiency. Firstly, the entries in a node are stored randomly. As a result, the split algorithm has to find two farthest objects by comparing every pair of objects in the entry, which is obviously not efficient for the splitting operation. Secondly, in both the insert and split algorithms, locating the parent of the current node is needed frequently, but the searching has to inefficiently travel from the root node to all sub-trees until the current node is located [9]. Thirdly, for *k*-NN search algorithm, the chosen node is added into the priority queue *PR* without sorting the position of sub-trees according to the lower bounds between the query object and sub-trees. As a result, it influences the order of accessing nodes. The first two disadvantages will

largely decrease the indexing efficiency, and the second and the third disadvantages will add much unnecessary query time. In order to improve the indexing and query efficiency for the M-tree, the SC-tree is proposed in the next section.

3 SC-Tree

SC-tree proposed in this paper is a high-dimensional data indexing structure that sorts entries in nodes, maintains a pointer to its parent for each node, and supports indexing and querying data from metric space. The entries in the SC-tree are sorted by the distance between routing objects and their parents. The pointer from current entry to its parent is called a "clue". Details about the indexing structure, indexing algorithm, and query algorithm of the SC-tree are described in the following subsections.

3.1 Indexing Structure

The indexing structure of the SC-Tree includes two parts: nodes and entries. The entries in a node are sorted according to distances between routing objects and their parent objects, represented as *distFromParent*. There are a fixed number of entries in an internal node or leaf node, which are inserted into the node in ascending order of *distFromParent*. More formally, the entry and node structure are defined as follows:

An entry has five attributes: the feature vector of an routing object, O_n, the pointer to the root of the sub-tree, *sub-tree*, the object identifier of the entry, *oid*, the covering radius of its sub-tree, *coverRadius,* and the distance between the routing object and its parent, *distFromParent*. If the entry is internal, set *oid*=1. If the entry is a leaf one, set *sub-tree=Nil* and *coverRadius=0.*

A node has five attributes: the number of total entries in the node, *totalEntries*, the entries in a node, $entries_i$ ($i < totalEntries$), the number of non-empty entries in the node, *currentEntries*, the pointer to the parent node, *parentNode*, and the index of the entry in *parentNode, entryIndex,* which points to the current node. To locate the parent entry of current object is to simply return *parentNode[entryIndex]*. The *parentNode[entryIndex].routObjectFeature* is the feature vector of the current object.

3.2 Indexing Algorithm

The indexing algorithm specifies how objects are inserted and how to deal with node overflow when a node has already been full before inserting a new object. In this section, an insert algorithm and a split algorithm are described in detail, with < change i > denoting the major differences between the SC-tree and the M-tree.

Insert Algorithm: Insert(treeNode, entry(O_n))
Input parameters: treeNode, entry(O_n)
Return: updated treeNode with the inserted entry(O_n)
S1. Get all the entries in treeNode.
S2. If treeNode is not a leaf node
  S2.1. Select those entries whose covering radiuses will not increase if entry(O_n) is inserted into them.

S2.2. If the selected entries are not empty, select an entry, denoted as chosenEntry, whose routing object O_r is the closest to the routing object O_n of entry(O_n).
S2.3. Else select the chosenEntry with the minimal distance ($d(O_r, O_n)$ – coverRadius).
S2.4. Get the sub-tree of chosenEntry, recursively call Insert(sub-tree, entry(O_n)).
S3. Else the treeNode is a leaf node.
S3.1. If the treeNode is not full, insert entry(O_n) in the treeNode in ascending order of distFromParent and increase currentEntries by 1. <change 1>
S3.2. Else Split (treeNode, entry(O_n)).
S4. Return the updated treeNode.

