The TV-tree – an index structure for high-dimensional data

King-Ip Lin
Department of Computer Science
Univ. of Maryland, College Park
kilin@cs.umd.edu

H.V. Jagadish
AT&T Bell Laboratories
Murray Hill, NJ
jag@research.att.com

Christos Faloutsos[*]
Department of Computer Science
and Inst. for Systems Research (ISR)
Univ. of Maryland, College Park
christos@cs.umd.edu

Abstract

We propose a file structure to index high-dimensionality data, typically, points in some feature space. The idea is to use only a few of the features, utilizing additional features whenever the additional discriminatory power is absolutely necessary. We present in detail the design of our tree structure and the associated algorithms that handle such 'varying length' feature vectors. Finally we report simulation results, comparing the proposed structure with the R^*-tree, which is one of the most successful methods for low-dimensionality spaces. The results illustrate the superiority of our method, with up to 80% savings in disk accesses.

Type of Contribution:
New Index Structure, for high-dimensionality feature spaces. Algorithms and performance measurements.
Keywords:
Spatial Index, Similarity Retrieval, Query by Content

1 Introduction

Many applications require enhanced indexing, capable of performing similarity searching on several, non-traditional ('exotic') data types. The target scenario is as follows: given a

[*]This research was partially funded by the National Science Foundation under Grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc.

collection of objects (eg., 2-d images, or 3-d medical brain scans, or simply English words), we would like to find objects similar to a given sample object. We rely on a domain expert to provide the appropriate similarity/distance functions between two objects. A list of potential applications for such a system follows:

- Image databases: In [19] we showed how to query for similar shapes, describing each shape by the co-ordinates of a few rectangles that cover it ($\approx$20 features per shape). In [26] we supported queries on color, shape and texture. For colors, we used the color histograms (64-256 attributes per image) as feature vectors; for shapes we used the first 20 moments.

- Medical databases, where 1-d objects (eg., ECGs), 2-d images (eg., X-rays) and 3-d images (eg., MRI brain scans) [4] are stored. Ability to retrieve quickly past cases with similar symptoms would be valuable for diagnosis, as well as for medical teaching and research purposes.

- Time series, such as financial databases with stock-price movements. The goal is to aid forecasting, by examining similar patterns that may have appeared in the past. In [1] we used as features the co-efficients of the Discrete Fourier Transform (DFT).

- Multimedia databases, with audio (voice, music), video etc. [25]. Users might want to retrieve, eg., similar music scores, or video clips.

- DNA databases where there is a large collection of strings from a four-letter alphabet (A,G,C,T); a new string has to be matched against the old strings, to find the best candidates. The BLAST algorithm [2] uses successive overlapping n-grams to index on. Regarding n-grams as features, we need 4^n features or 1,024 features for n=5.

- Searching for names or addresses, say in a customer (mailing) list, where these are partially specified or have errors. For example "1234 Springs Road" instead of "1235 Spring Rd", or "Mr. John Smith" instead of "Dr. J. Smith, Jr." Similar applications include spelling, typing [22] and OCR error correction [20]. There, given a wrong string, we should search a dictionary to find the closest strings to it. Triplets of letters are often used [3] to assess the similarity of two words, in which case we have at least $\approx 26^3 = 17{,}576$ features per word (assuming that words consist exclusively of the 26 English letters, ignoring digits, upper-case letters etc.).

For all of those applications, we rely on an expert to derive features that adequately describe the objects of interest. As we have proposed in [19], once objects are mapped into points in some feature space, we can accelerate searching by organizing these points in a spatial access method.

For a feature space with low dimensionality, any of the known spatial access methods will work. However, in the above applications, the number of features per object may be of the order of 10 or 100. The spatial access methods of the past have mainly concentrated on 2-dimensional and 3-dimensional spaces, such as the R-tree based methods [14], and the linear-quadtree based ones (eg., the z-ordering [28]). Although conceptually they can be extended to higher dimensionalities, they usually require time and/or space that grows exponentially with the dimensionality.

In this paper we propose a tree-structure that tries to avoid the dimensionality problem. The idea is to use a *variable* number of dimensions for indexing, adapting to the number of objects to be indexed, and to the level of the tree that we are at. Thus, for nodes that are close to the root, we use only a few dimensions (and therefore, we can store many 'branches', and enjoy a high fanout); as we descend the tree, we become more discriminating, utilizing more and more dimensions. Given that the feature vectors 'contract' and 'extend' dynamically, resembling a telescope, we called our method the *Telescopic-Vector tree*, or *TV-tree*.

The paper is organized as follows: Section 2 gives a survey of the related work, highlighting the problems of high-dimensionality. Section 3 presents the intuition and motivation behind the proposed method. Section 4 presents the details of the implementation of our method. Section 5 gives the experimental results. Section 6 lists the conclusions.

