APPEARED IN THE PROCEEDINGS OF THE 1999 IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, MARCH 1999

The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces *

Kaushik Chakrabarti
Department of Computer Science
University of Illinois at Urbana-Champaign
kaushikc@cs.uiuc.edu

Sharad Mehrotra
Department of Information and Computer Science
University of California at Irvine
sharad@ics.uci.edu

Abstract

Feature based similarity search is emerging as an important search paradigm in database systems. The technique used is to map the data items as points into a high dimensional feature space which is indexed using a multidimensional data structure. Similarity search then corresponds to a range search over the data structure. Although several data structures have been proposed for feature indexing, none of them is known to scale beyond 10-15 dimensional spaces. This paper introduces the hybrid tree – a multidimensional data structure for indexing high dimensional feature spaces. Unlike other multidimensional data structures, the hybrid tree cannot be classified as either a pure data partitioning (DP) index structure (e.g., R-tree, SS-tree, SR-tree) or a pure space partitioning (SP) one (e.g., KDB-tree, hB-tree); rather, it "combines" positive aspects of the two types of index structures a single data structure to achieve search performance more scalable to high dimensionalities than either of the above techniques (hence, the name "hybrid"). Furthermore, unlike many data structures (e.g., distance based index structures like SS-tree, SR-tree), the hybrid tree can support queries based on arbitrary distance functions. Our experiments on "real" high dimensional large size feature databases demonstrate that the hybrid tree scales well to high dimensionality and large database sizes. It significantly outperforms both purely DP-based and SP-based index mechanisms as well as linear scan at all dimensionalities for large sized databases.

1. Introduction

Feature based similarity search is emerging as an important search paradigm in database systems. The technique used is to map the data items as points into a high dimensional feature space. The feature space is usually indexed using a multidimensional data structure. Similarity search then corresponds to a range search on that data structure. To support efficient similarity search in a database system, robust techniques to index high dimensional feature spaces needs to be developed. Traditional multidimensional data structures (e.g., R-trees [11], kDB-trees [20], grid files [17]), which were designed for indexing spatial data, are not suitable for multimedia feature indexing due to (1) inability to scale to high dimensionality and (2) lack of support for queries based on arbitrary distance measures. Recently, there has been significant research effort in developing indexing mechanisms suitable for multimedia feature spaces. One of the techniques is *dimensionality reduction* (DR). Standalone DR techniques have several limitations: (1) they work well only when the data is strongly correlated (2) they usually do not support similarity queries based on arbitrary distance functions [2] and (3) they are not suitable for dynamic database environments. While the DR approach has merit and should be used whenever it is possible to use (e.g., correlated data, fixed distance function, more or less static datasets), a robust solution to feature indexing requires multidimensional data structures that scale to high dimensionality and supports arbitrary distance measures.

This paper introduces the hybrid tree for this purpose. What distinguishes the hybrid tree from other multidimensional data structures is that it is *neither a pure DP-based nor a pure SP-based technique*. Experience has shown that neither of these techniques are suitable for high dimensionalities but for different reasons. Simple sequential scan performs better beyond 10-15 dimensions [5]. BR-based techniques tend to have low fanout and a high degree of overlap between bounding regions (BRs) at high dimensions. On the other hand, SP-based techniques have fanout independent of dimensionality and no overlap between subspaces. But SP-based techniques suffer from problems like no guaranteed utilization (e.g., kDB-trees) or require storage of redundant information (e.g., hB-trees). The main contribution of this paper is the *"hybrid"* approach to multidimensional indexing: a technique that combines positive aspects of the two types of index structures a single data structure to achieve search performance more scalable to high dimensionalities than either of the two techniques. On one hand, like SP-based index structures, the hybrid tree performs node splitting based on a single dimension and represents space partitioning using kd-trees. This makes the fanout independent of dimensionality and enables fast intranode search. On the other hand, space partitions, like the BRs in DP-based techniques, are allowed to overlap whenever clean splits necessicate downward cascading splits, thus retaining the guaranteed utilization property. The tree construction algorithms in the hybrid tree are geared towards providing optimal search performance. As desired, the hybrid tree allows search based on arbitrary distance functions. The distance function can be specified by the user at query time. Our experiments on "real" high dimensional large size feature databases show that the hybrid tree scales well to high dimensionality and large database sizes. It significantly outperforms both purely DP-based and SP-based index mechanisms as well as linear scan at all dimension-

*This work was supported in part by the National Science Foundation under Grant No. IIS-9734300, in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003 and in part by NSF/DARPA/NASA Digital Library Initiative Program under Cooperative Agreement No. 94-11318.

Index Structure	Number of dimensions used to split	Number of (k-1)-d hyperplanes used to split	Number of kd-tree nodes used to represent the split	Fanout	Degree of Overlap	Node Utilization Guarantee	Storage Redundancy
KDB-tree	1	1	1	High (Independent of k)	None	No	None
hB-tree	d ($1 \leq d \leq k$)	d	d	High (Independent of k)	None	Yes	Yes
R-tree	k	2k	-	Low for large k ($\propto \frac{1}{k}$)	High	Yes	None
Hybrid tree	1	1 or 2	1	High (Independent of k)	Low	Yes	None

Table 1. Splitting strategies for various index structures. k is the total number of dimensions.

alities for large sized databases.

The rest of the paper is organized as follows. Recently, many multidimensional data structures have been developed for the purpose of high dimensional feature indexing. In Section 2, we develop a classification of these data structures that allows us to compare them to the hybrid tree. Section 3 introduces the hybrid tree and is the main contribution of this paper. In Section 4, we present the performance results. Section 5 offers the final concluding remarks and future work.

