International Journal of Computer Science & Applications
Vol. III, No. I, pp. 1 - 25

The SH-Tree: A Novel and Flexible Super Hybrid Index Structure for Similarity Search on Multidimensional Data†

DANG Tran Khanh
Faculty of Information Technology
HCMC University of Technology
National University of Ho Chi Minh City, Vietnam
dtkhanh@hcmut.edu.vn

Abstract

Approaches to indexing and searching feature vectors are an indispensable factor to support similarity search effectively and efficiently. Such feature vectors extracted from real world objects are usually presented in the form of multidimensional data. As a result, many multidimensional data index techniques have been widely introduced to the research community. These index techniques are categorized into two main classes: SP (space partitioning)/KD-tree-based and DP (data partitioning)/R-tree-based. Although there are a variety of "mixed" index techniques, which try to inherit positive aspects from more than one index technique, the number of techniques that are derived from these two main classes is just a few. In this paper, we introduce such a "mixed" index, the SH-tree: a novel and flexible *super hybrid* index structure for multidimensional data. Theoretical analyses indicate that the SH-tree is a good combination of the two index technique families with respect to both the presentation and search algorithms. It overcomes shortcomings and makes use of their positive aspects to facilitate efficient similarity searches in multidimensional data spaces. Empirical experiment results with both uniformly distributed and real data sets will confirm our theoretical analyses.

Keywords: similarity search, multidimensional access method, bounding sphere, minimum bounding rectangle, SH-tree.

1 Introduction

The de facto standard technique widely applied to dealing with the searching problem in many modern database applications is the so-called feature transformation: Important properties of (complex) objects as images, audio/video, etc. are extracted and mapped into points of a multidimensional vector space. These multidimensional vectors are called feature vectors. Feature based similarity search has a long development process which is still in progress now. Its application range includes multimedia databases [43], time-series databases [17], CAD/CAM systems [6], medical image databases [35], flexible query answering systems (FQASs) [36], etc. In these large databases, feature spaces are usually indexed by means of multidimensional access methods (MAMs) in order to speed up the search process and minimize related costs, in which similarity queries are naturally become neighborhood queries in the feature space.

† This work had been partly done as the author was at the School of Computing Science, Middlesex University, London, UK and FAW-Institute, Johannes Kepler University of Linz, Austria

Since Morton introduced the space-filling curves in 1966, many MAMs have been developed. A survey schema summarizes the history of MAMs from 1966 to 1996 is presented in [22]. In [20], we also introduced an evolution schema for most prominent MAMs published recently as well as many discussions of great worth. These surveys together show that multidimensional index techniques can be divided into two main classes: (1) index structures based on space partitioning or KD-tree [7] (SP- or KD-tree-based) as KDB-tree [41], hB-tree [38], LSD-tree and LSDh-tree [32, 27], GNAT tree [14], mvp-tree [13], etc., and (2) index structures based on data partitioning or R-tree [24] (DP- or R-tree-based) consist of R-tree and its variants [44, 11], X-tree [10], SS-tree [45], SR-tree [33], M-tree [19], etc. The remains, which can not be categorized into the above schema, may be hybrid techniques of both index technique classes just mentioned above or may be special index techniques, which are called dimensionality reduction index techniques like Pyramid technique [5, 37], UB-tree [3], or space-filling curves based index techniques (see [22] for a survey). Recently, the Hybrid tree [16][1], a hybrid technique has been proposed. It is formed by combining some positive aspects of both SP- and DP-based techniques. Although this hybrid technique has been proven to be efficient in managing and querying multidimensional data sets, it still has deficiencies, which is one of reasons that lead us to introduce a new efficient and flexible index structure. Section 2 below discusses these impulsive reasons.

The rest of this paper is organized as follows: Section 2 discusses motivations for introducing the SH-tree. In section 3, we provide insights into the SH-tree structure and introduce a new concept of balance to SH-trees. Section 4 discusses basic operations of the SH-tree. Issues related to implementation details and performance evaluation results are shown in section 5. Next, in section 6, we present issues relevant to the integration process of the SH-tree into standard DBMSs and advanced query types. Finally, section 7 gives conclusions and reveals open research directions.

2 Motivations

The SR-tree has shown superiorities over the R*-tree and SS-tree by dividing feature space into both small volume regions (using bounding rectangles–BRs) and short diameter regions (using bounding spheres–BSs). However, the SR-tree must incur the fan-out problem: only one third of the SS-tree and two thirds of the R*-tree [33]. The low fan-out causes the SR-tree based searches to read more nodes (i.e. IO-cost increases) and to reduce the query performance. This problem does not occur in the KD-tree based index techniques: the fan-out is constant for arbitrary dimension number.

As mentioned above, the Hybrid tree makes use of positive characteristics of both SP- and DP-based index techniques. It depends on the KD-tree based layout for internal nodes and employs bounding regions (BRs) as hints to prune while traversing the tree. To overcome the access problem of unnecessary data pages, the Hybrid tree also applies a dead space eliminating technique by coding actual data regions (CADR) [27]. Although the CADR technique partly softens the unnecessary disk access problem, it is still not a high efficient solution to solve the problem entirely. It strongly depends on the number of bits used to code the actual data region and, in some cases, this technique does not benefit regardless of how many bits are used to code the space. Figures 1a, 1b show such examples in a 2-dimensional example space. Therein, the whole region is coded irrespective of how many bits are used. Figure 1c shows another example where the benefit from coding the actual data region is not much interesting, especially for range and nearest neighbor (NN) queries. This is due to the high remaining dead space ratio in the coded data region. Besides, when new objects locate outside the bounds of feature space already indexed by the Hybrid tree, the encoded live space (ELS) [16] must be recomputed from scratch.

[1] Internal nodes presentation is very similar to that of the Spatial KD-tree [39]

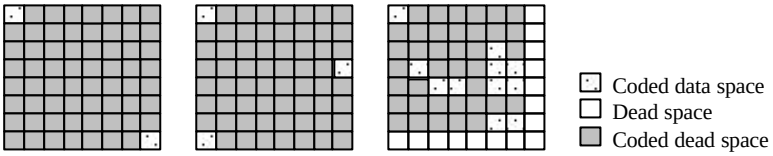


Figure 1: Some problems with coding actual data region

Furthermore, the SP/KD-tree based index techniques *in common* recursively partition space into two subspaces using a single dimension until the data object number in each subspace can be stored in a single data page as the LSDh-tree, the Hybrid tree, etc. This partitioning way may lead cluster of data to be quickly destroyed because the objects stored in the same page may be "far away" from each other in the real space. This problem could significantly influence the search performance, especially, it could cause the number of disk accesses per range queries to be increased [22]. It is contrary to the DP/R-tree based index techniques as the SS-tree, the SR-tree, etc., in which they try to keep near objects in the feature space into each data page.

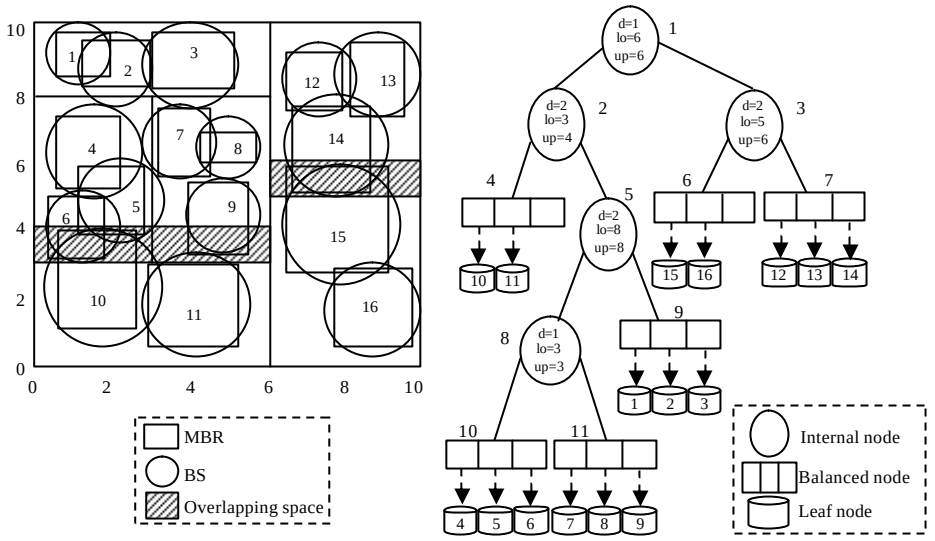
To alleviate these problems and take inherent advantages of the SR-tree (the R-tree based techniques as a whole), together with introducing novel noteworthy concepts we present the SH-tree in next sections. With the SH-tree, the fan-out problem will be overcome by employing the KD-tree-like presentation for partitions of its internal nodes. The data cluster problem as mentioned above, however, is softened by still keeping the SR-tree-like structure for presentation of balanced nodes of the SH-tree. In addition, to manage both spatial points and extended spatial objects (lines, polygons, etc.) the SH-tree also allows overlaps between partitions.

3 The SH-tree Structure

3.1 Multidimensional Space Partitioning and the SH-tree Structure

The SH-tree is planned to apply to both point and extended data objects so we choose a no overlap-free space partitioning strategy for directory nodes. The idea of this approach is to easily control objects that cross a selected split position and to solve the storage utilization problem. The former is described in the SKD-tree [39] and the latter has happened to the KDB-tree, which shows uninterestingly slow performance even in 4-dimensional feature vector spaces [23].

There are three node kinds in the SH-tree: Internal, balanced and leaf nodes. Each internal node i has the structure like $<d, lo, up, other\_info>$, where d is the split dimension, *lo* represents the left (lower) boundary of the right (higher) partition, *up* represents the right (higher) boundary of the left (lower) partition, and *other_info* consists of additional information as node type information, pointers to its left, right child and its parent node as well as *meta-data* like the data object number of its left, right child. While $up \leq lo$ means no overlap between partitions of its two child nodes, $up > lo$ indicates that the partitions are overlapping. This structure is similar to ones introduced in the SKD-tree and Hybrid tree. The supplemental information like the meta-data as mentioned above also gives hints to develop a cost model for NN queries in high-dimensional space or to estimate the selectivity of range queries, etc. Moreover, let BR_i denote bounding rectangle of internal node i. The bounding rectangle of its left child is defined as $BR_i \cap (d \leq up)$. Note that $\cap$ denotes geometric intersection. Similarly, the bounding rectangle of its right child is defined as $BR_i \cap (d \geq lo)$. This allows us to apply algorithms used in the DP/R-tree based index techniques to the SH-tree.