2 Survey

As mentioned above, for a variety of applications, feature extraction functions map objects into points in feature space; these points must be stored in a spatial access method. The prevailing methods form three classes: (a) R^*-trees [6] and the rest of the R-tree family [14, 18]; (b) linear quadtrees [31]; and (c) grid-files [27].

Different kinds of queries arise; the most typical ones are listed below:

- *Exact match* queries. Find if a given query object is in the database. For example, check if a certain inventory item exists in the database.

- *Range* queries. Given a query object, find all objects in the database that is within a certain distance from the object. Similarity queries also falls within this category. For example, find all buildings within 2 miles from the Washington National Airport;

find all words within a one-letter substitution from the word 'tex'; find all shapes that look like a Boeing 747.

- *Nearest neighbors* queries. Given a query item, find the item which is closest or most similar to the query item. For example, find the fingerprint that is most similar to the one given. Similarly, k-nearest neighbor queries can be asked.

- *'All pair'* queries. Given a set of objects, find all pairs within distance ϵ; or find the k closest pairs. For example, given a map, find all pairs of houses that are within 100 feet from each other.

- *Sub-pattern* matching. Instead of looking at the objects as a whole, we want to find sub-pattern within an object that matches our description. For example, we want to find stock movements that contain a certain pattern; or we want to find all x-ray images that contain tissue with tumor-like texture.

Previous work comparing the performance of different spatial data structures appeared in [13, 16]. [13] compared the R-tree, R+-tree and the K-D-B tree and the 2-D ISAM and it concluded that the R-tree and the R-+ tree give the better performance. In [16] the PMR-quadtree is compared to the R-tree variants for large line segment databases. Their results show that each data structure is suited for different kind of queries.

However most multidimensional indexing methods explode exponentially with the dimensionality, eventually reducing to sequential scanning. For linear quadtrees, the effort is proportional to the hypersurface bounding the query region [17]; the hypersurface grows exponentially with the dimensionality. Grid files face similar problems, since they require a directory that grows exponentially with the dimensionality. The R-tree and its variants will suffer if a single feature vector requires more storage space than a disk page can hold; in this case, the tree will have a fanout of '1', reducing to a linked list.

Similar problems with high dimensionality have been reported for methods that mainly focus on nearest-neighbor queries: Voronoi diagrams do not work at all for dimensionalities higher than 3 [5]. The method by Friedman et al. [10] does almost as much work as linear scanning for dimensionalities > 9. The 'spiral search' method by Bentley and Weide [7] also has a complexity that grows exponentially with the dimensionality.

Relevant to our work is a wide variety of clustering algorithms (see, for example, [15, 30, 24] for surveys). However, their main goals are to detect patterns in the data, and/or to assess the quality of the clustering scheme using the 'precision' and 'recall' measures;

there is usually little attention to measures like the space overhead and the time required to create, search and update the structure.

3 Intuition behind the proposed method

As mentioned above, several of the target applications require indexing in a high-dimensional feature space. Current spatial access methods suffer from the '*dimensionality curse*', typically exploding exponentially with the dimensionality.

The solution we propose is to 'contract' and 'extend' the feature vectors dynamically, that is, to use as few of the features as necessary to discriminate among the objects. This agrees with the intuitive way that humans use to classify objects: for example, in zoology, the species are grouped in a few broad classes first, using a few features (eg., vertebrates versus invertebrates). As the classification is further refined, more and more features are gradually utilized (eg., the feature of warm-blood versus cold-blood, for the vertebrates; similarly, the feature of lungs versus branchia etc.).

The basis of our proposed TV-tree is to use dynamically contracting and extending feature vectors. Like any other tree, it organizes the data in a hierarchical structure: Objects (ie., feature vectors) are clustered into leaf nodes of the tree, and the description of their *Minimum Bounding Region (MBR)* is stored in its parent node. Parent nodes are recursively grouped too, until the root is formed.

Compared to a tree that uses a fixed number of features, our tree provides a higher fanout at the top levels, using only a few, basic features, as opposed to many, possibly irrelevant, ones.

As more objects are being inserted into the tree, more features might be needed to discriminate among the objects. At that time, new features are introduced. The key point here is that features are introduced on a 'when needed' basis and thus we can soften the effect of the 'dimensionality curse'.

The basic telescopic vector concept can be applied to a tree with nodes describing bounding regions of any shape (cubes, spheres, rectangles etc.). Also, there is flexibility in the choice of the 'telescoping function', which selects the features on which to focus at any level of the tree. We discuss these design choices in the next two subsections.