2. Classification of Multidimensional Index Structures

The increasing need of applications to be able to store multidimensional objects (e.g., features) in a database and index them based on their content has trigerred a lot of research on multidimensional index structures. In this section, we develop a classification of multidimensional indexing techniques which allows us to compare the hybrid tree with the previous research in this area.

Existing multidimensional techniques can be classified in two different ways. One way to classify them is into **Data Partitioning (DP)-based and Space Partitioning (SP)-based** index structures. A DP-based index structure consists of bounding regions (BRs) arranged in a (spatial) containment hierarchy. At the data level, the nearby data items are clustered within BRs. At the higher levels, nearby BRs are recursively clustered within bigger BRs, thus forming a hierarchical directory structure. The BRs may overlap with each other. The BRs can be bounding boxes (e.g., R-tree[11], X-tree[4]) or bounding spheres/diamonds (e.g., SS-tree[23], M-tree[9], TV-tree[15]). On the other hand, a SP-based index structure consists of space recursively partitioned into mutually disjoint subspaces. The hierarchy of partitions form the tree structure (e.g., kDB-tree[20], hB-tree[16] and LSDh-tree[12]). We compare these two types of index structures with the hybrid tree as a solution to high dimensional feature indexing in Section 3.6.

An alternative way of classification is into **Feature-based and Distance based** techniques. In feature based techniques, the data/space partitioning is based on the values of the vectors along each independent dimension and is independent of the distance function used to compute the distance among objects in the database or between query objects and database objects. Examples of DP-based techniques that are feature based include R-tree and X-tree. Examples of SP-based techniques that are feature based include kDB-tree, hB-tree, LSDh-tree. On the other hand, distance based techniques partition data/space based on the distance of objects from one or more selected pivot point(s), where the distance is computed using a given distance function. Examples of DP-based techniques that are distance based include SS-tree, M-tree and TV-tree. Examples of SP-based techniques that are distance based include vp-tree [8] and mvp-tree [6]. A comparison between the two classes can be found in [7].

3. The Hybrid Tree

In this section, we introduce the hybrid tree. We discuss how the hybrid tree partitions the space into subspaces and how the space partitioning is represented in the hybrid tree. We discuss the node splitting algorithms and show how they optimize expected search performance. We describe the tree operations and conclude with a discussion on where the hybrid tree fits into the classification developed in Section 2.

3.1. Space Partitioning in the Hybrid Tree

First, we describe the "space partitioning strategy" in the hybrid tree i.e. how to partition the space into two subspaces when a node splits. The first issue is the number of dimensions used to partition the node. The hybrid tree always splits a node using a *single* dimension. 1-d split is the *only* way to guarantee that the fanout is totally independent of dimensionality. This is in sharp contrast with DP-based techniques which are at the other extreme: they use all the k dimensions to split, leading to a linear decrease in fanout with increase in dimensionality. Some index structures follow intermediate policies [16]. The only disk-based index structure that follows a 1-d split policy is the kDB-tree [20]. Single dimension splits in the kDB-tree necessitate costly cascading splits and causes creation of empty nodes. Due to the above reasons, kDB-tree shows poor performance even in 4 dimensional feature spaces [10]. kDB-trees cause cascading splits since it requires the node splits to be necessarily *clean* i.e. the split *must* divide the indexed space into two mutually disjoint partitions. We relax the above constraint in the hybrid tree: the indexed subspaces need *not* be mutually disjoint. The overlap is allowed only when trying to achieve an overlap-free would cause downward cascading splits and hence a possible violation of utilization constraints. The splitting strategies of the various index structures is summarized in the Table 1.

It is clear from the above discussion that the hybrid tree is more similar to SP-based data structures than DP-based index structures. But the above "relaxation" necessicates several changes in terms of representation and algorithms for tree operations as compared to the pure SP-based index structures. The first change is in the representation. As in other SP-based techniques, the space partitioning within each index node in a hybrid tree is represented using a kd-tree. Since regular kd-trees can represent only overlap free splits, we need to modify the kd-tree in order to represent possibly overlapping splits. Each internal node of the regular kd-tree represents a split by storing the split dimension and the split position. We add a second split position field to the kd-tree internal node. The first split position represents the right (higher side) boundary of the left (lower side) partition (denoted by lsp or left side partition) while the second split position represents the left boundary of the right partition (denoted by rsp or right side partition). While $lsp = rsp$ means non-overlapping partitions, $lsp > rsp$ indicate overlapping partitions. The second change is in the algorithms for regular tree operations, namely, search, insertion and deletion. The tree operations in SP-based index structures are based on the assumption that the partitions are mutually disjoint. This is not true for the hybrid tree. We solve the problem by treating the indexed subspaces as BRs in a DP-based data structure (which can overlap). In other words, we define a *mapping* the kd-tree based representation to an "array of BRs" representation. This allows us to directly apply the search, insertion and deletion algorithms used in DP-based data structures to the hybrid tree. The mapping is defined recursively as follows: *Given any index node N of the hybrid tree and the BR R_N corresponding to it, we define the BRs corresponding to each child of N.* The BR of the root node of the hybrid tree is the entire data space. Given that, the above "mapping" can compute the BR of any hybrid tree node.