(a) (b)

Figure 2: A possible partition of an example data space and the corresponding mapping to the SH-tree

Balanced nodes are just above leaf nodes and they are not hierarchical (see Figure 2). Each of them has a similar structure to that of an internal node of the SR-tree. This is a specific characteristic of the SH-trees. It conserves the data cluster, in part, and makes the height of the SH-tree smaller as well as employing the SR-tree's superior aspects during the querying phase. Moreover, it also shows that the SH-trees are not simple in binary shape as in typical KD-tree based index techniques. They are also multi-way trees as R-tree based index techniques:

BN: <B_1, B_2, …B_n> (minBN_E ≤ n ≤ maxBN_E)

B_i: <BS, MBR, num, child_pointer>

In general, a balanced node consists of n entries B_1, B_2, …, B_n with $minBN\_E \le n \le maxBN\_E$, where *minBN_E* and *maxBN_E* are the minimum and maximum number of entries in the node, respectively (cf. section 5.1). Each entry B_i keeps information of a leaf node including four components: a bounding sphere *BS*, a minimum bounding rectangle *MBR*, the object number of leaf node *num* and a pointer to it *pointer*.

One question is why the SH-tree does not employ minimum bounding spheres (MBSs) instead of BSs. As stated in [33], the SR-tree also uses MBRs and BSs to express subspaces due to the computational complexity of MBSs. Obviously, the query performance will be improved if MBRs or MBSs (or both of them) are used instead of BRs or BSs. For the SH-tree, we can use both MBRs and MBSs to express boundaries of data pages. The MBR and MBS of a data page will be used to make the final decision if this data page will be accessed during the tree traversal. This is similar to that of the SR-tree but only applied to balanced nodes of the SH-tree. Computing MBRs has been done and well-known with the R-tree family. The problem here is how to compute MBS of a given object set? This is not feasible in a high-dimensional space because the time complexity is exponential in dimension number [34]. An approximate computing algorithm for MBS of a given object set is also presented in [34]. It can be applied to the SH-tree but due to the computational complexity, which negatively affects the CPU-cost when building and updating the tree, the SH-tree uses MBRs and only BSs.

Furthermore, each leaf node of the SH-tree has the same structure as that of the SS-tree (because the SR-tree is just designed for point objects but the SH-tree is planned for both points and extended objects):

LN: <L_1, L_2, …L_m> (minO_E ≤ m ≤ maxO_E)

L_i: <obj, info>

As we see above, each leaf node of the SH-tree consists of m entries L_1, L_2, … L_m, with $minO\_E \le m \le maxO\_E$, where $minO\_E$ and $maxO\_E$ are the minimum and maximum number of entries in a leaf, respectively. Each entry L_i consists of a data object *obj* and information in the structure *info* as coordinate values of the data object's feature vector, a radius that bounds the data object's extent in the feature space, the data object's MBR, etc. If data objects in the database are complex, *obj* is only an identifier instead of a real data object. In addition, if the SH-tree is only applied to point data objects, each L_i is similar to that of the SR-tree: L_i: <*obj*, *feature_info*>. In this case, the other information of the objects is no longer needed. For instance, the parameter *radius* is always equal to zero and MBR is the point itself.

Figure 2 shows a possible partition of an example feature space and its corresponding mapping to the SH-tree. Assume we have a 2-dimensional feature space D with a size of (0, 0, 10, 10). With the split information (d, lo, up) = (1, 6, 6), the BRs of left and right children of the internal node 1 are $BR_2 = D \cap (d \le 6) = (0, 0, 6, 10)$ and $BR_3 = D \cap (d \ge 6) = (6, 0, 10, 10)$, individually. For the internal node 2, assume (d, lo, up) = (2, 3, 4), we have $BR_4 = BR_2 \cap (d \le 4) = (0, 0, 6, 4)$, $BR_5 = BR_2 \cap (d \ge 3) = (0, 3, 6, 10)$. Similarly, for the internal node 3, with (d, lo, up) = (2, 5, 6) we obtain $BR_6 = BR_3 \cap (d \le 6) = (6, 0, 10, 6)$, $BR_7 = BR_3 \cap (d \ge 5) = (6, 5, 10, 10)$, etc. Note that the BRs coordinate information is not stored in the SH-tree explicitly, but it is dynamically computed as needed.

Moreover, the storage utilization constraints of the SH-tree must ensure each balanced node is filled with at least *minBN_E* entries and each data page contains at least *minO_E* data objects. Hence, each subspace according to a balanced node holds *DON* data objects (DON: data object number) and *DON* satisfies the following condition:

$$minO\_E \times minBN\_E \le DON \le maxO\_E \times maxBN\_E \quad (\mathbf{1})$$

3.2 Splitting Nodes in the SH-tree

Inputs for creating the SH-tree are classified into two types: dynamic (in dynamic databases) and static (given data sets). Dynamic creation of the tree means the SH-tree is incrementally built up and in the course of this building data objects can be added or deleted. Conversely, as the SH-tree is built on some given static data set, operations related to both the tree and data set as insertion, deletion, etc. are not allowed during the building process. The problem of building MAMs over such static inputs relates to the bulk-loading problem (see [20]). If the SH-tree is incrementally created from a dynamic database, we consider only the split problem of leaf and balanced nodes. This is contrary to the creation of the SH-tree from a given data set in which we just take care of the internal node splitting. Below, we introduce splitting algorithms applied to all node types in the SH-tree. They can be suitably employed in different contexts.

3.2.1 Leaf Node Splitting

The boundary of a leaf node in the SH-tree is the geometric intersection between its MBR and BS (this information is maintained at its parent node, a balanced node), but BS is isotropic thus it is not suitable for choosing the split dimension. Choosing the split dimension therefore depends on its MBR. This problem in the SH-tree is solved in the same way as that of the Hybrid tree including overlap free splitting, i.e. there is no overlap between two leaf nodes after the split. The selected split dimension must minimize the expected disk access number per query. Without loss of generality, assume that the space is d-dimensional and the extent of MBR along the i^{th} dimension is e_i, $i = \overline{1, d}$. Let a range query Q be a bounding box with each dimension of length r. Prove as done in [16], we get the following result: the split dimension is k if $r/(e_k+r)$ is the minimum. Therefore, the split dimension k is chosen such that its extent in MBR is the maximum, i.e., $e_k = \max(e_i)$, $\forall i = \overline{1, d}$.

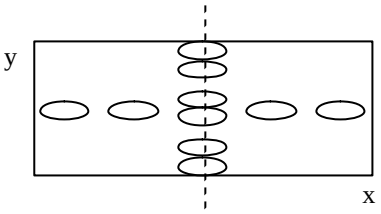


Figure 3: Problem with leaf node splitting in the Hybrid tree: No suitable split position satisfies the storage utilization constraint

The next step is to select the split position. First, we check if it is possible to split in the middle of the selected split dimension without violating the utilization constraint. If it is impossible, we distribute data items equally into two nodes after sorting data items according to their feature values in the selected split dimension[2]. This way also solves a special case in the hB-tree [38]. Figure 3 shows this case as an illustrative example in a 2-dimensional space: Assume the split dimension *x* is chosen and the minimum data object number of each partition is three (the storage utilization constraint). There is no suitable split position if we apply the way of the Hybrid tree as in [16]. In this case and other similar cases, the SH-tree will distribute data items equally into two nodes.

In Figure 4 below we succinctly summarize important steps to split an overfull leaf node in the SH-tree:

LeafNodeSplit(newObject)
Step 1: Create a new leaf node ***newLeaf*** and update necessary information, e.g. pointer to the parent node.
Step 2: Choose split dimension ***dim*** so that its extent in the MBR of the split node ***oldLeaf*** and the inserted object ***newObject*** is maximum. Note that the MBR computed here also includes ***newObject***.
Step 3: Choose the split position ***splitPos*** as the middle of the MBR with respect to the dimension ***dim***.
Step 4: Sort all data objects (in ***oldLeaf*** and also include ***newObject*** that causes this leaf to be overfull) in ascending order of their feature values but only in the dimension ***dim***.
Step 5: Distribute all data objects in ***oldLeaf*** and ***newObject*** as well into two parts: The first part includes all objects that their feature values in the dimension ***dim*** are less than ***splitPos***, and the second part consists of the remains.
Step 6: Check whether or not these two parts violate the storage utilization constraint. If not, put all objects in the second part into ***newLeaf*** and then update involved information such as MBRs, BSs, etc. for both ***oldLeaf*** and ***newLeaf***, and finish this algorithm here. Otherwise:
Step 7: Put a half of objects with the larger feature values in the dimension ***dim*** into ***newLeaf*** and carry out updates as the above case and then terminate the algorithm.

Figure 4: Steps of slitting a leaf node in the SH-tree

3.2.2 Balanced Node Splitting

Because the balanced node has a similar structure to internal nodes of the SR-tree and R*-tree, the internal node splitting algorithm of the R*-tree [11] can be applied to splitting overfull balanced nodes of the SH-tree. With the SH-tree, however, if the sibling of an overfull balanced node is also a balanced node and still not full, an entry of the overfull balanced node can be shifted to the sibling to avoid a split. This method also increases the storage utilization [26, 27]. Note that, the best entry of the overfull

[2] The split of a leaf node is an overlap-free split but this is just totally true for point data. For extended spatial data objects, it means that no data object at the same time appears in more than one leaf of the SH-tree

balanced node to be shifted is one that the sibling needs least enlargement to enclose. Thus, the modified splitting algorithm for overfull balanced nodes can be concisely described as follows: First, try to avoid a node splitting as just discussed. If it fails, the split algorithm similar to that of the R*-tree is employed (see [11] for details of the split algorithm in the R*-tree). Moreover, notice that, in the SH-tree, the balanced node split does not cause propagated splits upwards or downwards, which is called cascading splits [38] and happened to the KDB-tree [41]. The storage utilization constraint of the SH-tree therefore is also not affected by the splits at all.

3.2.3 Internal Node Splitting

As mentioned above, this kind of split appears only in the case of creating the SH-tree from a given static data set. Here "static" does not mean the data set is never changed, but it only requires that during the time of dealing with this data set, no any other action is allowed to intervene in. Such given data sets may be external data sets or even only intermediate or temporary results at a certain time during executing a batch of sequential queries (see, e.g., [30]), etc. Building a tree from such a static data set is usually called bulk-loading of the tree. With the SH-tree, the first step in this case is to recursively partition the space into subspaces until the data object number *DON* in each subspace satisfies inequality as shown in equation 1. Each of such subspaces represents a balance node in the SH-tree. Next step is to suitably cluster data objects in each subspace in order to create leaf nodes and their involved information as MBRs, BSs, etc. for the corresponding balanced node. How to choose effective "pivots" to cluster data objects in the best way is still an open issue of great interest [18]. Here, we will analyze and propose an algorithm for partitioning the data space into the subspaces as mentioned above. Each intermediate subspace of this partitioning process is according to an internal node in the SH-tree. The whole data space is according to the root node of the tree. Our split algorithm depends on philosophy of the R*-tree's.