3.1 Telescoping Function

In general, the telescoping problem can be described as follows:

Given an $n \times 1$ feature vector $\vec{x}$ and an $m \times n$ $(m \leq n)$ contraction matrix A_m, the $m \times 1$ vector $A_m \vec{x}$ is an m-contraction of $\vec{x}$. A sequence of such matrices A_m, with $m = 1, \ldots$ describes a telescoping function provided that the following condition is satisfied:
If the m_1-contractions of two vectors, $\vec{x}$ and $\vec{y}$, are equal, then so are their respective m_2-contractions, for every $m_2 \leq m_1$.

While a variety of telescoping functions can be defined (see Appendix B), the most 'natural' choice is simple truncation. That is, each matrix A_m has a 1 in positions (1,1) through (m,m), along a diagonal, and 0 everywhere else. In this paper we assume that truncation is the telescoping function selected.

The proposed method treats the features asymmetrically, favoring the first few ones over the rest, when truncation is used as the telescoping function. For similarity queries, which are likely to be frequent in the application domains we have in mind, it is intuitive that a good ordering of the features will result in a more focussed search. Even for exact match queries, since the depth of the tree will typically not be enough to have considered all features, a good choice of order will boost the response time of our method. Notice however that the *correctness* is not affected: The difference between a good ordering and a poor ordering is that the latter may make our method examine many 'false alarms', and thus do more work, but it will never create 'false dismissals'.

In most applications, it will be appropriate to 'transform' the given feature vector to achieve such a good ordering. Ordering the features on 'importance' order is exactly what the *Karhunen Loeve* (K-L) transform achieves: Given a set of n vectors with d features each, it returns d new features, which are linear combinations of the old ones, and which are sorted in discriminatory power. Figure 1 gives a 2-d example, where the vectors $k1$ and $k2$ are the results of the Karhunen Loeve transform on the illustrated set of points. For more details on the Karhunen Loeve transform, see [11].

The Karhunen Loeve transform is optimal if the set of data is known in advance, that is, the transform is data-dependent. Sets of data with rare or no updates appear in real applications: for example, databases that are published on CD-ROM; dictionaries; files with customer mailing lists that are updated in batches, etc. The Karhunen Loeve transform will also work well if a large sample of data is available in advance, and if the new data have the same statistical characteristics as the old ones.

In a completely dynamic case, we have to resort to data-independent transforms, such as the Discrete Cosine (DCT) [33] the Discrete Fourier (DFT), the Hadamard, the wavelet [29] transform etc. Fortunately, many data-independent transforms will perform as well as the Karhunen Loeve if the data follows specific statistical models. For example, the DCT is

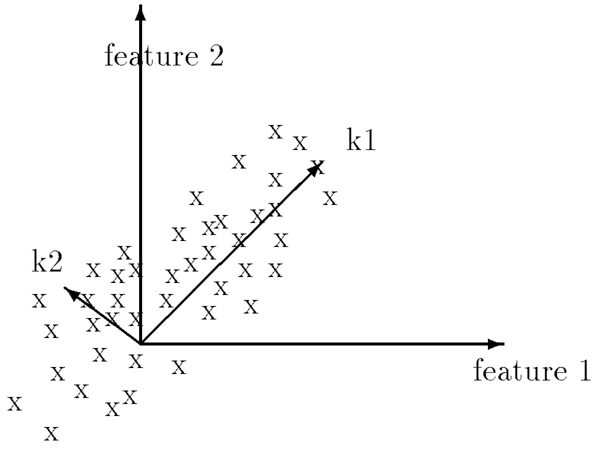


Figure 1: Illustration of the Karhunen Loeve transform

an excellent choice if the features are highly correlated. This is the case in 2-d images, where nearby pixels have very similar colors. The JPEG image compression standard [33] exactly exploits this phenomenon, effectively ignoring the high-frequency components of the Discrete Cosine Transform. Since the retained components carry most of the information, the JPEG standard achieves good compression with negligible loss of image quality.

We have observed similar behavior for the Discrete Fourier Transform in time series [1]. For example, *random walks* (also known as *brown noise* or *brownian walks*) exhibit a skewed spectrum, with the lower-frequency components being the strongest (and, therefore, most 'important' for indexing). Specifically, the amplitude spectrum is approximately $O(f^{-1})$, where f is the frequency). Stock movements and exchange rates have been successfully modeled as random walks (e.g., [9, 23]). Birkhoff's theory [32] claims that *'interesting'* signals, such as musical scores and other works of art, consist of *pink noise*, whose spectrum is similarly skewed ($O(f^{-0.5})$).

In general, if the statistical properties of the data are well understood, a data-independent transform in many common situations will obtain near optimal results, producing features sorted on 'importance' order. We should stress again that the use of a transform is *orthogonal* to the TV-tree: A suitable transform will just accelerate the retrieval.