Split Algorithm: Split(treeNode, entry(O_n))
Input parameters: treeNode, entry(O_n)
Return: splitTreeNode
S1. Set combinedEntries = {entries of treeNode ∪ entry(O_n)} and sort combinedEntries by distFromParent.
S2. Get the parent node of treeNode by its parentNode pointer. <change 2>
S3. If treeNode is the root node (its parentNode is empty)
S3.1. Set entry1 = the first entry of treeNode. <change 3>
S3.2. Set entry2 = the last entry of treeNode. <change 4>
S4. Else if treeNode is not the root node
S4.1. Set entry1 = parentNode[entryIndex]. <change 5>
S4.2. Set entry2 = the last entry of combinedEntries. <change 6>
S4.3. Set routObject1 = the routing object of entry1.
S4.4. Set routObject2 = the routing object of entry2.
S5. Divide combinedEntries into two tree nodes, treeNode1 and treeNode2, based on the distances from the objects in combinedEntries to routObject1 and routObject2.
S6. If treeNode is the root node
S6.1. Allocate a newRootNode.
S6.2 Store entry1 and entry2 in newRootNode.
S6.3 Record the parentNode and entryIndex for treeNode1 and treeNode2. <change 7>
S6.4. Set splitTreeNode = newRootNode.
S7. Else if treeNode is not the root node
S7.1. Replace parentNode[entryIndex] = entry1. <change 8>
S7.1. If parentNode is full
S7.1.1. Split(parentNode, entry2).
S7.2. Else if parentNode is not full
S7.2.1. Store entry2 in parentNode.
S8. Set splitTreeNode = parentNode.
S9. Return splitTreeNode.

In the construction of an M-tree, the split algorithm is frequently called in the insert algorithm, hence its efficiency will largely influence the efficiency of the insert algorithm. There are two disadvantages in the split algorithm of the M-tree. The first is that the distance between each object in an entry and its parent object has to be calculated in order to choose two routing objects from the split entry. The second is that the split algorithm has to travel from the root node to all its sub-trees until the

current node is reached in order to find the parent node of current node. To overcome these two disadvantages, there are several noticeable modifications in the SC-tree, compared with the M-tree. In step S3.1 of the insert algorithm, the entry is inserted into the tree node in ascending order of *distFromParent*. This modification makes it possible to implement steps S3.1, S3.2, and S4.2 in the split algorithm. Due to the modifications, the SC-tree simply selects the first object and the last object from the entry as the routing objects for the split sub-trees because they are sorted by the distance between any object and their parent object. Furthermore, the modifications also speed up step S5 of the split algorithm because it is almost done for distributing objects to the first tree node, *treeNode1*, in which the objects are already sorted. Another modification is in step S6.3 of the split algorithm, in which the parent node of the current node is recorded .This modification, which is based on the indexing structure of SC-Tree, makes steps S2, S4.1, and S7.1 in the split algorithm much more efficient. To get the parent object of current node, *parentNode* and *entryIndex* attributes of current node can be directly returned.

3.3 *k*-NN Query Algorithm

The *k*-NN query algorithm of the SC-tree implements its search logic, which is described as follows:

***k*-NN Query Algorithm: *k*-NN(startNode, query, *k*)**
Parameters: startNode, query , *k*
Return: an array *NN* storing *k*-NN query results
S1. If startNode is a rootNode
S1.1. Initialize an array *NN*.
S1.2. Choose active sub-trees based on their lower bounds, and insert them into the priority queue *PR*.

Note: Different from the M-tree, the SC-tree inserts active sub-trees in ascending order of their lower bounds, which are defined by

$$d_{min} = \max\{\text{distFromRoutObjectToQuery} - \text{coverRadius}\} \quad (4)$$

where *distFromRoutObjectToQuery* represents the distance between the routing object and the query object. <change 9>

S1.3. If the *PR* is not empty, select the first sub-tree, denoted as chosenNode, for which the d_{min} is minimal. Set $d_k = d_{min}$ when $d_{min} < d_k$. <change 10>
S1.3.1. Select the parentNode of chosenNode by its pointer to parent. <change 11>
S1.3.2. Calculate the distance *distFromParentToQuery* between the routing object of parentNode[entryIndex] and the query object.
S1.3.3. If the entry in chosenNode satisfies the following condition:

$$|\text{distFromParentToQuery} - \text{distFromParent}| \leq (d_k + \text{coverRadius}) \quad (5)$$

where *distFromParentToQuery* is the distance between the parent object to the query object, and *distFromParent* is the distance between the routing object and the parent object.
S1.3.3.1. Calculate *distFromRoutObjectToQuery*.
S1.3.3.2. If *distFromRoutObjectToQuery* < d_k

S1.3.3.2.1. Perform an ordered insertion of *distFromRoutObjectToQuery* into *NN*, and get back the new *k*-NN distance d_k.