Let N be an index node of the hybrid tree. Let K_N be the kd-tree that represents the space partitioning within N and R_N be the BR of N. We define a BR associated with each node (both internal as well as leaf nodes) of K_N. This defines the BRs of the children of N since the leaf nodes of K_N are the children of N. For example, the leaf nodes $L1$ to $L7$ are the children of the hybrid tree node N shown in the Figure 1. The BR associated with the root of K_N is R_N. Now given an internal node I of K_N and the corresponding BR R_I, the BRs of the two children of I are defined as follows. Let $I = \langle dim, lsp, rsp \rangle$, where dim, lsp and rsp are the split dimension, left split position and right split position respectively. The BR of the left child of I is defined as $R_I \cap (dim \leq lsp)$ where, in the expression $(dim \leq lsp)$, dim denotes the variable that represents the value along dimension dim (for simplicity) and $\cap$ represents geometric intersection. Similarly, the BR of the right child of I is defined as $R_I \cap (dim \geq rsp)$. For example, $(0, 0, 6, 6)$ is the BR for the hybrid tree node shown in Figure 1 (BR is denoted as $x_{lo}, y_{lo}, x_{hi}, y_{hi}$). The BR of I1 (the root) is $(0, 0, 6, 6)$. The BRs of I2 and I3 are $(0, 0, 6, 6) \cap (x \leq 3) = (0, 0, 3, 6)$ and $(0, 0, 6, 6) \cap (x \geq 3) = (3, 0, 6, 6)$ respectively. Similarly, the BR of L3, which, being a leaf of K_N, is a child of N, is obtained by $BR(I2) \cap (y \geq 2)$ i.e. $(0, 0, 3, 6) \cap (y \geq 2) = (0, 2, 3, 6)$. The children of internal nodes with $lsp > rsp$ have overlapping BRs (e.g., BRs of I4 and L3 (children of I2) overlap). Figure 1 shows all the BRs – the shaded rectangles are the BRs of the children of the node while the white ones correspond to the internal nodes of K_N.

Note that the above mapping is "logical". The search/insert/delete algorithm does not actually compute the "array of BRs" during tree traversal: rather it navigates the node using the kd-tree and computes the BR only when necessary (cf. Section 3.4). The kd-tree based navigation allows faster intranode search compared to array-based navigation. While searching for a correct lower level node using a kd-tree usually requires order $\log n$ comparisons (for a balanced kd-tree), searching in a array requires linear number of comparisons. Also, in a kd-tree representation, BRs share boundaries. In an array representation, the boundaries are checked redundantly while in a kd-tree, a boundary is checked only once [16].

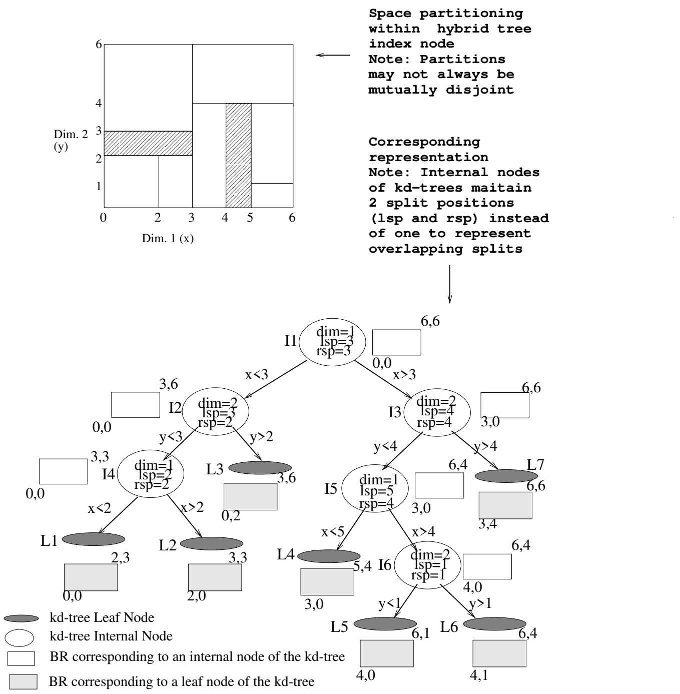


Figure 1. Mapping between each node and the corresponding BR. The shaded area represents overlap between BRs

3.2. Data Node Splitting

The choice of a split of a node consists of two parts: the choice of the split dimension and the split position(s). In this section, we discuss the choice of splits for data nodes in the hybrid tree.

Choice of split dimension: When a data node splits, it is replaced by two nodes. Assuming that the rest of the tree has not changed, the expected number of disk accesses per query (EDA) would increase due to the split. The hybrid tree chooses as the split dimension the one that minimizes the increase in EDA due to the split, thereby optimizing the expected search performance for future queries.

Let N be the data node being split. Let R be the k-dimensional BR associated with N. Let s_i be the extent of R along the ith dimension, $i = [1, k]$. Consider a bounding box range query Q with each side of length r. We assume that the feature space is normalized (extent is from 0 to 1 along each dimension) and the queries are uniformly distributed in the data space. Let $P_{overlap(Q,R)}$ denote the probability that Q overlaps with R. To determine $P_{overlap(Q,R)}$, we move the center point of the query to each point of the data space marking the positions where the query rectangle intersects the BR. The resulting set of marked positions is called the Minkowski Sum which is the original BR having all sides extended by query side length r [1]. Therefore, $P_{overlap(Q,R)} = (s_1+r)(s_2+r)...(s_k+r)$. This is the probability that Q needs to access node N (1 disk access)

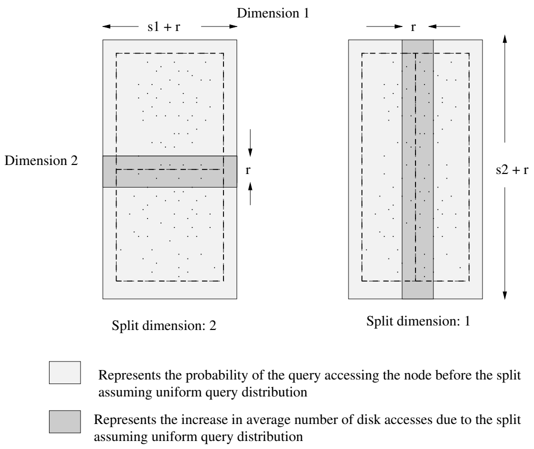


Figure 2. Choice of split dimension for data nodes. The first split is the optimal choice in terms for search performance.