3.2 Shape of Bounding Region

As mentioned earlier, points are grouped together, and their minimum bounding region (MBR) is stored in the parent node. The shape of the MBR can be chosen to fit the application; it may be a (hyper-)rectangle, cube, sphere etc. The shape that is simplest to represent is the sphere, requiring only the center and a radius. A sphere of radius r is the set of points with Euclidean distance $\leq r$ from the center of the sphere. Note that the Euclidean distance is a special case of the L_p metrics, with $p = 2$:

$$L_p(\vec{x}, \vec{y}) = [\sum_i (x_i - y_i)^p]^{1/p} \tag{1}$$

For the L_1 metric (or *Manhattan*, or *city-block* distance), the equivalent of a sphere is a diamond shape; for the L_∞ metric, the equivalent shape is a cube.

DEFINITION:

The *L_p-sphere* of center $\vec{c}$ and radius r is the set of points whose L_p distance from the center is $\leq r$.

The up-coming algorithms for the TV-tree will work with *any* L_p-sphere, without any modifications to the TV-tree manipulation algorithms. The only algorithm that depends on the chosen shape is the algorithm that computes the MBR of a set of data. For the 'diamond' shape, the algorithm is presented in Appendix A.

Minor modifications are required in the TV-tree algorithms to accommodate other popular shapes, like 'rectangles' or 'ellipses'. Compared to L_p-spheres, these shapes differ only in the fact that they have a different 'radius' for each dimension. The required changes in the TV-tree algorithms are in the decision-making steps, such as the criteria for choosing where to split, for choosing which branch to traverse during insertion and so on.

For the rest of this paper, we concentrate on L_p-spheres as Minimum Bounding Regions.

4 The TV-tree

4.1 Node Structure

Each node in the TV-tree represents the minimum bounding region (an L_p-sphere) of all its descendents. Each region is represented by a center, which is a vector determined by the telescoping vectors representing the objects, and a scalar radius. We will call the center of the region a 'telescopic' vector also (in the sense that it also contracts and extends depending on the objects stored within the region). We use the term *Telescopic Minimum Bounding Region* (TMBR) to denote an MBR with such a 'telescopic' vector as a center.

DEFINITION: A 'telescopic' L_p-sphere with center $\vec{c}$ and radius r, with *dimensionality* d and with α *active dimensions* contains the set of points $\vec{y}$ such that

$$c_i = y_i \quad i = 1, \ldots, d - \alpha \tag{2}$$

and

$$r^p \geq \sum_{i=d-\alpha+1}^{d} (c_i - y_i)^p \tag{3}$$

For example, in Figure 2 (a), D2 has 1 inactive dimension, the first one, and 1 active dimension, the second one. D1 also has one active dimension, the first one. The dimensionality of D1 is 1 (only the first dimension has been taken into account in specifying D1) and the dimensionality of D2 is 2 (both dimensions have been considered).

The reason we need this concept is that, as the tree grows, some leaf node will eventually consist of points that all agree on their first, say, k dimensions. In this case, the TMBR should exploit this fact; its first k dimensions are *inactive* dimensions, in the sense that these dimensions cannot distinguish between the node's descendents.

According to the algorithms of the TV-tree, *the active dimensions are always the last ones.* Moreover, we can control the number of active dimensions α and ensure that all the TMBRs in the tree have the same α. This number is a design parameter of the TV-tree.

DEFINITION: The *number of active dimensions* (α) of a TV-tree is the (common) number of active dimensions that all its TMBRs have.

The notation TV-1 denotes a TV-tree with α=1; Figure 2 shows the TMBRs of TV-1 and TV-2 trees. α determines the discriminatory power of the tree. Whenever more discriminatory power is needed, new dimensions will be introduced to ensure the number of active dimensions remains the same.

The data structure for an TMBR is as follows:

```
struct TMBR      { TVECTOR v;
                   integer radius;}
struct TVECTOR { list_of (float feature_value);
                   integer no_of_dimensions;}
```

where TVECTOR stands for 'telescopic vector'.