S1.3.3.2.2. Remove entries in the *PR* if their lower bounds d_{min} exceed d_k.

Firstly, the position in the *PR* where the first sub-tree with its lower bound exceeds d_k is found by binary search. Secondly, all the entries after that position in *PR* will be removed. <change 12>

S1.4. Else the *PR* is empty

S1.4.1 Return the array *NN*.

S2. If startNode is not a rootNode

Return null (empty tree node).

In M-tree active sub-trees and their lower bounds are not sorted in the priority queue *PR*, which means that the sub-tree with the minimal lower bound has to be searched before it can be accessed by the query object. Another disadvantage of M-tree is that its algorithm has to travel from the root node to all sub-trees in order to locate the parent node of the current node. In order to avoid the first disadvantage, in step S1.2 of the *k*-NN query algorithm of the SC-tree, the sub-trees are sorted in ascending order of their lower bounds in the *PR*. As a result, in step S1.3, the first sub-tree stored in the *PR* is the one with the minimal lower bound, i.e., the nearest one to the query object. Furthermore, it also speeds up step S1.3.3.2.2 as a result of the sorted lower bounds of sub-trees. To avoid the second disadvantage, the SC-tree simply returns the attributes *parentNode* and *entryIndex* of current node.

In order to test the indexing and query efficiency of the SC-tree, experiments are carried out and analyzed in the next section.

4 Experimental Results and Analyses

Experiments have been carried out on five high-dimensional datasets: Census, Corel, FaceR, Forest, and Synthetic, in order to evaluate the performance of the proposed method. The Census data set contains 22,784 139-dimensional feature vectors, which is a highly clustered dataset with about 80% vectors clustered in 20% regions. The Corel dataset contains 68,040 32-dimensional image feature vectors. The FaceR dataset contains 2000 99-dimensional feature vectors extracted from face images. The Forest dataset contains 41012 54-dimensional feature vectors among which 44 are Boolean attributes and 10 are real-valued attributes. Finally, the Synthetic data set, generated by Aggarwal [1], contains 12,040 40-dimensional feature vectors, which is a very sparse data set. In our experiments, the proposed SC-tree and the M-tree are tested on the five data sets. All objects in the data set are inserted into the M-tree and the SC-tree one by one in the indexing stage. Every object acts as a query object during the *k*-NN query stage. The number of children nodes for each node (fan-out), denoted as *CN*, is chosen from 10, 20, 30, 40, 50, and 60. The indexing efficiency is measured by indexing time, while the *k*-NN query efficiency is measured by the query time spent on all the query objects in a data set. The query accuracy is measured by a query accuracy ratio, which is

defined as the ratio of the number of correctly returned query results to the total number of query results.

Firstly, the indexing performance of the M-tree and the SC-tree are compared in Fig. 1~5. It is clear that the indexing time of the SC-tree is much shorter than that of the M-tree. In fact, the SC-tree is about 20% quicker than the M-tree on five data sets. It is probably because in the M-tree objects were inserted into nodes randomly, but in the SC-tree objects were inserted into nodes in order. As a result, in the split operation of the SC-Tree, the first and the last entries can be easily used for choosing routing objects. However, the split operation of the M-tree has to calculate the distance between every pair of entries in a node to select two entries with the farthest distance. Both the SC-tree and the M-tree construct indexing structures by inserting objects in the data set one by one. It is reasonable to analyze the efficiency of inserting and splitting operations to reflect the indexing efficiency. Let the number of entries in a node be np . For the SC-tree, the time to insert one object into a node is $O(\log(np))$, and the time to select two new routing objects from the split node in splitting operation equals to $O(1)$ by selecting the first object and the last object as the new routing objects, thus the total indexing time complexity of SC-tree can be approximated as $[O(\log(np)) + O(1)] \approx O(\log(np))$. For the M-tree, the time to insert one object into a node is $O(1)$ by adding the object to the end of the node directly, the time to select routing objects from the split node in splitting operation is $C_n^2 = np(np-1)/2$ by comparing every pair of objects in the split node, and thus the total indexing time complexity of M-tree can be approximated as $[O(np(np-1)/2) + O(1)] \approx O(np^2)$. From the analysis of the indexing time complexity, the indexing time of the SC-tree is much shorter than that of the M-tree.