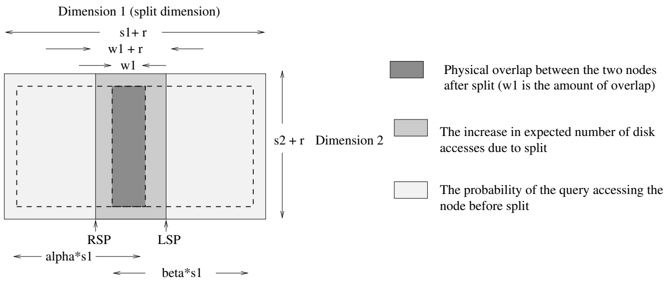


Figure 3. Index node splitting (with overlap). s_j, w_j and split positions (LSP and RSP) only along dimension 1 are shown.

(It is the volume of lightly shaded region in Figure 2).

Now let us consider the splitting of N and let j be the splitting dimension. Let $N1$ and $N2$ be the nodes after the split and $R1$ and $R2$ be the corresponding BRs. $R1$ and $R2$ have the same extent as R along all dimensions except j i.e. s_i, $i = [1, k], i \neq j$. Let αs_j and βs_j be the extents of $R1$ and $R2$ along the jth dimension. Since the split is overlap-free, $\beta = 1 - \alpha$. The probabilities $P_{overlap(Q,R1)}$ and $P_{overlap(Q,R2)}$ are $(s_1 + r)...(\alpha s_j + r)...(s_k + r)$ and $(s_1 + r)...((1 - \alpha)s_j + r)...(s_k + r)$ respectively. Since $R = R1 \cup R2$ (where $\cup$ is the geometric union) and Q is uniformly distributed, $P_{overlap(Q,R)} = P_{overlap(Q,R1 \cup R2)} = P_{overlap(Q,R1) \cup overlap(Q,R2)}$. Thus, the probability $P_{overlap(Q,R1) \cap overlap(Q,R2)}$ that both $N1$ and $N2$ are accessed is equal to $P_{overlap(Q,R1)} + P_{overlap(Q,R2)} - P_{overlap(Q,R)}$. ($P_{overlap(Q,R1) \cap overlap(Q,R2)}$ is equal to the volume of the dark shaded region in Figure 2). If Q does not overlap with R, there is no increase in number of disk accesses due to the split. If it does, $P_{overlap(Q,R1) \cap overlap(Q,R2)}$ is the probability that the disk accesses increases by 1 due to the split. Thus, the conditional probability that Q overlaps with both $R1$ and $R2$ given Q overlaps with R, i.e. $\frac{P_{overlap(Q,R1) \cap overlap(Q,R2)}}{P_{overlap(Q,R)}}$ represents the increase in EDA due to the split. The increase in EDA if j is chosen as the split dimension evaluates out to be $\frac{r}{s_j + r}$. Note that $\frac{r}{s_j + r}$ is minimum if j is chosen such that $s_j = max_{i=1}^{k} s_i$, independent of the value of r. The hybrid tree always chooses the dimension along with the BR has the largest extent as the split dimension for splitting data nodes so as to minimize the increase in EDA due to the split.

An example of the choice of split dimension is shown in Figure 2. Note that the optimality of the above choice is independent of the distribution of data. It is also independent of the choice of split position. Previous proposals regarding choice of splitting dimensions include arbitrary/round-robin [12] and maximum variance dimension [24]. The maximum variance dimension is chosen to make the choice insensitive to "outliers" [24]. Since the number of disk accesses to be made depends on the size of the subspaces indexed by data nodes and is independent of the actual distribution of data items within the subspace, presence or absence of "outliers" is inconsequential to the query performance. We performed experiments to compare our choice of maximum extent dimension as the splitting dimension with the maximum variance choice and is discussed is Section 5.

Choice of split position: The most common choice of the split position for data node splitting is the median [20, 16, 24]. The median choice, in general, distributes the data items equally among the two nodes (assuming unique median). The hybrid tree, however, chooses the split position as close to the middle as possible. [1] This tends to produce more cubic BRs and hence ones with smaller surface areas. The smaller the surface area, the lower the probability that a range query overlaps with that BR, the lower the number of expected number of disk accesses [3]. Our experiments validate the above observation.

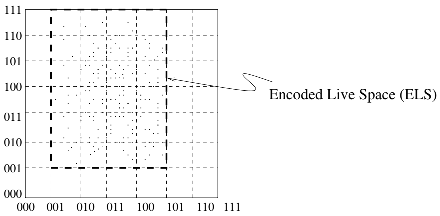
3.3. Index Node Splitting

In this section, we discuss the choice of split dimension and split position for index nodes.

Choice of the split dimension: Like data node splitting, the choice of split dimension for index nodes splitting is also based on minimization of the increase in EDA. However, unlike data node splitting where the choice is independent of the query size, the choice of the split dimension for index nodes depends on the probability distribution of the query size as discussed below.