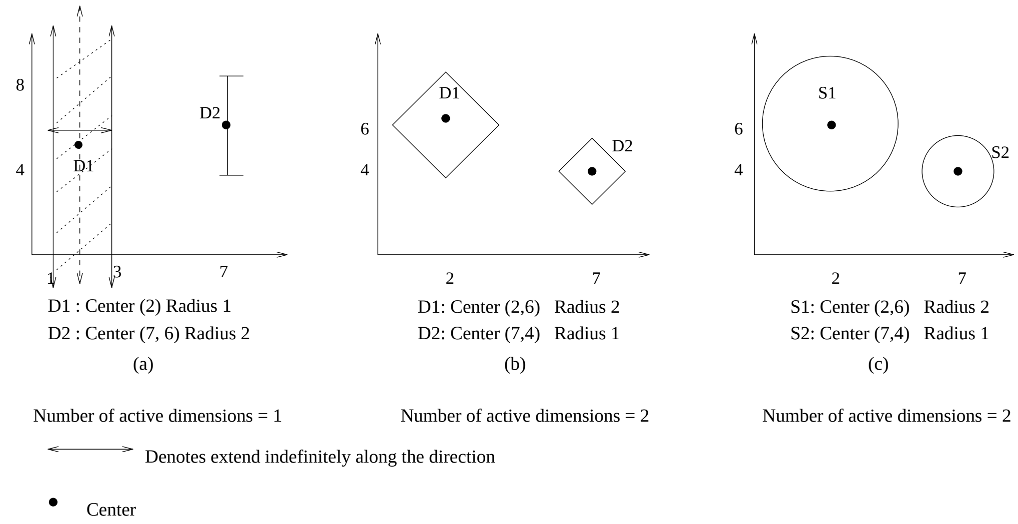


Figure 2: Example of TMBRs (diamonds and spheres), with different α

4.2 The tree structure

The TV-tree structure bears some similarity to the R-tree. Each node contains a set of branches; each branch is represented by a TMBR denoting the space it covers; all descendants of that branch will be contained within that TMBR; TMBRs are allowed to overlap. Each node occupies exactly one disk page.

Examples of TV-1 and TV-2 trees are given in Figures 3-4. A-I denote data points (only the first two dimensions are shown).

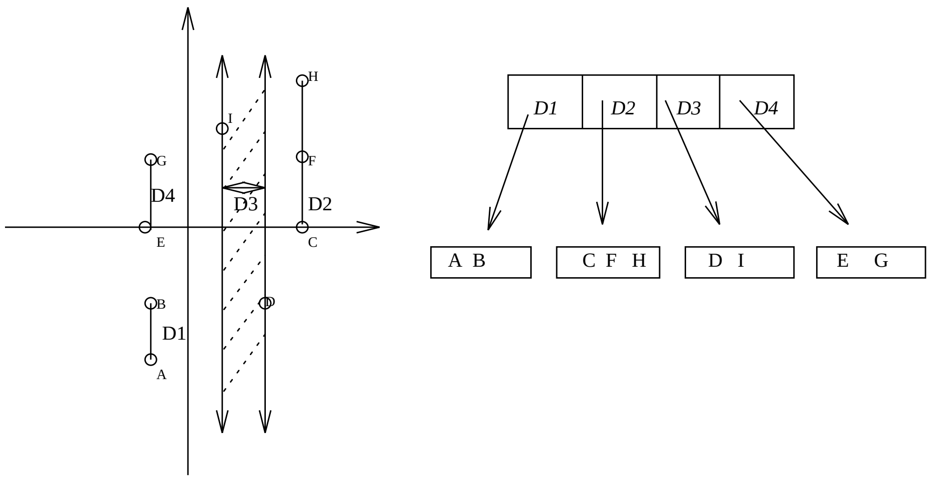


Figure 3: Example of a TV-1 tree (with diamonds)

In the TV-1 tree, the number of active dimensions is 1, thus the diamonds extend only

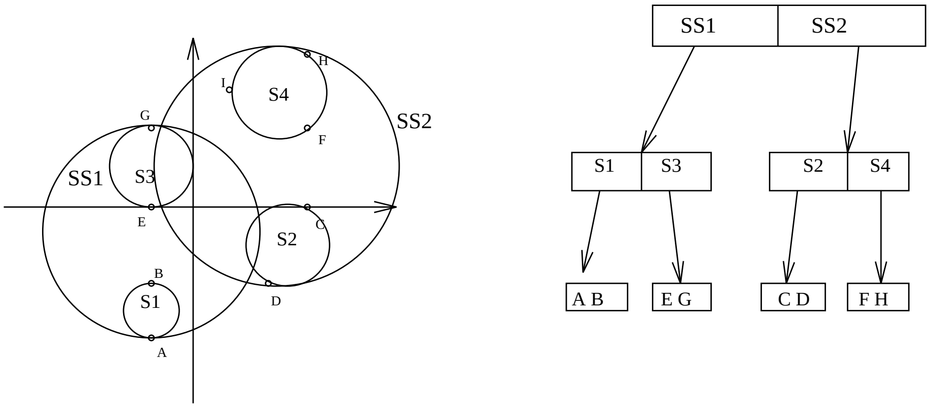


Figure 4: Example of a TV-2 tree (with spheres)

along 1 dimension at any time. As a result, the shapes are straight lines or rectangular blocks (extended infinitely). In the TV-2 case the TMBR resemble two dimensional L_p-circles.

At each stage, the number of active dimensions is exactly as specified. Sometimes, more than one level of the tree may involve using the same active dimensions. Fig. 4 is an instance of this – the same pair of active dimensions is used at both levels of the tree shown. More commonly, new active dimensions are used at each level. This would be the case in Fig. 3, for D3, when it has to be split any further.