Assume that there are given D_n data objects need to be indexed. For such a given static data set, we want to create a SH-tree with the *full* storage utilization, i.e. with the minimum necessary data page number. Therefore, each data page, i.e. each leaf node, is assumed to contain a maximum number of entries *maxO_E*. The minimum data page number D_p is therefore calculated as follows:

$$D_p = \left\lceil \frac{D_n}{\max O\_E} \right\rceil \qquad (2)$$

In the SH-tree, we have designed so that each disk page keeps a balanced node. In the other words, the capacity of a disk page will determine the maximum entry number for each balanced node in the tree. Hence, the minimum number of balanced nodes is calculated as follows: $b\_min = \left\lceil \frac{D_p}{maxBN\_E} \right\rceil$. With these parameters, we have minimum optimal height of the SH-tree is $H = 1 + \lceil \log_2 b\_min \rceil$. If N_1, N_2 are the object numbers of two partitions after splitting, they can be calculated as shown in formula 3 below:

$$N_1 = D_n - N_2$$
$$N_2 = (b\_min - \left\lceil \frac{b\_min}{2} \right\rceil) \times maxBN\_E \times maxO\_E \qquad (3)$$

Note that, N_2 can also be computed using other slightly variant formulas as follows: $N_2 = \left\lceil \frac{b\_min}{2} \right\rceil \times maxBN\_E \times maxO\_E$ or $N_2 = \left\lfloor \frac{b\_min}{2} \right\rfloor \times maxBN\_E \times maxO\_E$. From formula 3 we propose an internal node splitting algorithm modified from that of the R*-tree: For each dimension, it divides objects into two groups so that the objects with near feature vector co-ordinates in this dimension are clustered into the same group and each group has N_1, N_2 objects, individually. After that, the algorithm calculates the

overlap of the two groups in this dimension. The chosen splitting dimension is one that has the minimum overlap. With this algorithm, the resulting SH-tree is also balanced. Note that if the object number D_n is not small we can cluster objects into neighborhood groups first and then use centre objects of the groups instead of objects themselves as input for the algorithm. For each of the groups, the above algorithm can be applied to.

3.3 The Extended Balanced SH-tree

For almost index techniques based on the KD-tree, the tree structure is not balanced (e.g., LSD/LSDh-tree, SKD-tree). It means that, in such index trees, some leaf nodes are farther away from the root than all others are. Experiments as presented in [14], however, have shown that a good balance is not crucial for the performance of a MAM. In this section, we introduce a new concept of the balance problem in the SH-tree: *extended balance*. The motivation is to retain acceptable performance of the tree and reduce maintenance cost for its exact balance in the presentation structure.

Suppose that *p*, *b*, *b_min*, and *b_max* denote leaf node number, balanced node number, minimum and maximum number of balanced nodes in the SH-tree, respectively. The following inequality holds:

$$\text{b\_min} = \left\lceil \frac{p}{\max BN\_E} \right\rceil \le \text{b} \le \left\lceil \frac{p}{\min BN\_E} \right\rceil = \text{b\_max} \qquad (\mathbf{4})$$

We desire that the SH-tree's height *h* satisfies the following inequality:

$$1 + \lceil \log_2 b\_\min \rceil \le \text{h} \le \lceil \log_2 \text{b\_max} \rceil + 1 \qquad (\mathbf{5})$$

Inequality 5 is used to evaluate whether the SH-tree is "balanced" or not. The meaning of the balance here is loose: It does not mean that the path length of every leaf node from the root is equal to each other. We call this extended balance in the SH-tree. If the height *hl* of each leaf node in the SH-tree satisfies inequality 5, i.e. $1+\lceil \log_2 \text{b\_min} \rceil \le \text{hl} \le \lceil \log_2 \text{b\_max} \rceil + 1$, the SH-tree is called an *extended balanced tree (EBT)*, and otherwise it is not a balanced tree. The extended balance conception generalizes the conventional balance conception: If inequality 5 becomes $1 + \lceil \log_2 \text{b\_min} \rceil = \text{h} = \lceil \log_2 \text{b\_max} \rceil + 1$, an EBT becomes a *conventional balanced tree (CBT)*.

Let's see an example: If *minBN_E*=2 and *maxBN_E*=3, the SH-tree as illustrated in Figure 2 is not a CBT or an EBT; and in general it is not a balanced tree. However, inequality 5 can also be extended and rewritten as follows:

$$1 + \lceil \log_2 \text{b\_min} \rceil - x \le \text{h} \le \lceil \log_2 \text{b\_max} \rceil + 1 + x \qquad (\mathbf{6})$$

or a more general form:

$$1 + \lceil \log_2 \text{b\_min} \rceil - x \le \text{h} \le \lceil \log_2 \text{b\_max} \rceil + 1 + y \qquad (\mathbf{7})$$

In inequalities 6 and 7, parameters *x* and *y* are tolerant "errors". These parameters give more flexibility to the SH-tree but they must be selected carefully to prevent from creating a too much unbalanced tree. The SH-tree does not satisfy inequality 5 but satisfies inequalities 6 or 7 is called a *loosely extended balanced tree (LEBT)*. For example, as with the SH-tree in Figure 2 and assume that *minBN_E*=2 and *maxBN_E*=3, inequality 5 becomes $4 \le h \le 4$ (here $\text{b\_min}=\lceil 16/3 \rceil=6$ and $\text{b\_max}=\lceil 16/2 \rceil=8$). If the SH-tree satisfies this condition, it is a CBT (also EBT). We can readjust this condition with x=1 and get a new condition concerning inequality 6: $3 \le h \le 5$. With respect to the new condition, the above SH-tree is considered as a LEBT. Selecting parameters *x* and *y* in equations 6 and 7 depends on many other attributes, say *p*, *minBN_E*, *maxBN_E*, etc. If these parameters are chosen suitably, the maintenance cost for the balance of the SH-tree is substantially decreased but it does not affect the querying performance.

In general, if the SH-tree fails to satisfy inequalities 6 or 7 (with certain values of x and y), it needs to be reformed. The reformation can reorganize the SH-tree entirely (also called dynamic bulk loading–see [20] for more details) or suitably change splitting algorithm as introduced in [25]. There, Henrich presented a hybrid split strategy for

KD-tree based access structures. It depends on weighted average of the split positions calculated using two split strategies, data dependence and distribution dependence. Note that the dynamic reformation operation usually incurs substantial costs including both I/O accesses and CPU time. An efficient algorithm for such a reformation for the whole SH-tree is still an open problem. In [20], we preliminarily introduced a *local* dynamic bulk-loading algorithm for the SH-tree, in which it just dynamically reforms a certain part of the tree, but not the whole SH-tree, as necessary.

4 The SH-tree's Basic Operations

In this section, typical operations for building and querying the SH-tree are presented. First, we introduce algorithms for insertion, deletion, then describe querying algorithms.

4.1 Insertion

Let NDO be a new data object to be inserted into the SH-tree. First, we must traverse the SH-tree from the root downwards to locate a leaf node *w*, which NDO will belong to. To complete this, we must differentiate the tree traversal between two node types: Internal and balanced nodes. With internal nodes, we must select one among two branches to continue going down the tree. A simple algorithm to accomplish this step is briefly introduced in Figure 5 below.

```
ChooseInternalNode(Object NDO)
Let NDO[D] be the feature value of the new data object NDO with
respect to the split dimension D of this internal node (see
section 3.1 for more information).
if (D.Up≤D.Lo)             // not overlap
   if (NDO[D]<D.Up) return Left;
   else
      if (NDO[D]>D.Lo)     return Right;
      else                 // D.Up≤NDO[D]≤D.Lo
         if ((D.Lo-NDO[D])=(NDO[D]-D.Up))
            return Right if the object number of Left is greater
            than that of Right, otherwise return Left;
         else
            return Right if ((D.Lo-NDO[D])<(NDO[D]-D.Up)),
            otherwise return Left;
else                       // overlap: D.Up>D.Lo
   if (NDO[D]<=D.Lo)       return Left;
   else
      if (NDO[D]>=D.Up)    return Right;
      else                 // D.Lo<NDO[D]<D.Up
         if ((NDO[D]-D.Lo)=(D.Up-NDO[D]))
            return Right if the object number of Left is greater
            than that of Right, otherwise return Left;
         else
            return Right if ((NDO[D]-D.Lo)>(D.Up-NDO[D])),
            otherwise return Left;
```

Figure 5: Choose an internal node to insert a new data object into

The algorithm as described in Figure 5 firstly checks if two partitions of the internal node are overlapping. If they are not overlapping, the algorithm will return a child node (Left or Right) based on the feature value of NDO in the split dimension D of this internal node. In this case, tie is broken based on the node's data object number. In the contrary case, meaning the Left and Right nodes are overlapping, the algorithm will also act similarly to the previous one: It returns a child node based on the feature value of NDO in the split dimension D of this internal node, and when this feature value falls

into the interval (D.Lo, D.Up), the algorithm will return a child node containing NDO farther inside it according to the involving split dimension D.

With a balanced node, the best candidate to hold NDO is a leaf node having the closest distance to NDO. To compute the distance *dist(NDO, Leaf)* from NDO to a leaf node, whose real covering region is the intersection between a MBR and a BS, we use a similar method to that of the SR-tree:

$$\begin{aligned} &dist(NDO, Leaf) = \max(dr, ds) \\ &dr = MINDIST(NDO, Leaf.MBR) \\ &ds = \max(0, \| NDO - Leaf.BS.Center \| - Leaf.BS.Radius) \end{aligned} \quad (8)$$

Here, the MINDIST distance metric is originally introduced in [40] and notation ||z|| denotes the norm (the distance in this case) of expression z. See appendix for an introduction to distance metrics MINDIST and MINMAXDIST.

After determining such a leaf, if it has an empty entry, NDO is inserted. Conversely, the leaf is *overfull*, we propose two strategies can be executed together in the SH-tree:

- **Reinsertion:** If REINSERTION flag of this leaf is FALSE, a part of its entries will be reinserted after setting the flag to TRUE. A similar strategy has also been employed by several prominent trees introduced recently as the SS-tree, SR-tree, Hybrid tree, etc. However, here we use a flag to control the reinsertion strategy, meaning the leaf can be performed this strategy again as the flag is changed from TRUE to FALSE. This is different from the ways the SS-tree and R*-tree control the reinsertion.
- **Redistribution:** One data object (one entry) of this overfull leaf can be redistributed to one of its certain siblings[3], which is still not full, to make space for NDO. This idea is the same as that of [26] but does not recursively go upwards like that: The siblings here are locally located in the balanced node. In fact, the predefined constant *l* of the algorithm proposed in [26] is similar to the current entry number (CEN) of the balanced node (minBN_E ≤ CEN ≤ maxBN_E). Particularly, the parameter CEN for the SH-tree's redistribution algorithm here is different from each balanced node, this is a difference from the one presented in [26].