The main difference here compared to data node splitting is splits are not always overlap free. Let w_j ($w_j \leq s_j$) be the amount of overlap between $R1$ and $R2$ along the jth dimension (how w_j is computed is discussed in the following paragraph on choice of split position). So $\alpha s_j + \beta s_j = s_j + w_j$. An example of an index node split is shown in Figure 3. The probabilities $P_{overlap(Q,R1)}$ and $P_{overlap(Q,R2)}$ are $(s_1 + r)...(\alpha s_j + r)...(s_k + r)$ and $(s_1 + r)...(\beta s_j + r)...(s_k + r)$ respectively. Proceeding in the same way as before, the increase in EDA if j is chosen as the split dimension evaluates out to be $\frac{w_j + r}{s_j + r}$. The choice of j that minimizes the above quantity optimizes search performance. But the choice depends on r and can differ for different values of r. For a given probability distribution of r, the hybrid tree chooses the dimension that minimizes the increase in EDA averaged over all queries. Let $P(r)$ be probability distribution of r. The increase in EDA averaged over all queries

[1] To find the position, we first check whether it is possible to split in the middle without violating utilization constraint. If yes, it is chosen. Otherwise the split position is shifted from the middle position in the proper direction just enough to satisfy the utilization requirement.



Live space encoding using 3 bit precision (ELSPRECISION=3)
Encoded Live Space BR = (001, 001, 101, 111)
Bit required: 2*number_of_dimensions*ELSPRECISION=12 bits

Figure 4. Encoded Live Space (ELS) Optimization

is equal to $\int_R^{R+\Delta R} P(r).\frac{w_j+r}{s_j+r}dr$ where r can vary from R to $R+\Delta R$. The dimension that minimizes the above quantity is chosen as the split dimension. For example, for uniform distribution, where $P(r) = \frac{1}{\Delta R}$, the above integral evaluates to be $\left(1 - (\frac{s_j - w_j}{\Delta R})log(1 + \frac{\Delta R}{s_j+R})\right)$. In this case, the hybrid tree chooses that j for which $(s_j - w_j)log(1 + \frac{\Delta R}{s_j+R})$ is maximum. In our experiments, we use all queries of the same size, say R. In this case, the dimension j that minimizes $\frac{w_j+R}{s_j+R}$ should be chosen as the split dimension which is indeed the case since $lim_{\Delta R \to 0}\left(1 - (\frac{s_j - w_j}{\Delta R})log(1 + \frac{\Delta R}{s_j+R})\right) = \frac{w_j+R}{s_j+R}$.

Choice of split position: Given the split dimension, the split positions are chosen such that the overlap is minimized without violating the utilization requirement. The problem of determining the best split positions along a given dimension is a 1-d version of the R-tree bipartitioning problem. In the latter, the problem is to equally divide the rectangles into two groups to reduce the total area covered by the bounding boxes. while in the former, the problem is to divide the line segments (indexed subspaces of the children projected along the split dimension) into two groups in a way to minimize the the overlap along the split dimension without violating the utilization constraint. We sort the line segments based on both their left (leftmost to rightmost) and right (rightmost to leftmost) boundaries. Then we choose new segments alternately from the left and right sorted lists and place them in left and right partitions respectively till the utilization is achieved. The remaining line segments are put in the partition that needs least elongation without caring about utilization. The above bipartitioning algorithm is similar to the R-tree quadratic algorithm but runs in $O(nlogn)$ time instead of $O(n^2)$ (where n is the number of children nodes) since 1-d intervals can be sorted based on their values (left and right boundaries) along the split dimension.

Before the split dimension is actually chosen, the best split positions are determined for all the dimensions. Then the w_j's and s_j's are calculated for each dimension and the one with the lowest $\int_R^{R+\Delta R} P(r).\frac{w_j+r}{s_j+r}dr$ is selected. After the selection of the split dimension, the split positions for the selected dimension determined during the pre-selection phase are used as split positions.

Implicit Dimensionality Reduction:

We conclude the subsection on index node splitting with the following observation. The hybrid tree *implicitly* eliminates "non-discriminating" dimensions i.e. those dimensions along which the feature vectors are not much different from each other. In other words, these dimensions are never used for node splitting. This is true for data node splitting due to the "maximum extent" choice. To ensure that these dimensions are indeed eliminated, we must guarantee that an eliminated dimension is never chosen for splitting the index node. Let N be an index node. Let $\mathcal{D}_N$ be the set of dimensions used for partitioning space within N. We can provide the above guarantee if the the split dimension d_N of N satisfies $d_N \in \mathcal{D}_N$, The reason is that a dimension not used to split any data node cannot be in $\mathcal{D}_N$. Suppose we restrict our choice of the split dimension of N to $\mathcal{D}_N$ instead of all dimensions. We show that even then we would make the EDA-optimal choice.

Lemma 1 (Implicit Dimensionality Reduction) *It is possible to make the EDA-optimal choice even when restricting the choice of the split dimension of node N to $\mathcal{D}_N$.*

Proof:

The EDA-optimal choice of the split dimension of N is the one with the lowest $\frac{r+w_j}{r+s_j}$ ratio. We need to show that the above ratio for any dimension $j \in \mathcal{D}_N$ is less than or equal to the ratio for every dimension $i \notin \mathcal{D}_N$. For any dimension $j \in \mathcal{D}_N$, $w_j \leq s_j$. So for any $j \in \mathcal{D}_N$ and for any value of r, $\frac{r+w_j}{r+s_j} \leq 1$. For any dimension $i \notin \mathcal{D}_N$, $w_i = s_j$, hence $\frac{r+w_j}{r+s_j} = 1$ for all r (worst case). Hence the proof. ■

The hybrid tree achieves implicit dimension elimination through the above choice. This effect is not seen in most paginated multidimensional data structures. For example, DP-based techniques, all dimensions are used for indexing - so nothing is eliminated. SP-based techniques which choose the split dimension arbitrarily/round robin fashion cannot provide the above guarantee.