Secondly, the query time of the two methods are compared in Fig. 6~10. It can be seen that the *k*-NN query time of the SC-tree is shorter than that of the M-tree. In the *k*-NN search algorithm, the chosen node in the M-tree is added into the priority queue *PR* without sorting the positions of sub-trees according to the lower bounds between the query object and sub-trees. As a result, it influences the order of accessing nodes. While the SC-tree sorts the sub-trees in ascending order of the lower bounds between the query object and sub-trees, thus the nearest sub-tree to the query can be accessed firstly. Consequently, the SC-tree reserves better candidate objects and prunes irrelative sub-tree at an earlier stage, which greatly reduces distance calculations. Another important difference between the SC-tree and the M-tree is the pointer to the parent of the current object. If there are *pp* objects in the indexing tree, the level of the tree is $O(\log pp)$. For instance, if a tree has two entries in each node and contains 8 objects, then the level of the tree equals $\log_2(8) = 3$. The M-tree has to locate its parent node by travelling the indexing tree, which starts from the root node until the node itself is reached. The time complexity of travelling in the M-tree is $O(\log pp)$, whilst it is $O(1)$ in the SC-tree directly using pointer *parentNode[entryIndex]*. Because locating a parent node is a very frequent operation in both indexing and query, this complexity has a great impact on the indexing and query efficiency.

Finally, the query accuracy ratios for the M-tree and the SC-tree are very similar, as shown in Fig. 11 and Fig. 12 respectively. The query accuracy ratios of both the

SC-tree and the M-tree are between 92% and 99% when *NC*=10~60, which are quite stable.

From the above experimental results and analyses, it can be concluded that, without sacrificing the query accuracy, the SC-tree largely improves the indexing and query efficiency in comparison with the M-tree.