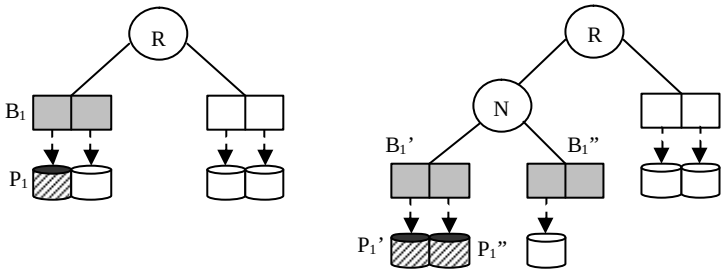


Figure 6: Split propagation in the SH-tree

In case a split is still compulsory, it can only propagate upwards at most one level. Figure 6 illustrates the split propagation in the SH-tree. Assume leaf P_1 is selected to insert a NDO and P_1's entry number is *maxO_E* already. Moreover, suppose the redistribution also failed and the REINSERTION flag of the node is TRUE. Consequently, P_1 is split into P_1' and P_1". Nevertheless, because *maxBN_E*=2 (*minBN_E*=1) in this example, the balanced node B_1 is later split into B_1' and B_1". At last, a new internal node N is created (Figure 6b sketches a possible resulting index structure after this split). The split process is stopped and has no more propagation to upper levels (root node R in this example). This characteristic of the SH-tree is far removed from the R-tree family, where a leaf split can propagate upwards to the root.

[3] In our implementation, it is a closest node to this overfull node among ones still have at least an empty entry

4.2 Deletion

After determining which leaf node contains the object and removing the object, the leaf may become *under-full* (i.e., the object number kept in this leaf is less than *minO_E*). There are some solutions to solve this problem [24]: An under-full leaf node can be merged with whichever sibling that needs to have the least enlargement or its objects can be scattered among sibling nodes. Both of them can cause the node splits, especially the latter can lead into a propagated splitting, say the balanced nodes splitting. The R-tree [24] employed re-insertion policy instead of the two above. The SR-tree, SS-tree, R*-tree and Hybrid-tree also employ this policy.

For a special structure of the SH-tree, we propose a new algorithm to solve the under-full leaf problem called *eliminate-pull-reinsert* algorithm. Our algorithm is quite similar to the eliminate-and-reinsert policy. However, because reinsertion can also cause the splits of leaf and balanced nodes, thus after deleting the object, if the leaf node is under-full, we apply a "pull" strategy to get one object from one among the siblings so that this sibling is still ensured to be satisfied the storage utilization constraints. This also depends on the idea in section 4.1 but in a contrary direction: While the under-full leaf "pulls" one object from some sibling, the overfull leaf "shifts" one object to the closest sibling that is still not full. Furthermore, in case the pull policy as discussed still does not solve the problem, the objects (entries) of the under-full leaf node will be reinserted. Note that, the pull policy can also propagate to only the siblings located in the same balanced node, which is the parent node of this leaf.

4.3 Search

Below, we describe algorithms to process exact match, range, NN and k-NN queries. With the SH-tree, internal nodes can be treated as ones of either the KD-tree or R-tree based MAMs (depending on query types), but balanced nodes and leaves are considered as ones of the SR-tree based MAMs. As the SH-tree is a distance based MAM, we first repeat aspects of a metric distance function D(x, y), where x, y are compared objects:

(1) $D(x, y) = D(y, x)$ (Symmetry)

(2) $D(x, y) \in [0, \infty)$ and $D(x, y) = 0$ iff $x = y$ (Non negativity)

(3) $D(x, y) \leq D(x, z) + D(z, y)$ (Triangle inequality)

The metric distance function D is used to measure the distance (similarity or dissimilarity) between spatial objects and feature vectors. The following search algorithms employ D as a generalized metric distance function. In practice, D depends on application domains. For example, D can be Euclidean metric (L_2), Manhattan metric (L_1), or any other Minkowski metric, etc.

Exact match queries: Given a spatial object O in d-dimensional space E, we must examine the database to check if O belongs to it. Answering this kind of queries is simple in the SH-tree. First, the SH-tree's traversal is carried out to locate balanced nodes that can contain the query object O. At these balanced nodes, O's feature values are evaluated to identify which leaf nodes are necessary to access next. These leaves' MBR and BS must overlap O. The return value is true if one of these leaves contains O or false, otherwise. Note that the query object O, if existing, is kept by only one leaf of the SH-tree. This is also different from some other MAMs such as the R^+-tree [44] and UB-tree [3] where an object can be split and kept by several leaf nodes.

Range queries: Given a query object Q in d-dimensional space E and a positive real value r, find all data objects O in the database satisfying condition: $D(Q, O) \leq r$ (bounding sphere range queries). Besides, another kind of range queries called bounding box range queries is defined as follows: Given a d-dimensional interval $[Q_{1min}, Q_{1max}]$, … $[Q_{dmin}, Q_{dmax}]$, find all data objects in the database that overlap this range. There have been many different algorithms proposed to deal with range queries, e.g. [3, 19] (see [20, 1] for more information). The SH-tree can flexibly and efficiently support different

kinds of range queries. An algorithm for the bounding sphere range queries is described at a high abstraction level below and its detailed version is presented in Figure 7:

[Bounding sphere range query algorithm]

(1) Traverse the SH-tree from root node to determine balanced nodes which the boundary of the range query is intersecting with.

(2) For each balanced node determined, the algorithm examines all of its entries. If both MBR and BS of a certain entry overlap the range query's boundary then the corresponding data page DP is loaded.

(3) With each loaded data page, the algorithm accesses all of its data objects. For each data object O in DP, a suitable distance function is employed to calculate the distance D(Q, O). If D(Q, O) ≤ r then O is reported as an element of the result set.

It is easy to modify this algorithm so that it can process the bounding box range queries. Concretely, in step 3 of the algorithm, it does not have to calculate the distance D(Q, O) but it must check whether or not the d-dimensional interval $[Q_{1min}, Q_{1max}]$, … $[Q_{dmin}, Q_{dmax}]$ overlaps the data object O. If it is, then O is put into the result set. Now, let's see the detail version of the above algorithm for range queries:

The detail version of the bounding sphere range query algorithm:

```
1:  BoundingSphereRangeQuery(bs,Rset,node,br)
2:  BOUNDINGSPHERE     bs          //bounding sphere range query
3:  RESULTBUFFER       Rset        //result set buffer
4:  NODEPOINTER        node        //pointer to a node of the tree
5:  BOUNDINGREC        br          //bounding rectangle of node
6:  Case (node.Type) of
7:  INTERNAL_NODE:
       /*Compute BRs for left and right, d:split dimension*/
8:     BOUNDINGREC     left_rec=br∩(d=node.up)
9:     BOUNDINGREC     right_rec=br∩(d≥node.lo)
       /*Compute distance from center of bs to left_rec and
       recursively call function if necessary*/
10:    If (MINDIST(bs.C,left_rec)=bs.R) then
11:      BoundingSphereRangeQuery(bs,Rset,node.Left,left_rec);
       /*Compute distance from center of bs to right_rec and
       recursively call function if necessary*/
12:    If (MINDIST(bs.C,right_rec)=bs.R) then
13:      BoundingSphereRangeQuery(bs,Rset,node.Right,right_rec);
14: BALANCED_NODE:
15:    For each entry Leaf in the balanced node do
         //compute distance between 2 centers
16:      FLOAT     dist = ObjectDistance(Leaf.Bs.C,bs.C);
         //if bs overlaps BS of Leaf
17:      If (dist=(bs.R+Leaf.Bs.R)) then
            //if bs also overlaps MBR of Leaf
18:         If (MINDIST(bs.C,Leaf.Mbr)=bs.R) then
19:            BoundingSphereRangeQuery(bs,Rset,Leaf,NULL);
20: LEAF_NODE:
21:    For each object Object in the leaf node do
         //compute distance between center of bs and Object
22:      FLOAT     dist = ObjectDistance(bs.C, Object);
         //if Object falls into the range query
23:      If (dist=bs.R) then
24:         Add Object to the result set Rset
25: EndCase
```

Figure 7: An algorithm for answering range queries in the SH-tree

This detail version is easy to understand and thus we do not go into the detailed explanations. We just want to emphasize that because both MBR and BS are used during the query processing to decide if a leaf needs to be further examined, accesses to

external memory for obtaining the real data objects are reduced substantially. This observation has been confirmed by empirical experiments in [33].

Nearest neighbor queries: Given a query object Q in d-dimensional space E. Find all data objects O in the database DB having a minimum distance from Q:

$$D(Q, O) \leq D(Q, O') \qquad \forall O' \in DB \in E \qquad (9)$$

There have been many algorithms proposed for answering NN queries, e.g. [40, 28, 29, 9] (see [20] for more details). However, algorithms introduced in [40, 28] are the state-of-the-art and both techniques proposed in these papers can be used for searching NN in the SH-tree with some minor modifications. The first change is as described in [33] when computing the distance from Q to each area represented by a leaf node in the SH-tree (corresponding to each region in the SR-tree). Here, because the boundary of a leaf in the SH-tree is the intersection of a MBR and a BS, the minimum distance from Q to a leaf area is defined as the longer distance between the minimum distances to the leaf node's MBR and BS from Q (see [33, 40] for the computational formulas). Second, if the SH-tree also employs MBRs for balanced nodes or uses the CADR technique[4] for any kind of nodes, they should be taken into account during the search process as well for further pruning unnecessary nodes to be accessed. This will improve the searching performance (meaning that it will reduce the searching costs as CPU-cost and IO-cost).

k nearest neighbor queries: This query type is a generalization of the NN queries: Given a query object Q in d-dimensional space E and a natural number k, find at least k objects closest to Q in the database. Here we say "at least" because it is possible that there are some k^{th} objects with the same distance to Q. With a few modifications as presented in [29] we can also employ their k-NN algorithms for the SH-tree. We shall elaborate on algorithms for k-NN queries and use them to evaluate the searching performance of the SH-tree in section 5.2.

5 Evaluating Performance of the SH-tree

In this section, we first present notable implementation details and two adaptations of the state-of-the-art algorithms for k-NN queries to the SH-tree, then give experimental results with both synthetic and real data sets to show the efficiency of the tree.

5.1 Implementation Details

There are three kinds of external data pages according to three node kinds of the SH-tree: Each leaf and balanced node is individually kept in a separate disk page; a group of internal nodes is kept in a disk page if the main memory has not enough room. This solution has been employed in the LSD/LSDh-tree [32, 27]. Theoretically, capacity of the disk pages can be different. Existing operating systems (e.g. Unix, Windows, etc.), however, do not support heterogeneous size paging in a single file, thus the implementation of the SH-tree using a single storage file should use an equal size paging as other MAM implementations have done. For convenience, we call a disk page which keeps a leaf or balanced node a leaf or balanced page, respectively. Similarity, we call a disk page that keeps a group of internal nodes an internal page.

Our current implementation of the SH-tree was carried out over Sun Solaris Unix operating system, so the page size for all kinds of nodes is set to 8KB to meet with the disk block size of the operating system. All programs are implemented in C++[5]. In addition, with a given page size, PAGE_SIZE, we can easily determine parameters as maxBN_E, maxO_E, and the maximum entry number of internal nodes can be stored in a disk page. The general formula to calculate these numbers is as follows:

[4] In our current implementation of the SH-tree, we do not employ MBRs for balanced nodes as well as the CADR technique for any kind of nodes in the index tree

[5] We compiled our programs by the compiler of egcs-1.1.2 release (*http://gcc.gnu.org/*)

$$\text{Max\_entry\_number} = \left\lfloor \frac{\text{PAGE\_SIZE - size of additional information}}{\text{size of an entry}} \right\rfloor \quad (\mathbf{10})$$

With reference to formula 10, additional information can be varied among data pages of different node types. This information should be common for all entries stored in the same page. For example, in each internal page we store an additional variable *ID* to identify the node type, or in each balanced page we store several additional variables like *ID*, *Height* to keep the height of this balanced node, and *Count* to keep the entry number of leaf nodes in this balanced node. Note that, parameters like minO_E, minBN_E, or the minimum entry number of internal nodes for a disk page are depended on users, application domains and they can be tuned at the tree building time. Interestingly, if parameters minBN_E=maxBN_E=1 (the minimum and maximum entry numbers of each balanced page, individually) and only MBRs are employed (without BSs), the SH-tree will look quite like the LSD-tree and LSDh-tree.