3.4. Dead Space Elimination

The hybrid tree, like other SP techniques, indexes dead space i.e. space the contains no data objects. DP-techniques, on other other hand, does not. Dead space indexing cause unnecessary disk accesses. This effect increases at higher dimensionality. Storage of the live space BRs would reduce the hybrid tree into a DP-based technique, making the fanout of the node sensitive to dimensionality. Instead, we encode the live space BR *relative* to the entire BR (defined by kd-tree partitioning) using a few bits as suggested in [12]. The live space encoding is explained in Figure 4. More the number of bits used, the higher the precision of the representation, lower the number of unnecessary disk accesses. We observed that using as few as 4 bits per dimension eliminates most dead space. For 8K page, 4 bit precision and 64-d space, the overhead is less than 1% of the database size and can be stored in memory. The overhead is even less for lower dimensionality. During search (say range search), the overlap check is performed in 2 steps: first, the BR defined by kd-tree is checked and if they overlap, the live space BR is decoded and checked, thus saving any unnecessary decoding/checking costs. We performed experiments to demonstrate the effect of ELS optimization in the hybrid tree as discussed in Section 5.

3.5. Tree Operations

The hybrid tree, like other disk based index structures (e.g., B-tree, R-tree) is completely dynamic i.e. insertions, deletions and updates can occur interspersed with search queries without

Property of index structure	BR-based index structures	kd-tree based index structures	Hybrid Tree
Representation of space partitioning	Array of bounding boxes	kd-tree	kd-tree (modified to represent overlapping partitions)
Indexed subspaces	May mutually overlap	Strictly disjoint	May mutually overlap
Node splitting	Using all dimensions	Using 1 or more dimensions	Using 1 dimension
Dead space † elimination	Yes	No	Yes (with live space encoding)

Table 2. Comparison of the hybrid tree with the BR-based and kd-tree based index structures. † Dead space refers to portions of feature space containing no data items (cf. Section 4.2).

requiring any reorganization. The tree operations in the hybrid tree are similar to the R-trees i.e. indexed subspaces are treated as BRs but the kd-tree based organization is exploited to achieve faster intranode search. In addition to point and bounding-box queries (i.e. feature-based queries), the hybrid tree supports distance-based queries: both range and nearest neighbor queries. Unlike several index structures (e.g., distance-based index structures like SS-tree, M-tree), the hybrid tree, being a feature-based technique, can support queries with arbitrary distance measures. This is important advantage since the distance function can vary from query to query for the same feature or even between several iterations of the same query in a relevance feedback environment [13, 21].

The insertion and deletion operations in the hybrid tree is also similar to that in R-trees. The insertion algorithm recursively picks the child node in which the new object should be inserted. The best candidate is the node that needs the minimum enlargement to accomodate the new object. Ties are broken based on the size of the BR. The deletion operation is based on the eliminate-and-reinsert policy as in [11].

3.6. Summary

It is clear from the above discussion that the hybrid tree resembles both DP and SP techniques in some aspects and differs from them in others: rather it is a "hybrid" of the two approaches. The comparison of the hybrid tree with the two techniques is shown in Table 2. Now we summarize the reasons why hybrid tree is more suitable for high dimensional indexing either DP or SP techniques. It is more suitable than than pure DP techniques since (1) its fanout is independent of dimensionality while DP-techniques have low fanout at high dimensionalities (2) enables faster intranode search by organizing the space partitioning as a kd-tree instead of an array and (3) eliminates overlap from the lowest level (since data node splits are always mutually non-overlapping) and reduces overlap at higher levels by using EDA-optimal 1-d splits instead of k-d splits as in DP techniques. The hybrid tree performs better than other SP-based techniques using 1-d splits (e.g., KDB-trees) since unlike the latter, it provides (1) guaranteed storage utilization (2) avoids costly cascading splits and (3) chooses EDA-optimal split dimensions instead of arbitrarily. It performs better than SP-based techniques using multiple dimensional splits (e.g., hB-trees) since (1) 1-d splits usually provide better search performance compared to multiple dimensional ones since the latter tends to produce subspaces with larger surface area and hence more disk accesses [3] and (2) it does not require storage of redundant information (e.g., posting full paths).

4. Experimental Evaluation

We performed extensive experimentation to (1) evaluate the various design decisions made in the hybrid tree and (2) compare the hybrid tree with other competitive techniques. We conducted our experiments over the following two "real world" datasets:

(1) The **FOURIER** dataset contains 1.2 million 16-d vectors produced by fourier transformation of polygons. We construct 8-d, 12-d and 16-d vectors by taking the first 8, 12 and 16 fourier coefficients respectively.

(2) The **COLHIST** dataset comprises of color histograms extracted from about 70,000 color images obtained from the Corel Database. We generate 16, 32 and 64 dimensional vectors by extracting 4x4, 8x4 and 8x8 color histograms [18] from the images.

The queries are randomly distributed in the data space with appropriately chosen ranges to get constant selectivity. In all experiments discussed below, the selectivity is maintained constant at 0.07 % for FOURIER and 0.2 % for COLHIST. All the experiments were conducted on a Sun Ultra Enterprise 3000 with 512MB of physical memory and several GB of secondary storage. In all our experiments, we use a page size of 4096 bytes.

We performed experiments to evaluate (1) the impact of EDA-optimal node splitting algorithms and (2) the effect of live space optimization in the hybrid tree. Both the experiments were performed on the 64-d COLHIST data. The performance is measured by (1) the average number of disk accesses required to execute a query and (2) the average CPU time required to execute a query. Figure 5(a) and (b) show the performance of the hybrid tree constructed using EDA-optimal node splitting algorithms compared to the hybrid tree constructed using the VAMSplit node splitting algorithm [24]. The EDA-optimal split algorithms consistently outperforms the VAMSplit algorithm. The performance gap increases with the increase in dimensionality. Figure 5(c) shows the effect of live space optimization. Using 4-bit ELS improves the performance significantly compared to no ELS but using more bits does not improve it much further.