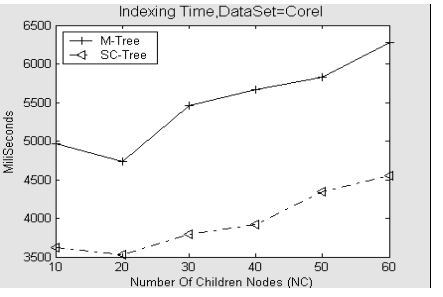

Fig. 1. Indexing time on Corel

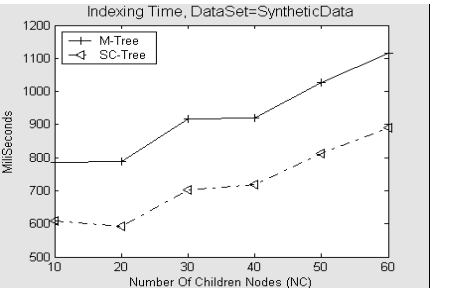

Fig. 2. Indexing time on Synthetic Data

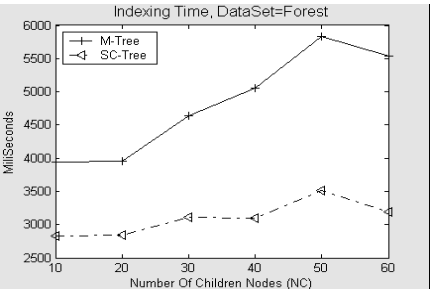

Fig. 3. Indexing time on Forest

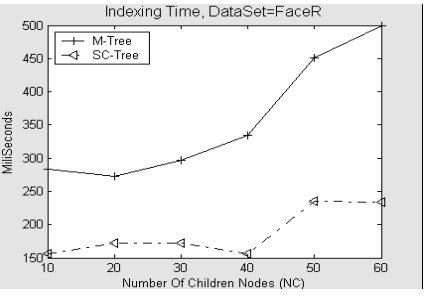

Fig. 4. Indexing time on FaceR

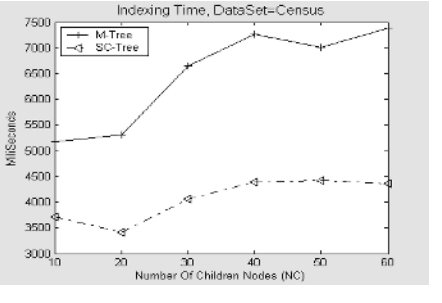

Fig. 5. Indexing time on Census

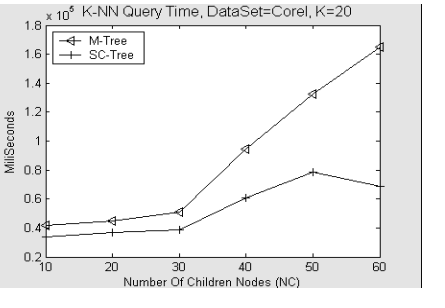

Fig. 6. *k*-NN query time on Core

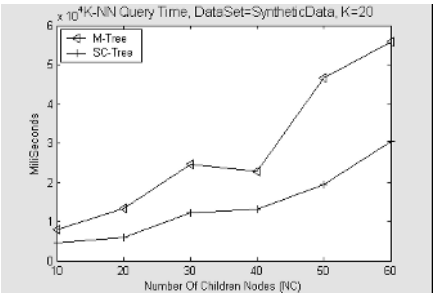

Fig. 7. *k*-NN query time on Synthetic Data

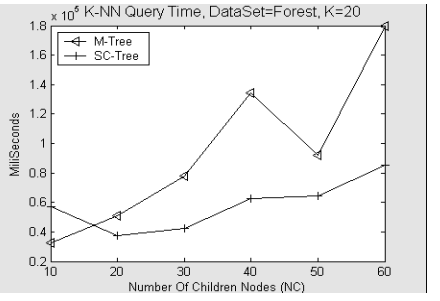

Fig. 8. *k*-NN query time on Forest

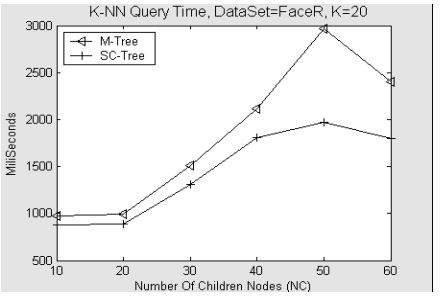

Fig. 9. *k*-NN query time on FaceR

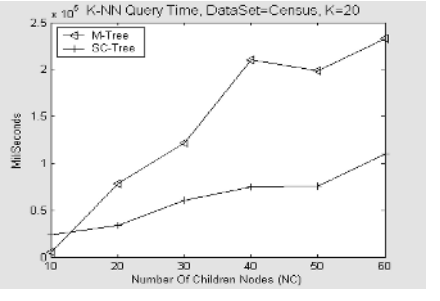

Fig. 10. *k*-NN query time on Census

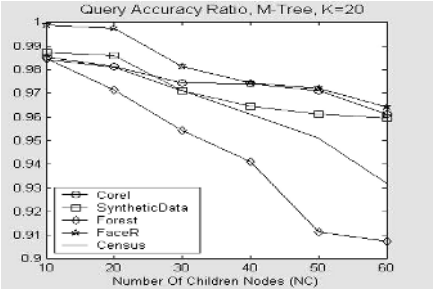

Fig. 11. *k*-NN query accuracy ratios on 5 data sets by M-tree

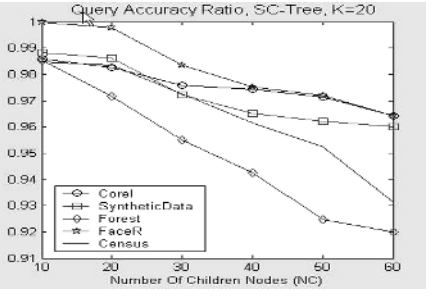

Fig. 12. *k*-NN query accuracy ratios on 5 data sets by SC-tree

5 Conclusion

M-tree is an efficient dynamic indexing structure which indexes and queries data objects from a generic metric space and utilizes the triangle inequality postulate to