4.3 Algorithms

Search For both exact and range queries, the algorithm starts with the root and examines each branch that intersects the search region, recursively following these branches. Multiple branches may be traversed because TMBRs are allowed to overlap. The algorithm is straightforward and the pseudo-code is omitted for brevity.

Spatial join can be handled as well. Recall that such a query requires all pairs of points that are close to each other (eg, closer than a tolerance ϵ). Again, a recursive algorithm that prunes out remote branches of the tree can be used; efficient improvements on this algorithm have recently appeared [8].

Similarly, nearest-neighbor queries can be handled with a branch-and-bound algorithm, as in [12]. The algorithm works as follows: given a (query)(query) point, examine the top-level branches, and compute upper and lower bounds for the distance; descend the most promising branch, disregarding branches that are too far away.

Insertion To insert a new object, we traverse the tree, at each stage choosing the branch that seems most suitable to hold the new object. Once we reach the leaf level, we insert the object in the leaf. Overflow is handled by splitting the node, or by re-inserting some of its contents. After the insertion/split/re-insert, we update the TMBRs of the affected nodes along the path. For example, we may have to increase the radius of a TMBR or to decrease its dimensionality (i.e. contract the telescopic vector of the center), to accommodate the new object (Figure 5).

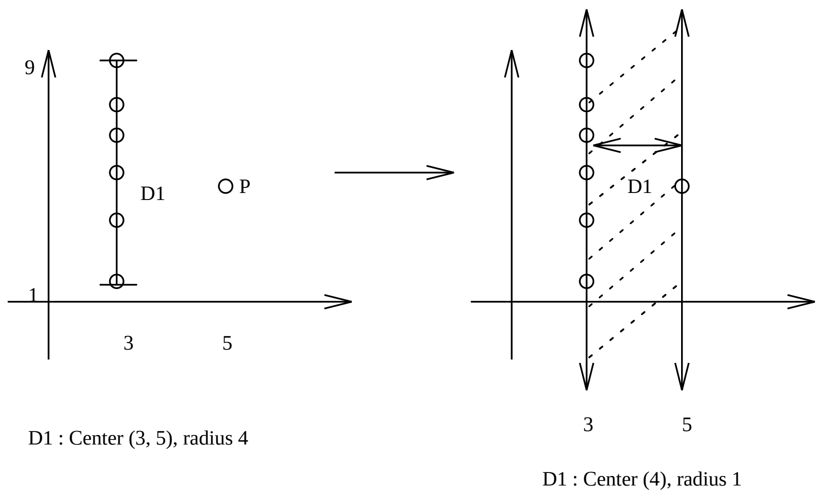


Figure 5: Decrease in dimensionality during insertion

The routine *PickBranch(Node N, element e)* examines the branches of the node N and returns the one that is most suitable to accommodate the element (point or TMBR) e to be inserted. In choosing a branch, we use the following criteria, in descending priority:

1. Minimum increase in overlapping regions within the node. (That is, choose the TMBR such that after updating it, the number of *new* pairs of overlapping TMBR within the node introduced is minimized.) An example is in figure 6 (a).

2. Minimum decrease in dimensionality. (That is, choose the TMBR with which the new object can agree on as many coordinates as possible, so that it can accommodate the new object by contracting its center as little as possible. For example, in figure 6 (b), R1 is picked over R2 so to avoid contracting of R2.

3. Minimum increase in radius (Figure 6 (c)).

4. Minimum distance from the center of the TMBR to the point (in case that the previous two criteria tie) (Figure 6 (d)).

Figure 6 shows the different instances of the branch selection.

Handling of overflowing nodes is another important aspect for the insertion algorithm. Here an overflow can be caused not only by an insertion into a full node but by an attempt to extend a 'telescopic' vector as well. Splitting of the node is the most obvious way to handle overflow. However, reinsertion can also be applied, selecting certain items to be reinserted from the top. This provides a chance to discard dissimilar items from a node, usually achieving better clustering.

In our implementation we have chosen the following scheme to handle overflow, treating the leaf node and the internal node differently:

- For a leaf node, a pre-determined percentage (p_{ri}) of the leaf contents will be reinserted if it is the first time a leaf node overflows during the current insertion. Otherwise, the leaf node is split in two. Once again, different policies that can be used to choose the elements to be reinserted. Here we choose those that are furthest away from the center of the region.

- For an internal node, the node is always split; the split may propagate upwards.

Algorithm 1 Insert algorithm

begin

/ Insert element e into tree rooted at N */*

Proc Insert(Node N, element e)

1) *Use $PickBranch()$ to choose the best branch to follow; descend the tree until the leaf node L is reached.*

2) *Insert the element into the leaf node L.*

3) *If leaf L overflows*

    If it is the first time during insertion

      Choose the p_{ri} elements furthest away from the center of L and re-insert them from the top.