We conducted experiments to compare the performance of the hybrid tree with the following competitive techniques: (1) SR-tree [14] (2) hB-tree [16] (3) Sequential Scan. We chose SR-tree since it is one of the most competitive BR-based data structures proposed for high dimensional indexing. Similarly, hB-tree is among the best known SP-based techniques for high dimensionalities. We normalize the I/O cost and the CPU cost of each of the 3 indexing techniques against the cost of linear scan. We define the normalized costs as follows:

- *The Normalized I/O cost*: the ratio of the average number of disk accesses required to execute a query using the

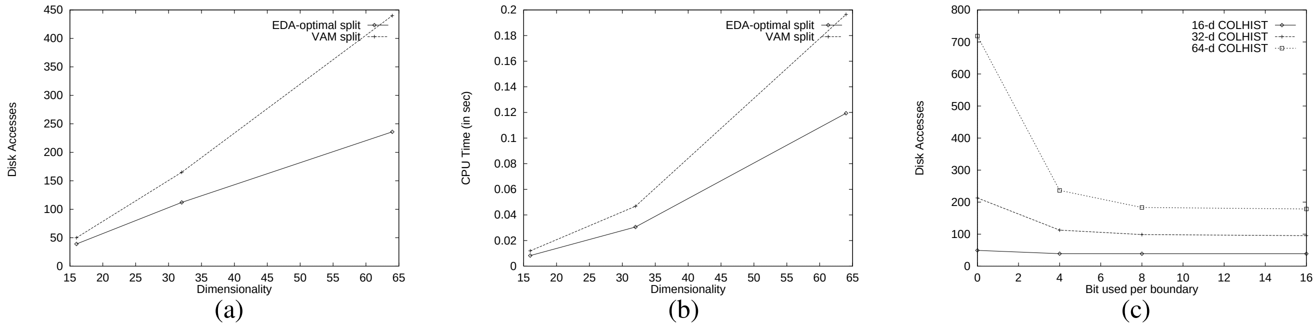


Figure 5. (a) and (b) shows the effect of EDA Optimization on query performance. (c) shows the effect of ELS Optimization on query performance. Both experiments were performed on 64-d COLHIST data.

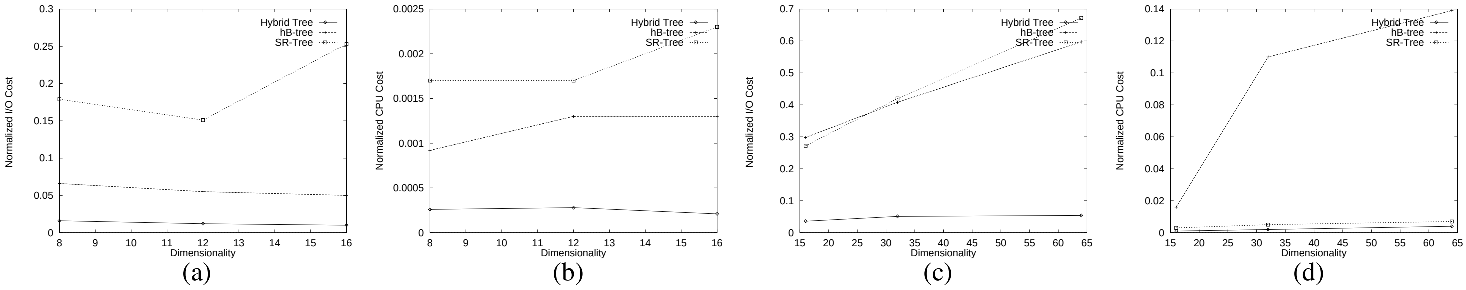


Figure 6. Scalability to dimensionality. (a) and (b) shows the query performance (I/O and CPU costs) for medium dimensional data (FOURIER dataset(400K points)). (c) and (d) shows the same for high dimensional data (COLHIST dataset(70K points))

indexing technique to the number of disk accesses to execute a linear scan. The latter is computed by $\frac{DatabaseSize}{PageSize}$ i.e. $\frac{NumberOfObjects*Dimensionality*sizeof(float)}{PageSize}$. Note that since sequential disk accesses are about 10 times faster compared to random accesses, the normalized I/O cost of linear scan is 0.1 instead of 1.0. Hence, for any index mechanism, a normalized I/O cost of more than 0.1 indicate worse I/O performance compared to linear scan.

- *The Normalized CPU cost*: the ratio of average CPU time required to execute a query using the index mechanism to the average CPU time required to perform a linear scan. The normalized CPU cost of linear scan is 1.0.

Using normalized costs instead of direct costs (1) allows us to compare each of the techniques against linear scan as the latter is widely recognized as a competitive search technique in high dimensional feature spaces [5] while still comparing them to each other and (2) makes the measurements independent of the experimental settings (e.g., H/W platform, pagesize).

Figures 6 shows the scalability of the various techniques to medium dimensional and high dimensional feature spaces respectively. The hybrid tree performs significantly better than any other technique including linear scan. The hB-tree performs better compared to SR-tree since SP-based techniques are more suited for high dimensional indexing than BR-technques as argued in [22]. The fast intranode search in the hybrid tree due to its kd-tree based organization account for the faster CPU times.

Figures 7(a) and (b) compares the different techniques in terms of their scalability to very large databases. The hybrid tree significantly outperforms all other techniques by more than an order of magnitude for all database sizes. The hybrid tree shows a decreasing normalized cost with increase in database size indicating sublinear growth of the actual cost with database size. Figures 7(c) and (d) compares the query performance of various techniques [2] for distance-based queries. As suggested in [18], we use the L1 metric. Again, the hybrid tree outperforms the other techniques.