Other implementation aspect of the SH-tree is the update for BSs in the course of the tree building. Because the SH-tree only employs BSs, which are approximations of MBSs, to reduce the computational cost while still remaining good approximations we apply an update policy for BSs similarly to that of the SS-tree [45]: After several times, say k, that a leaf node is changed, we will recalculate and update information related to its BS. Currently, the value of k is set to 5 as proposed in the SS-tree.

Besides, for the testing purpose, we implemented the SH-tree as a "memory-based version", although the SH-tree is essentially a disk-based MAM (with notes as posed above). It means that the SH-tree in our current implementation is entirely loaded into the main memory and therefore, to compute the IO-cost during query processing we count the disk page number that must be accessed instead of the "real" IO-time that the system must wait for these disk page accesses (this was also used for many MAMs for the testing purpose)[6]. Note that, with this memory-based version of the SH-tree, our setting for the disk page size (8KB) can be changed without affecting the "real" IO-time needed to load the required disk pages. In addition, each tree node also maintains a pointer pointing to its parent node. This will facilitate later traversal operations over the SH-tree. Those pointers will also be used as links to siblings as developing concurrency control and recovery techniques for the SH-tree (see [20]). Again, information about such pointers is not stored in the secondary storage, but it is dynamically created during loading (or creating) the tree into the main memory.

Except for some things that we present above, there are many other problems which one must deal with as developing a program to build any MAM from scratch. See [20, 31, 2] for more discussions.

5.2 The SH-tree Performance with k-Nearest Neighbor Queries

As mentioned before, the state-of-the-art algorithms for processing k-NN queries in spatial databases are shown in [40] and [28]. First, this section presents adaptations of these algorithms to the SH-tree, and then experimental results as well as other critical comments. Importantly, for both adapted algorithms, we do not employ the MINMAXDIST metric which was originally introduced in [40] to prune the tree during the traversal. The key reasons for this are because internal nodes of the SH-tree do not use MBRs, but they employ on BRs instead. Therefore, the MINMAXDIST metric cannot be applied to these nodes. Moreover, as discussed in [15], the authors also proved that only the MINDIST metric is also enough for k-NN algorithm introduced in [40] because any disk pages that can be pruned by making use of the MINMAXDIST metric can also be pruned without this concept. We therefore do not apply the

[6] Such an IO-cost computation method has also been used in most performance evaluations of MAMs. Several authors have also reported the number of accessed objects together with the disk access number. The accessed object number can also be represented for MAMs' computational cost (CPU-cost), somewhat

MINMAXDIST metric to the SH-tree at all (cf. appendix). In addition, this also reduces unnecessary computational costs during the query processing.

Algorithms: First, we introduce a k-NN algorithm adapted from the one proposed in [40]. Figure 8 below shows pseudo-code of this adapted algorithm.

```
Adapted algorithm 1:
1:    kNN_Query(Node,Obj,kNN_buffer,k,BR)
2:    NODE          Node          //current node
3:    DATAOBJECT    Obj           //query object
4:    OBJECTLIST    kNN_buffer    //buffer of result objects
5:    INT           k             //length of kNN_buffer
6:    BOUNDINGREC   BR            //bounding region of Node
7:    Case (Node.type) of
8:    INTERNAL_NODE:
9:      //Compute BR for left and right child
10:     BOUNDINGREC BRleft=BR∩(d≤Node.up);
11:     BOUNDINGREC BRright=BR∩(d≥Node.lo);
12:     //MINDIST from Obj to BRleft and BRright
13:     FLOAT   leftdist=MINDIST(Obj,BRleft);
14:     FLOAT   rightdist=MINDIST(Obj,BRright);
15:     If leftdist<rightdist then
16:     If (kNN_buffer.num<k)or(kNN_buffer[k].dist>leftdist)then
17:           kNN_Query(Node.left,Obj,kNN_buffer,k,BRleft);
18:     else  return;
19:     If (kNN_buffer.num<k)or(kNN_buffer[k].dist>rightdist)then
20:           kNN_Query(Node.right,Obj,kNN_buffer,k,BRright);
21:     else //leftdist³rightdist
22:     If (kNN_buffer.num<k)or(kNN_buffer[k].dist>rightdist)then
23:           kNN_Query(Node.right,Obj,kNN_buffer,k,BRright);
24:     else  return;
25:     If (kNN_buffer.num<k)or(kNN_buffer[k].dist>leftdist)then
26:           kNN_Query(Node.left,Obj,kNN_buffer,k,BRleft);
27:   BALANCED_NODE:
28:     For each entry Leaf do
29:       If (kNN_buffer.num<k)or
             (kNN_buffer[k].dist>Distance(Obj,Leaf)) then
30:           kNN_Query(Leaf,Obj,kNN_buffer,k,NULL);
31:   LEAF_NODE:
32:     For each object Object do
33:       FLOAT  dist=ObjectDistance(Obj, Object);
34:       If (dist<kNN_buffer[k].dist) then
35:           kNN_buffer[k]=Object;
36:           kNN_buffer.AscendingSort();
37:   EndCase
```

Figure 8: Pseudo-code of the adapted k-NN algorithm 1

The algorithm presented in Figure 8 implements a depth-first search that is similar to one introduced in [40]. Nevertheless, depending on the special structure of the SH-tree, this adapted algorithm does not use the so-called ABL - Active Branch List. At the internal nodes, the algorithm recursively calls itself according to the MINDIST from the query object *Obj* to its left child, *leftdist*, and its right child, *rightdist*. At the balanced node, the leaf page is loaded if the distance *Distance(Obj, Leaf)* (line 29) from *Obj* to *Leaf* is less than the current k^{th} nearest neighbor or there are less than k nearest neighbors have been found so far. Note that, the distance *Distance(Obj, Leaf)* is defined as the longer one between the minimum distances to the leaf's MBR and BS (Bounding Sphere). The function *ObjectDistance(Obj, Object)* (line 33) is used to compute the distance between two objects, which is the Euclidean metric (L_2 metric) in our implementation. Because the ABL is not used here, its maintenance costs are omitted, which includes all costs for generating and sorting the ABL (again, see [40] for the

detail description of the original algorithm). This is also a reason that improves the CPU-cost of this adapted algorithm as we will see in later experimental results. One important thing to note is that this algorithm only works as we know the value k in advance. This is different from the next adapted algorithm.

Next, we introduce a k-NN algorithm adapted from the one introduced in [28]. The original algorithm has been proven to be optimal in terms of the accessed page number [12], which will be confirmed later by our experimental results. Figure 9 shows pseudo-code of our adapted algorithm to the SH-tree. It employs a priority queue *PQ* and the result objects are output one by one (line 25). Nonetheless, there is a difference here: This adapted algorithm needs the *effective* bounding regions (BRs) of the data space indexed and its subspaces in order to compute MINDIST for the left and right children of internal nodes from the given query object. This requirement is easy to be fulfilled because information about the effective data space is stored as a part of the tree meta-data during creating any SH-tree. Therefore, BRs of descendants (subspaces) of the indexed data space can also be computed when necessary as presented in section 3.1. Besides, the algorithm also employs two functions *Distance* and *ObjectDistance* which are described above. Although this adapted algorithm results in the better IO-cost, it must incur the maintenance costs for the priority queue *PQ* and thus the CPU-cost maybe higher than that of the first adapted algorithm in certain cases.

Adapted algorithm 2':

```
1:    kNN_Query(Obj,PQ,BR)
2:    DATAOBJECT     Obj    //query object
3:    PRIORITYQUEUE  PQ     //priority queue
4:    BOUNDINGREC    BR     //bounding region of the data space
5:    PQ.Push(Root,Root.type,0.0,BR);
6:    While not PQ.IsEmpty() do
7:      PRIORITYQUEUE      top=PQ.Top();
8:      Case top.type of
9:      INTERNAL_NODE:
10:      //compute BR for its left and right child
11:      BOUNDINGREC BRleft=top.BR∩(d≤top.up);
12:      BOUNDINGREC BRright=top.BR∩(d≥top.lo);
13:      //compute MINDIST from Obj to BRleft and BRright
14:      FLOAT      leftdist=MINDIST(Obj,BRleft);
15:      FLOAT      rightdist=MINDIST(Obj,BRright);
16:      PQ.Push(top.left,top.left.type,leftdist,BRleft);
17:      PQ.Push(top.right,top.right.type,rightdist,BRright);
18:     BALANCED_NODE:
19:      For each entry Leaf do
20:      PQ.Push(Leaf,LEAF_NODE,Distance(Obj,Leaf),NULL);
21:     LEAF_NODE:
22:      For each object Object do
23:      PQ.Push(Object,OBJTYPE,ObjectDistance(Obj,Object),NULL);
24:     OBJTYPE:
25:      Report Object as the next nearest neighbor object;
26:     EndCase
27:   EndWhile
```

Figure 9: Pseudo-code of the adapted k-NN algorithm 2'

However, we have done intensive experiments with this adapted algorithm to compare the SH-tree with the SR-tree and found that the IO-cost during the query processing over the SH-tree is much less than that of the SR-tree but the CPU-cost is approximately equal to that of the SR-tree. This is not expected, especially as the main memory can keep the whole SH-tree. Because the main reason of this is the maintenance costs for the priority queue *PQ* so we will introduce another algorithm which is also adapted from the one proposed in [28, 29] but it needs a smaller memory space for the priority queue. All experimental results with the SH-tree later that is related to the adaptation of the

original algorithm presented in [28] are carried out with this new adapted algorithm. Figure 10 below sketches the pseudo-code of the algorithm.

```
Adapted algorithm 2:
1:    kNN_Query(Obj,PQ,BR)
2:    DATAOBJECT     Obj    //query object
3:    PRIORITYQUEUE  PQ     //priority queue
4:    BOUNDINGREC    BR     //bounding region of the data space
5:    PQ.Push(Root,Root.type,0.0);
6:    While not PQ.IsEmpty() do
7:      PRIORITYQUEUE      top=PQ.Top();
8:      Case top.type of
9:      INTERNAL_NODE:
10:       //compute BR for its left and right child
11:       BOUNDINGREC      BRleft=BR∩(d≤top.up);
12:       BOUNDINGREC      BRright=BR∩(d≥top.lo);
13:       //compute MINDIST from Obj to BRleft and BRright
14:       FLOAT     leftdist=MINDIST(Obj,BRleft);
15:       FLOAT     rightdist=MINDIST(Obj,BRright);
16:       PQ.Push(top.left,top.left.type,leftdist);
17:       PQ.Push(top.right,top.right.type,rightdist);
18:     BALANCED_NODE:
19:       For each entry Leaf do
20:         PQ.Push(Leaf,LEAF_NODE,Distance(Obj,Leaf));
21:     LEAF_NODE:
22:       For each object Object do
23:         PQ.Push(Object,OBJTYPE,ObjectDistance(Obj,Object));
24:     OBJTYPE:
25:       Report Object as the next nearest neighbor object;
26:     EndCase
27:   EndWhile
```

Figure 10: Pseudo-code of the adapted k-NN algorithm 2

Algorithm described in Figure 10 looks very similar to that in Figure 9 except for one main modification: The priority queue *PQ* does not keep BRs of internal nodes, but they are calculated approximately using the BR of the whole effective data space. In fact, this makes the traversal of the SH-tree through internal nodes look like selection process for a suitable balanced node (and a leaf node) to insert a new data object (cf. section 4.1)[7]. Specially, as a query object is chosen among data objects indexed, this algorithm is very fast to determine the leaf node it should belong to. Hence, its NNs are hopefully to be found efficiently. Besides, even in the case we cannot immediately find the correct NNs of such a query, this method is still powerful to search for its approximate NNs.