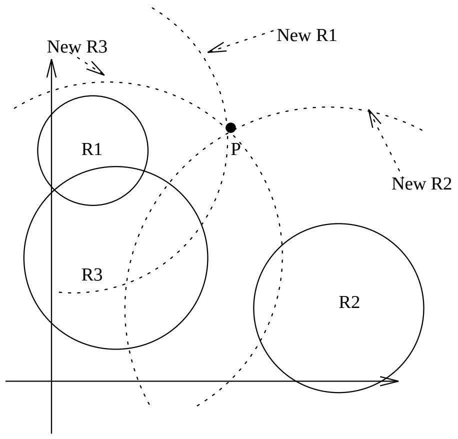
    else

      Split the leaf into two leaves.

4) *Update the TMBRs that have been changed (because of insertion and/or splitting). Split an internal node if overflow occurs.*

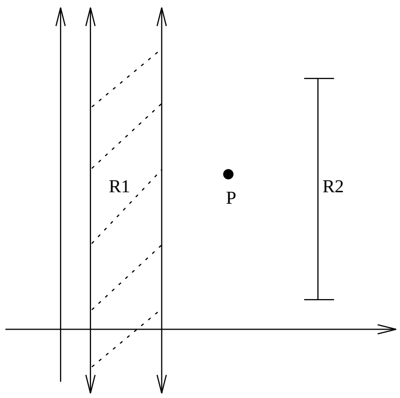
end

Splitting The goal of splitting is to redistribute the set of TMBRs (or vectors, when leaves are split) into two groups so as to facilitate future operations and provide high space



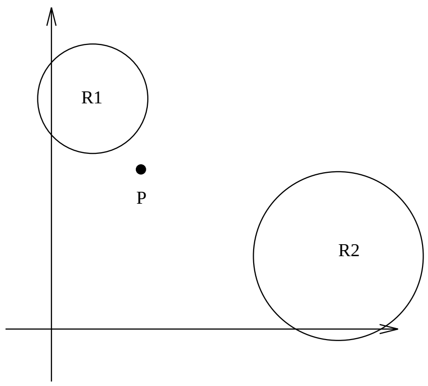
(a)

R1 is selected because extending R2 or R3 will lead to a new pair of overlapping regions



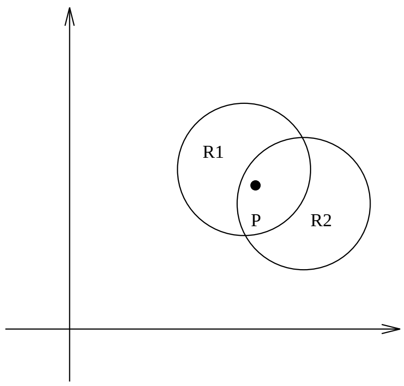
(b)

R1 is selected over R2 beacuse selecting R2 will result in a decrease in dimensionality of R2



(c)

R1 is selected over R2 because the resulting region will have a smaller radius



(d)

R1 is selected over R2 because R1's center is closer to the point to be inserted

Figure 6: Illustration of choose-branch criteria

utilization.

There are several ways to do the split. One way is to use a clustering technique. The goal is try to group vectors so that similar ones will reside in the same TMBR.

Algorithm 2 Splitting by clustering

begin

/ assume N is a internal node; similar for leaf nodes */*

Proc Split(Node N, Branch NewBranch, float min_percent)

1) *Pick as seeds the branches $B1$ and $B2$ with the two most un-similar TMBRs (ie., with the smallest common prefix in their centers; on tie, pick the pair with the largest distance between the centers). Let $R1$ and $R2$ the groups headed by $B1$ and $B2$ respectively.*

2) *For each of the remaining branches B:*

   Add B to that group $R1$ or $R2$ according to the $PickBranch()$ function

end

Another way of doing the split is by ordering: The vectors (that is, the centers of the TMBRs) are ordered in some way and a 'best' partition along the ordering is found. The current criteria being used are (in descending priority) :

1. Minimum sum of radius of the two TMBRs formed

2. Minimum of (sum of radius of TMBRs - Distance between their centers)

In other words, we first try to minimize the 'area' that the TMBRs cover; and then to minimize the overlap between the diamonds.

Ordering can be done in a few different ways. We have implemented one that sorts the vectors lexicographically. Other orderings, like some form of space-filling curves (e.g. the Hilbert curve [21]) can also be used.