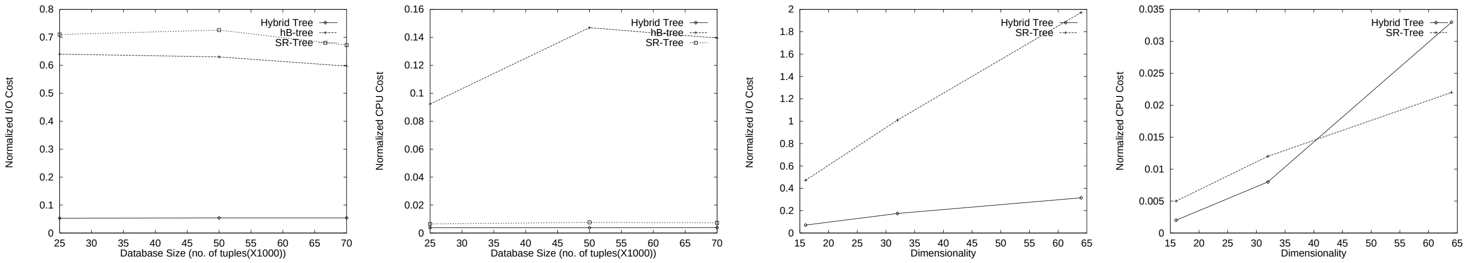
From the experiments, we can conclude that the hybrid tree scales well to high dimensional feature spaces, large database sizes and efficiently supports arbitrary distance measures.

5. Conclusion

Feature based similarity search is emerging as an important search paradigm in database systems. Efficient support of similarity search requires robust feature indexing techniques. In this paper, we introduce the hybrid tree - a multidimensional data structure for indexing high dimensional feature spaces. The hybrid tree combines positive aspects of bounding region based and space partitioning based data structures into a single data structure to achieve better scalability. It supports queries based on arbitrary distance functions. Our experiments show that the hybrid tree is scalable to high dimensional feature spaces and provides efficient support of distance based retrieval. The hybrid tree is a fully operational software and is currently being deployed for feature indexing in MARS [19].

As part of future work, we intend to support new types of queries like approximate nearest neighbor queries efficiently us-

[2] hB-tree is not used since it does not support distance-based search.



(a) (b) (c) (d)

Figure 7. (a) and (b) compares the scalability of the various techniques with database size of high dimensional data. (c) and (d) compares the query performance of the various techniques for distance-based queries (Manhattan Distance). Both experiments were performed on 64-d COLHIST data.

ing the hybrid tree. We also plan to explore techniques to support queries in interactive environments (e.g., relevance feedback [13, 21]) efficiently using the hybrid tree.

6. Acknowledgements

We thank Stefan Berchtold for providing us with the FOURIER dataset. We thank Corel Corporation for making the large collection of images used in the COLHIST dataset available to us. We obtained the hbπ-tree code from ftp://ftp.ccs.neu.edu/pub/people/georgio/code/. We implemented SR-trees by approriately modifying the R-tree implementation available from http://emerald.ucsc.edu/ tonig/rtrees/.

References

[1] S. Berchtold, C. Bohm, D. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high dimensional data spaces. *PODS*, 1997.

[2] S. Berchtold, C. Bohm, and H. P. Kriegel. The pyramid technique: Towards breaking the curse of dimensionality. *Proc. of ACM SIGMOD*, 1998.

[3] S. Berchtold and D. A. Keim. Indexing high-dimensional spaces: Database support for next decade's application. *SIGMOD Tutorial*, 1998.

[4] S. Berchtold, D. A. Keim, and H. P. Kriegel. The x-tree: An index structure for high-dimensional data. *Proc. of VLDB*, 1996.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Proc. of ICDT*, 1999.

[6] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high dimensional metric spaces. *Proc. of SIGMOD*, 1997.

[7] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for indexing high dimensional feature spaces. *Extended Version, Technical Report, MARS-TR-99-01, Available from http://luke.ics.uci.edu:8000/pages/publications*, 1998.

[8] T. Chiueh. Content-based image indexing. *Proc. of VLDB*, 1994.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proc. of VLDB*, 1997.

[10] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of ICDE, pages 606-615*, 1989.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf., pp. 47–57.*, 1984.

[12] A. Henrich. The lsdh-tree: An access structure for feature vectors. *Proceedings of ICDE*, 1998.

[13] Y. Ishikawa, R. Subramanya, and C. Faloutsos. Mindreader: Querying databases through multiple examples. *Proc. of VLDB*, 1998.

[14] N. Katayama and S. Satoh. The sr-tree: An index structure for high dimensional nearest neighbor queries. *Proc. of SIGMOD*, 1997.

[15] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree - an index stucture for high dimensional data. In *VLDB Journal*, 1994.

[16] D. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing mechanism with good guaraneed performance. *ACM Transactions on Database Systems*, 15(4), 1990.

[17] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems(TODS)*, 1984.

[18] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting similarity queries in mars. *Proc. of ACM Multimedia 1997*, 1997.

[19] M. Ortega, Y. Rui, K.Chakrabarti, S. Mehrotra, and T. Huang. Supporting ranked boolean similarity queries in mars. *Accepted for publication in IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1998.

[20] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD*, 1981.

[21] Y. Rui, T. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. *Proc. of IEEE Int. Conf. on Image Processing*, 1997.

[22] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. *Proc. of VLDB*, 1998.

[23] D. White and R. Jain. Similarity indexing with the ss-tree. *Proc. of ICDE*, 1995.

[24] D. White and R. Jain. Similarity indexing: Algorithms and performance. *Proc. of SPIE*, 1996.