Experimental Results: Our tests mainly concentrate on comparing performance of the SR-tree to that of the SH-tree with respect to the k-NN query processing. We choose the SR-tree to carry out comparisons because one of the main purposes of the SH-tree is to alleviate its fan-out problem (cf. section 2) and the SR-tree is also one of the most prominent index trees at the moment. The SR-tree source code has been made available at *http://research.nii.ac.jp/~katayama/homepage/* by the authors. In addition, as the SH-tree, searching on the SR-tree is also implemented with two original algorithms in [40] and [28]. In the below charts, legends *"SH-tree with algorithm 1"* and *"SH-tree with algorithm 2"* (respectively, *"SR-tree with algorithm 1"* and *"SR-tree with algorithm 2"*) are correspondent with each of these algorithms, individually.

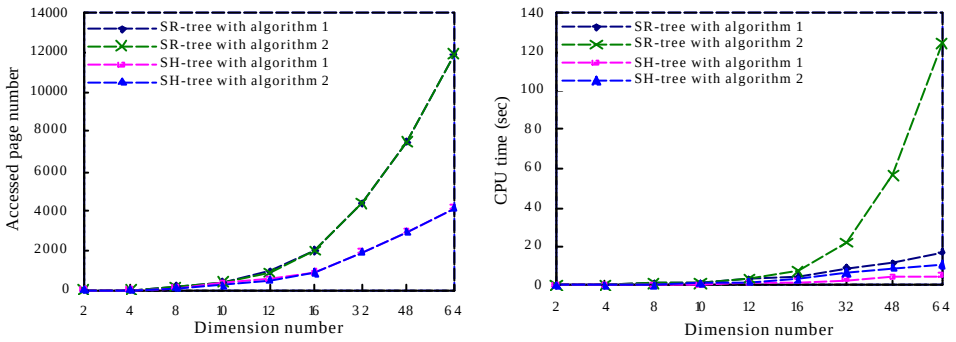
For the tests, we use both uniformly distributed and real data sets. The dimension number *d* of uniformly distributed data sets varies from 2 to 64 and each of them has 100000 tuples of such multidimensional data points. The real data set consists of 60000 9-dimensional image feature vectors (downloaded from *http://kdd.ics.uci.edu/*), which are extracted from images and based on their color moments. All the tests are tackled on

[7] The SH-tree internal nodes are treated as ones of the KD-tree (alg. 2) and the R-tree (alg. 2') based indexes

a Sun Ultra Sparc-II 300MHz with the main memory is 256 Mbytes and some Gigabytes of hard disk capacity. As mentioned above, all programs are implemented in C++ and the page size is 8Kb to meet with the disk block size of the operating system.

There are some special test conditions we have done with the SH-tree: For uniformly distributed data sets, the minimum storage utilization factor is set to 40% for leaf nodes and the reinsertion factor is set to 60%. For the real data set, these parameters are 30% and 50%, individually. For all experiments, we found that if the dimension number $d \leq 12$, we do not use redistribution policy between leaves of a balanced node during the insertion (cf. section 4.1). In our experiments, this policy is employed for uniformly distributed data sets, whose dimension number is greater 12 and in these cases, the reinsertion factor is again set to only 30% as proposed in [11] for the R*-tree. Moreover, we carry out queries to find 15 NNs in all experiments. For each test, 100 query points are randomly selected among the corresponding data sets.

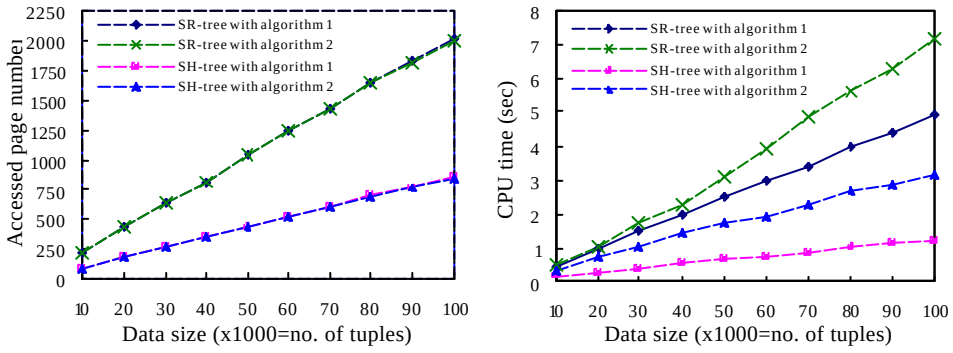
Figure 11 shows experiment results to evaluate the performance according to various dimension numbers of the uniformly distributed data sets. The charts indicate that the SH-tree totally outperforms the SR-tree with respect to both adapted algorithms in terms of both CPU-time and IO-cost. Besides, the first adapted algorithm (Figure 8) shows better CPU-time over the second one (Figure 10) for all the tests with the SH-tree. However, the accessed page number of the second algorithm is less. Although research results in [8] have pointed out that experiments with high-dimensional synthesis data sets are not meaningful, we still want to show here that the fan-out problem of the SR-tree really causes more disk page accesses during the query processing.



(a) (b)

Figure 11: Variety in dimension number of uniformly distributed data sets

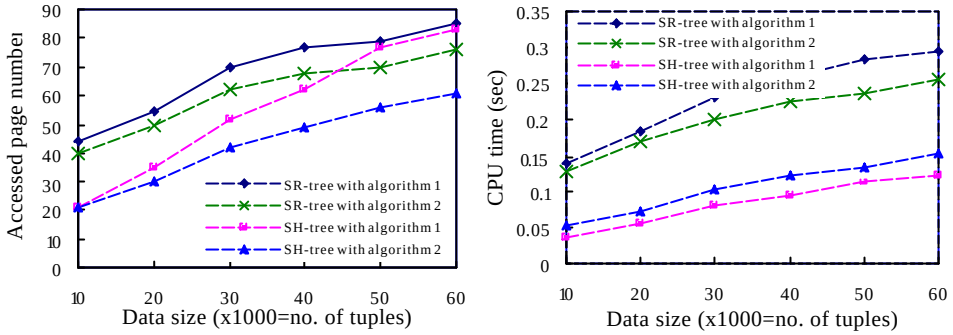
Figure 12 shows the SH-tree performance with a variety in data size of the 16-dimensional uniformly distributed data set. The results in both Figures 11 and 12 partly confirm a conclusion in [29] that the second adapted algorithm outperforms the first one, but only in terms of the IO-cost in this case, i.e. with the SH-tree. It also indicates well-scaled possibility of the SH-tree concerning a variety of data size.



(a) (b)

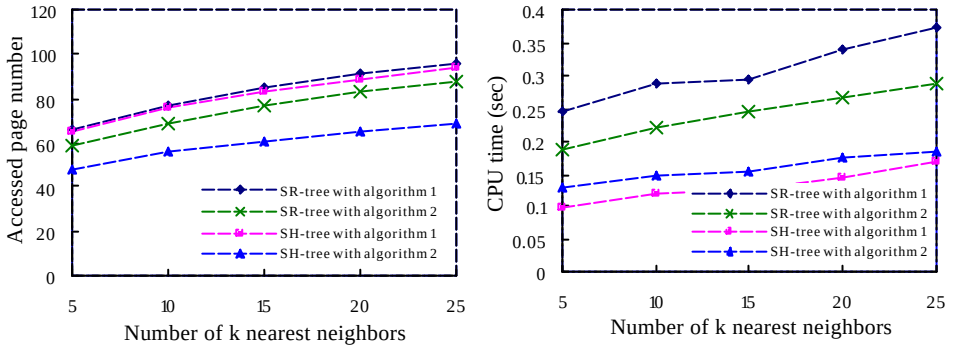
Figure 12: Variety in data size of 16-d uniformly distributed data set

Figure 13 gives the performance evaluation of the SH-tree and SR-tree with a variety in data size of the 9-dimensional real data set. Experimental results also prove superiority of the SH-tree to the SR-tree. Specially, the second adapted algorithm shows a better result over the first one. This again confirms a conclusion in [12]: the second algorithm is optimal in terms of the accessed page number.



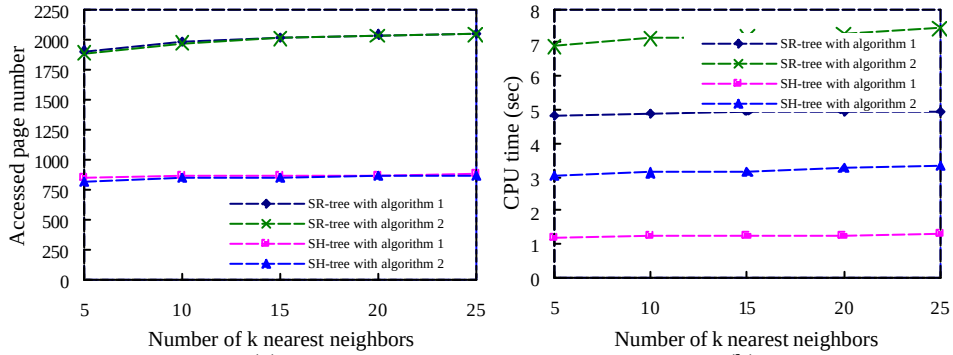
(a) (b)

Figure 13: Variety in data size of 9-d real data set



(a) (b)

Figure 14: Variety in number of NN (9-d real data set)



(a) (b)

Figure 15: Variety in number of NN (16-d synthetic data set)

Figures 14 and 15 depict the SH-tree performance concerning various numbers of k nearest neighbors for both real data set and 16-dimensional synthetic data set. The SH-tree also outperforms the SR-tree in all cases.