prune irrelative sub-trees during the query stage. This paper proposes an SC-tree indexing structure which inherits the advantages of the M-tree and overcomes its disadvantages. Experimental results and complexity analyses show that the SC-tree is much more efficient than the M-tree with respect to the query time and indexing time without sacrificing its query accuracy.

References

1. Aggarwal, C. C., Procopiuc, C., Wolf, J.L., Yu, P. S., Park, J. S.: Fast algorithms for projected clustering. Proc. of the ACM SIGMOD Conference, Philadelphia, USA (1999) 61-72
2. Bartolini, I., Ciaccia, P., Patella, M.: String matching with metric trees using an approximate distance. Proc. of the 9th Int. Symposium on String Processing and Information Retrieval (SPIRE), Lisbon, Portugal (2002) 271-283
3. Beckmann, N., Kriegel, H. P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. Proc. of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ (1990) 322-331
4. Berchtold, S. Keim, D. A., Kriegel, H. P.: The X-tree: An index structure for high-dimensional data. Proc. 22nd Int. Conference on Very Large DataBases (VLDB), Bombay, India (1996) 28-39
5. Bozkaya, T., Ozsoyoglu, M.: Distance-based indexing for high-dimensional metric spaces. Proc. of ACM SIGMOD, Tucson, USA (1997) 357-368
6. Brin, S.: Near neighbor search in large metric spaces. Proc. 21nd Int. Conference on Very Large DataBases (VLDB), San Francisco, USA (1995) 574-584
7. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. Proc. Int. Conference of VLDB, Athens, Greece (1997) 522-525
8. Ciaccia, P., Patella, M., Zezula, P.: Processing complex similarity queries with distance-based access methods. Proc. of the 6th EDBT, Spain (1998) 9-13
9. Ciaccia P., Patella, M.: PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. Proc. of the 16th Int. Conference on Data Engineering (ICDE), California, USA (2000) 244-255
10. Ciaccia , P., Patella, M.: Searching in metric spaces with user-defined and approximate distances. ACM Transactions on Database Systems, Vol. 27, (2002) 398- 437
11. Ciaccia,P., Nanni, A., Patella, M.: A query-sensitive cost model for similarity queries with M-tree. Proc. of the 10th Australasian Database Conference (ADC), New Zealand, (1999) 65-76
12. Finkel, R., Bentley, J. : Quad-trees: A data structure for retrieval on composite keys. ACTA Informatica, Vol. 4, (1974) 1-9
13. Guttman, A.: R-trees: A dynamic index structure for spatial searching. Proc. of ACM SIGMOD, Boston, USA (1984) 47-57
14. Heisterkamp, D. R., Peng, J.: A kernel vector approximation file for nearest neighbor search using kernel methods. Proc. of the 6th Kernel Machines Workshop at Neural Information Processing Systems Conference, Whistler, Canada (2002) 1-12
15. McNames, J.: A fast nearest neighbor algorithm based on a principal axis search tree. IEEE Transactions on Pattern Analysis and Intelligence, Vol. 23, (2001) 964-976
16. Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: An adaptable, symmetric multikey file structure. ACM Trans. on Database Systems, Vol. 9, (1984) 38-71

17. Robinson, J.: The KDB-tree: A search structure for large multidimensional dynamic indexes. Proc. of the ACM SIGMOD Int. Conference on Management of Data, Ann Arbor, Michigan (1981) 10-18
18. Uhlmann, J. K.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters, Vol. 40, (1991) 175-179
19. Zezula,P., Savino, P., Amato, G., Rabitti, F.: Approximate similarity retrieval with M-trees. VLDB Journal, Vol. 7, (1998) 275-293