Interestingly, our adapted algorithm in Figure 10 does not use BRs of internal nodes like the one in Figure 9 but the accessed page number of the two adapted algorithms is

comparable (nearly the same in all of our experiments). This is a surprise result and we still have no any mathematical explanation for this phenomenon.

To conclude this section, we give explanations for the above experimental results. For all the tests, the CPU-cost of the SH-tree is less than that of the SR-tree because the algorithms do not have to compute the distance from a given query object to BSs at each internal node of the SH-tree, which must be done during the SR-tree based search. For the adapted algorithm in Figure 10, its maintenance costs for the priority queue are reduced because we proposed an adapted algorithm using a much smaller priority queue in size. Besides, the SH-tree based search accesses less data pages because of its special structure as discussed in section 3. Our experiments confirmed this theoretical analysis[8]. But, the SH-tree just employs BRs at the internal nodes, so pruning as traversing may not be suitable for very skewed data. In such cases, the IO-cost of the SH-tree can be higher than that of the SR-tree. Moreover, "push" operations of the priority queue at lines 16, 17, 20 and 23 of the second adapted algorithm as described in Figure 10 are expensive, thus it accounts for the higher CPU-time of this algorithm as comparing to that of the first adapted algorithm in Figure 8. These operations must ensure the priority queue PQ in ascending order. The same operation in Figure 8 only occurs at line 36. In our tests, although the IO-cost of the second algorithm is less than that of the first algorithm, it is still not enough to absolutely compensate for the CPU-time as experimental results shown in [29]. But, the total elapsed time for the second algorithm with the real data set is better. For those reasons, selection of an efficient algorithm is also depended on size and distribution of data indexed. The first adapted algorithm is suitable for SH-trees that fit in the main memory because its CPU-cost is lower, while the second one is optimal in terms of IO-cost as shown in [12]. In addition, the second adapted algorithm is the most suitable one for processing k-NN queries without knowing values k in advance[9]. In such circumstances, the second algorithm shows much better cumulative costs than the first one [29].

6 Discussions

Despite MAMs' advantages, to support similarity search capabilities efficiently and flexibly in FQASs such as the VQS [36, 37] or Content-Based Information Retrieval systems [46, 47], we should give facilities for supporting MAMs as powerful data access methods in DBMSs. But, to integrate MAMs into standard DBMSs smoothly, we must first address several non-trivial issues to make MAMs functionate properly and efficiently in those DBMSs. These issues are well-known, but still not well-addressed yet, such as a new MAM should be in concord with available facilities in the DBMS as query optimizer, efficient transaction processing, etc.

As shown in previous sections, the SH-tree is a very promising MAM. However, before the SH-tree can be integrated as an access method to a commercial strength DBMS as well as can become useful for a wide range of application domains, we need to develop efficient techniques to provide facilities for the query optimizer, transactional access to data via the SH-tree, and quick building of the tree. In [20], we disclosed and discussed these problems theoretically, together with introducing a simple but efficient approach to estimating approximate IO-cost and accessed object number for k-NN and range queries over the SH-tree, an algorithm for local dynamic bulk loading of the tree, and an approach to preserving the tree consistency in presence of concurrent operations as insertions, deletions, and modifications. More work should be carried out towards bringing the SH-tree out to the commercial world.

[8] With 16-d synthetic data set, the effective storage utilization of leaves without the redistribution policy is just 69.39%, but as we apply this policy, it is 96.67%. With 9-d real data set, this value is 78.64% even without this policy (in this case, if we apply this policy, the search performance is degraded)

[9] In [20], we introduced a variant of the second adapted algorithm as shown in Figure 9 that brings out a much better CPU-cost as comparing to that of the first adapted algorithm but it must sacrifice this advantage

Another interesting problem that we have intensively investigated and got some initial encouraging results is to facilitate advanced query types using the SH-tree. A vast number of advanced query types were introduced in [20], including approximate similarity queries, multi-feature k-NN queries, spatial joins, etc., and we also presented efficient approaches to some of them. Interested readers can refer to [20] for valuable discussions about similarity search in modern database applications.

7 Conclusions and Future Work

We introduced the SH-tree for indexing multidimensional data and presented issues that need to be prepared for integrating it into commercial strength DBMSs. The SH-tree is a flexible MAM to support similarity searches. It is a well-combined structure of both the SR-tree and KD-tree based index techniques. The SH-tree carries positive aspects of both the KD-tree and R-tree families. While the fan-out problem of the SR-tree is overcome by employing the KD-tree like representation for partitions of internal nodes, the SH-tree still takes advantages of the SR-tree by using balanced nodes, which are the same as internal nodes of the SR-tree. Besides, the SH-tree has been designed to manage both point and extended spatial objects. In addition, we also introduced a new concept for the SH-tree, called the extended balanced tree. It implies that the SH-tree is unnecessary to be exactly balanced, but the querying performance is still not deteriorated and the maintenance cost for the tree balance is reduced dramatically.

Furthermore, we also presented algorithms for important similarity query types as range, NN and k-NN queries. Notably, we presented two adapted algorithms, which are originated from the state-of-the-art research results, for the SH-tree to efficiently process k-NN queries in multidimensional databases. The experiments were conducted on both uniformly distributed and real data sets. The results have shown that the SH-tree with these adapted algorithms processes k-NN queries efficiently and outperforms the SR-tree by an order of magnitude in all experiments. Our experimental evaluations confirmed conclusions of previous researches in [29, 12] concerning the optimality in terms of the IO-cost of the algorithm presented in [28]. Besides, these results also proved the correctness of our theory analyses: The SH-tree can efficiently scale to high-dimensional spatial databases. These achievements give us a solid base with respect to the indexing aspects for further research activities in the future.

As a part of the future work, we intend to compare the SH-tree to other prominent MAMs such as the LSDh-tree, X-tree, SS-tree, M-tree, etc. Besides, extending the concepts of the SH-tree to form other new MAMs is also worth considering. For instance, motivating from experimental results and comments in section 5.2, we are thinking about *multi-level balanced nodes* in the SH-tree: Instead of using one-level balanced nodes as discussed in previous sections, we apply multi-level balanced nodes and each sub-tree of the SH-tree (from point that a DP/R-tree based index technique is applied) will look like that of the R-tree and its variants. The real effectiveness and efficiency of this SH-tree are still an open question. Also, dealing with related issues towards integrating the SH-tree into existing commercial DBMSs will be a subject of great interest for our future research activities.

References

[1] P.K. Agarwal, J. Erickson: *Geometric Range Searching and Its Relatives*. Advances in Discrete and Computational Geometry, Contemporary Mathematics 223, American Mathematical Society Press, 1999, pp. 1-56

[2] P.M. Aoki: *Generalizing "Search" in Generalized Search Trees (Extended Abstract)*. Proc. 14th Int. Conf. on Data Engineering, Feb 23-27, 1998, Orlando, Florida, USA, pp. 380-389

[3] R. Bayer: *The Universal B-Tree for Multidimensional Indexing: General Concepts*. Proc. Int. Conf. on Worldwide Computing and Its Apps., Mar 1997, Tsukuba, Japan, pp. 198-209

[4] C. Boehm , S. Berchtold , D.A. Keim: *Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases*. ACM Computing Surveys (CSUR), 33(3), Sept 2001, 322-373
[5] S. Berchtold, C. Boehm, H-P. Kriegel: *The Pyramid Technique: Towards Breaking the Curse of Dimensionality*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Jun 2-4, 1998, Seattle, Washington, USA, pp. 142-153
[6] S. Berchtold, H.P. Kriegel: *S3: Similarity Search in CAD Database Systems*. Proc. ACM SIGMOD Int. Conf. on Management of Data, May 13-15, 1997, Arizona, USA, pp. 564-567
[7] J.L. Bentley: *Multidimensional Binary Search Trees Used for Associative Searching*. Communications of the ACM, 18(9), Sept 1975, 509-517
[8] K.S. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft: *When Is "Nearest Neighbor" Meaningful?*. Proc. 7th Int. Conference on Database Theory, Jan 10-12, 1999, Jerusalem, Israel, pp.217-235
[9] S. Berchtold, D.A. Keim, H-P. Kriegel, T. Seidl: *Indexing the Solution Space: A New Technique for Nearest Neighbor Search in High-Dimensional Space*. IEEE Trans. on Knowledge and Data Engineering, 12(1), Jan/Feb 2000, 45 –57
[10] S.Berchtold, D.A. Keim, H-P. Kriegel: *The X-tree: An Index Structure for High-Dimensional Data*. Proc. 22nd Int. Conf. on Very Large Data Bases, Sept 1996, Mumbai, India, pp. 28-39
[11] N. Beckmann, H-P. Kriegel, R. Schneider, B. Seeger: *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*. Proc. ACM SIGMOD Int. Conf. on Management of Data, May 23-25, 1990, Atlantic City, NJ, USA, pp. 322-331
[12] C. Boehm: *A Cost Model for Query Processing in High Dimensional Data Spaces*. ACM Trans. on Database Systems (TODS), 25(2), Jun 2000, 129-178
[13] T. Bozkaya, Z.M. Oezsoyoglu: *Distance-Based Indexing for High-Dimensional Metric Spaces*. Proc. ACM SIGMOD Int. Conf. on Management of Data, May 13-15, 1997, Tucson, Arizona, USA , pp. 357-368
[14] S. Brin: *Near Neighbor Search in Large Metric Spaces*. Proc. 21st Int. Conf. on Very Large Data Bases, Sept 11-15, 1995, Zurich, Switzerland, pp. 574-584
[15] K.L. Cheung, A.W-C. Fu: *Enhanced Nearest Neighbour Search on the R-tree*. ACM SIGMOD Record, 27(3), Sept 1998, 16-21
[16] K. Chakrabarti, S. Mehrotra: *The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces*. Proc. 15th Int. Conf. on Data Engineering, Mar 23-26, 1999, Sydney, Australia , pp. 440-447
[17] K. Chakrabarti, E.J. Keogh, S. Mehrotra, M.J. Pazzani: *Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases*. ACM Trans. on Database Systems (TODS), 27(2), Jun 2002, 188-228
[18] E. Chávez, G. Navarro, R. Baeza-Yates, J.L. Marroquín: *Searching in Metric Spaces*. ACM Computing Surveys (CSUR), 33(3), Sept 2001, 273 – 321
[19] P. Ciaccia, M. Patella, P. Zezula: *M-tree: An Efficient Access Method for Similarity Search in Metric Spaces*. Proc. 23rd Int. Conf. on Very Large Data Bases, Aug 25-29, 1997, Athens, Greece, pp. 426-435
[20] T.K. Dang: *Semantic Based Similarity Searches in Database Systems (Multidimensional Access Methods, Similarity Search Algorithms)*. PhD Thesis, FAW-Institute, Johannes Kepler University of Linz, Austria, May 2003
[21] T.K. Dang, J. Kueng, R. Wagner: *Efficient Processing of k-Nearest Neighbor Queries in Spatial Databases with the SH-tree*. Proc. 3rd Int. Conf. on Information Integration and Web-based Applications and Services, Sept 10-12, 2001, Linz, Austria, pp. 425-435
[22] V. Gaede, O. Guenther: *Multidimensional Access Methods*. ACM Computing Surveys (CSUR), 30(2), Jun 1998, 170 – 231
[23] D. Greene: *An Implementation and Performance Analysis of Spatial Data Access Methods*. Proc. 5th Int. Conf. on Data Engineering, Feb 6-10, 1989, California, USA, pp. 606-615
[24] A. Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching*. Proc. ACM SIGMOD Conference, Jun 18-21, 1984, Boston, Massachusetts, USA, pp. 47-57
[25] A. Henrich: *A Hybrid Split Strategy for k-d-Tree Based Access Structures*. Proc. 4th ACM Workshop on Advances on Advances in Geographic Information Systems, Nov 15-16, 1996, Rockville, Maryland, USA, pp. 1-8
[26] A. Henrich: *Improving the Performance of Multi-Dimensional Access Structures Based on k-d-Trees*. Proc. 12th Int. Conf. on Data Engineering, Feb 26-Mar 1, 1996, New Orleans, Louisiana, USA, pp. 68-75
[27] A. Henrich: *The LSDh-tree: An Access Structure for Feature Vectors*. Proc. 14th Int. Conf. on Data Engineering, Feb 23-27, 1998, Florida, USA, pp. 362-369

[28] G.R. Hjaltason, H. Samet: *Ranking in Spatial Databases*. Proc. 4th Int. Symposium on Large Spatial Databases, Aug 6-9, 1995, Portland, Maine, USA, pp. 83-95
[29] G.R. Hjaltason, H. Samet: *Distance Browsing in Spatial Databases*. ACM Trans. on Database Systems (TODS), 24(2), Jun 1999, 265-318
[30] G.R. Hjaltason, H. Samet: *Speeding up Construction of PMR Quadtree-based Spatial Indexes*. VLDB Journal, 11(2), 2002, 109-137
[31] J.M. Hellerstein, J.F. Naughton, A. Pfeffer: *Generalized Search Trees for Database Systems*. Proc. 21st Int. Conf. on Very Large Data Bases, Sept 11-15, 1995, Switzerland, pp. 562-573
[32] A. Henrich, H-W. Six, P. Widmayer: *The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects*. Proc. 15th Int. Conf. on Very Large Data Bases, Aug 22-25, 1989, Amsterdam, The Netherlands, pp. 45-53
[33] N. Katayama, S. Satoh: *The SR-Tree: An Index Structure for High Dimensional Nearest Neighbor Queries*. Proc. ACM SIGMOD Int. Conf. on Management of Data, May 13-15, 1997, Tucson, Arizona, USA, pp. 369-380
[34] R. Kurniawati, J.S. Jin, J.A. Shepherd: *SS+-Tree: An Improved Index Structure for Similarity Searches in a High-Dimensional Feature Space*. Proc. Storage and Retrieval for Image and Video Databases V (SPIE), Vol. 3022, Feb 8-14, 1997, San Jose, CA, USA, pp. 110-120
[35] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, Z. Protopapas: *Fast Nearest Neighbor Search in Medical Image Databases*. Proc. 22nd Int. Conf. on Very Large Data Bases, Sept 3-6, 1996, Mumbai, India, pp. 215-226
[36] J. Kueng, J. Palkoska: *VQS–A Vague Query System Prototype*. Proc. 8th Int. Workshop on Database and Expert Systems Applications, Sept 1-2, 1997, Toulouse, France, pp. 614-618
[37] J. Kueng, J. Palkoska: *An Incremental Hypercube Approach for Finding Best Matches for Vague Queries.* Proc. 10th Int. Conf. on Database and Expert Systems Applications, Aug 30–Sept 3, 1999, Florence, Italy, pp. 238-249
[38] D.B. Lomet, B. Salzberg: *The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance*. ACM Trans. on Database Systems (TODS), 15(4), Dec 1990, 625-658
[39] B.C. Ooi, K.J. McDonell, R. Sacks-Davis: *Spatial kd-tree: An Indexing Mechanism for Spatial Databases*. Proc. 11th Annual Int. Computer Software and Applications Conference (COMPSAC'87), 1987, Tokyo, Japan, pp. 433- 438
[40] N. Roussopoulos, S. Kelley, F. Vincent: *Nearest Neighbor Queries*. Proc. ACM SIGMOD Int. Conf. on Management of Data, May 22-25, 1995, San Jose, California, USA, pp. 71-79
[41] J.T. Robinson: *The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Apr 29-May 1, 1981, Ann Arbor, Michigan, USA, pp. 10-18
[42] T. Seidl, H-P. Kriegel: *Optimal Multi-Step k-Nearest Neighbor Search*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Jun 2-4, 1998, Washington, USA, pp. 154-165
[43] T. Seidl, H-P. Kriegel: *Efficient User-Adaptable Similarity Search in Large Multimedia Databases*. Proc. 23rd Int. Conf. on Very Large Data Bases, Aug 25-29, 1997, Athens, Greece, pp. 506-515
[44] T.K. Sellis, N. Roussopoulos, C. Faloutsos: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Proc. 13th Int. Conf. on Very Large Data Bases, Sept 1-4, 1987, Brighton, England, pp. 507-518
[45] D.A. White, R. Jain: *Similarity Indexing with the SS-Tree*. Proc. 20th Int. Conf. on Data Engineering, Feb 26-Mar 1, 1996, New Orleans, Louisiana, USA, pp. 516-523
[46] W. Niblack, R. Barber, W. Equitz, M. Flickner, E.H. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, G. Taubin: *The QBIC Project: Querying Images by Content, Using Color, Texture, and Shape*. Proc. Storage and Retrieval for Image and Video Databases (SPIE), Vol. 1908, Jan 31–Feb 5, 1993, San Jose, CA, USA, pp. 173-187
[47] T. Huang, S. Mehrotra, K. Ramchandran: *Multimedia Analysis and Retrieval System (MARS) Project.* Proc. 33rd Annual Clinic on Library Application of Data Processing –Digital Image Access and Retrieval, University of Illinois at Urbana-Champaign, March, 1996

Appendix: MINDIST and MINMAXDIST

Definition 1 (MINDIST): *The distance of a point $P(p_1, p_2, \ldots, p_n)$ in Euclidean space of n dimensions from a hyper-rectangle $R(lb_1, lb_2, \ldots, lb_n, ub_1, ub_2, \ldots, ub_n)$ in the same space, denoted MINDIST(P, R), is computed as follows:*

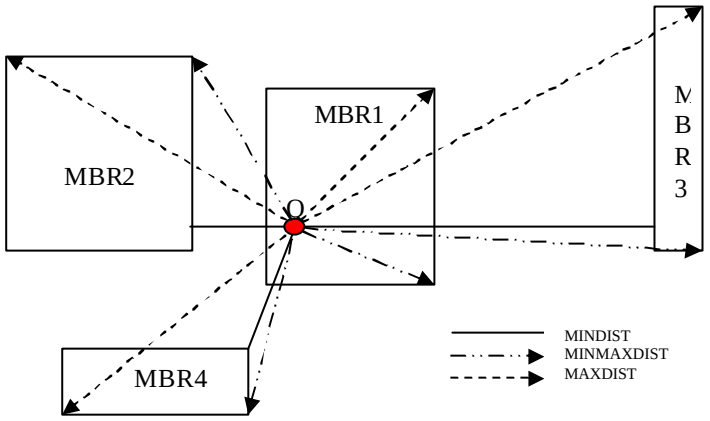
$$MINDIST^2(P, R) = \sum_{i=1}^{n} |p_i - r_i|^2 \quad where\ r_i = \begin{cases} lb_i & if\ p_i < lb_i \\ ub_i & if\ p_i > ub_i \\ p_i & otherwise \end{cases}$$

Definition 2 (MINMAXDIST): *Given a point $P(p_1, p_2, \ldots, p_n)$ in Euclidean space of n dimensions and a hyper-rectangle $R(lb_1, lb_2, \ldots, lb_n, ub_1, ub_2, \ldots, ub_n)$ in the same space. We define MINMAXDIST(P, R) as:*

$$MINMAXDIST^2(P, R) = \min_{1 \le k \le n} \left(|p_k - rm_k|^2 + \sum_{\substack{i \ne k \\ 1 \le i \le n}} |p_i - rM_i|^2 \right)$$

where

$$rm_k = \begin{cases} lb_k & if\ p_k \le \frac{lb_k + ub_k}{2} \\ ub_k & otherwise \end{cases}, \text{ and } rM_i = \begin{cases} lb_i & if\ p_i \ge \frac{lb_i + ub_i}{2} \\ ub_i & otherwise \end{cases}$$



MINDIST, MINMAXDIST, and MAXDIST

During the search process, MINDIST (MINMAXDIST) ordering is the optimistic (pessimistic) heuristic to choose a pruning strategy. MINMAXDIST ensures that there is at least a data object in the intersection of the hyper-rectangle R and the range centered at P with a radius MINMAXDIST. The reason here is that in the R-tree every surface hyper-plane of a MBR must contain a point or have at least a point in common with a data object. This is not always the case for other MAMs, e.g. the SH-tree, and so in [4] the authors introduced MAXDIST metric which is the greatest possible distance from the query object to a data object in a page region. MAXDIST is always greater than zero, even if the query object is located inside the page region, which can be a hyper-rectangle, a hyper-sphere, etc., or an intersection between some of them. This metric is a more pessimistic heuristic than MINMAXDIST metric during the pruning, but it can be used for any MAMs. Depending on MINDIST and MINMAXDIST metrics, the authors of [40] proposed three pruning strategies for MBRs during the search as follows:

(1) A MBR X with MINDIST(Q, X), where Q is the query object, greater than the MINMAXDIST(Q, Y) of another MBR Y is discarded because it cannot contain the NN. This strategy is used in the *downward pruning*.

(2) A data object O with the actual distance from the query dist(Q, O) greater than MINMAXDIST(Q, X) of a MBR X is discarded because X contains at least an object O' which is nearer to Q than O. This strategy is used in the *downward pruning*, too.

(3) Every MBR X with MINDIST(Q, X) greater than the actual distance from the query to a data object O dist(Q, O) is discarded because X cannot enclose an object nearer to Q than O. This strategy is used in the *upward pruning*.

The details of these three strategies and many other interesting discussions can be found in the original paper. Figure below illustrates MINDIST, MINMAXDIST, and MAXDIST in a possible example two-dimensional space (note that, MINDIST(Q, MBR1) = 0 in this example figure).

There is an interesting fact that in a recent paper [15], the authors proved that any disk pages can be pruned by making use of MINMAXDIST metric can also be pruned without this concept. Therefore, in their modified algorithm, they used only the third pruning strategy above. Their motive observation is that the computation of MINMAXDIST is computationally expensive with a complexity of O(d), where d is the number of dimensions, and it should be avoided. Avoiding this computation will lead to improve the search performance in CPU-time. This is a reason why we also do not take MINMAXDIST metric into account when developing NN searching algorithms for the SH-tree.