Algorithm 3 Splitting by ordering

begin

/ assume N is a internal node; similar for leaf nodes */*

/ min_fill is the minimum percentage (in bytes) of the node to be occupied */*

Proc Split(Node N, Branch NewBranch, float min_fill)

1) *Sort the TMBRs of the branches by ascending row-major order of there centers.*

2) *Find the best break-point in the ordering, to create two sub-sets: (a) ignore the case where one of the subsets is too empty ($\leq$ min_fill bytes) (b) among the rest cases, choose that one break-point that the TMBRs of the two sets will have the least total radius. Break ties by minimum (sum of radius of TMBRs - distance between the centers).*

3) *If requirement (a) above leaves no candidates, then sort the branches by their byte size and repeat the above step, skipping step (a), of course.*

end

The last step in the algorithm guards against the (rare) case where one of the TMBRs has a long vector for center, while the rest have short vectors. In this case, a seemingly good split might leave one of the two new nodes highly under-utilized. The last step makes sure that the new nodes have similar sizes (byte-wise).

Deletion Deletion is straightforward, unless it causes an underflow. In this case, the remaining branches of the node are deleted and re-inserted. The underflow may propagate upwards.

Extending and contracting As mentioned previously, extending and contracting of telescopic vectors (TVECTORs) are important aspects of the algorithm. Extending is done at the time of split and reinsertion. When the objects inside a node are redistributed (either by splitting into two or removing at reinsertion), it may be the case that the remaining objects have the same values in the first few (or all) active dimensions. Thus during the recalculation of the new TMBR, extending will occur, meaning new active dimensions will be introduced and those on which all the objects agree will be rendered inactive.

An example on extending of diamonds is given in figure 7. After the extending the diamond extends only along the y-dimension.

On the other hand, contracting occurs during insertion. When an object is inserted into a TMBR such that the inactive dimensions of the TMBR do not agree completely with that of the object, the new TMBR will have to have some dimensions contracted, resulting in a TMBR with lower dimensionality.

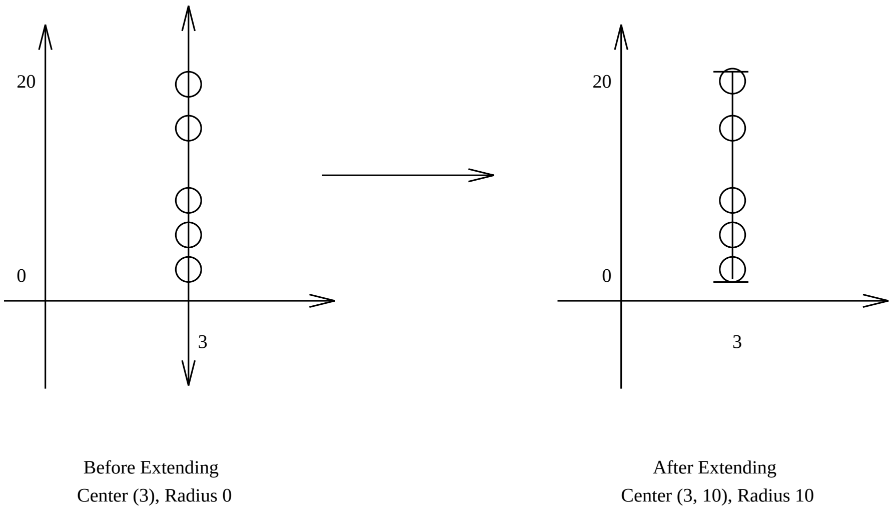


Figure 7: Extending of an TMBR (diamond), with α=1

5 Experimental results

We implemented the TV-tree as described above, in $C++$ under UNIX,[1] and we ran several experiments. The experiments form two sets: in the first, we tried to determine what is a good value for the number of active dimensions (α) for the TV-tree; in the second set we compared the proposed method with the R^*-tree, which we believe is the fastest known variation of R-trees.

5.1 Experimental setup

The test database was a collection of objects of fixed size, using dictionary words from `/usr/dict/words` as keys. The queries were exact match and range queries, to find the closest matches in the presence of typing errors. For features, we used the letter count for each word, ignoring the 'case' of the letters. Thus, each word is mapped to a vector v with 27 dimensions, one for each English alphabet letter, and an extra one for the non-alphabetic characters. The L_1 distance among two such vectors is a good measure for the editing distance; for this reason, we have used L_1-spheres (diamonds) as our bounding shapes.

Finally, we apply the Hadamard Transform[2] For $n = 2^k$, the Hadamard Transform

[1] UNIX is a registered Trademark of UNIX System Laboratories

[2] Actually, we are using the 32-dimension Hadamard Transform matrix and padding extra 0s to the feature vectors.

matrix is defined as follows:

$$H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, H_{k+1} = \begin{pmatrix} H_k & H_k \\ H_k & -H_k \end{pmatrix}$$

on these 'letter-count' vectors, appropriately zero-padded. The Hadamard Transform is used in order to give each letter a more even weight, especially in the first few dimensions.

The TV-trees used in the experiment use the algorithms described in the last section, with forced re-insertion, and with the 'ordering' method for splitting. We used $min\_fill =$ 45% and the percentage of elements to be reinserted to be p_{ri}=30%. These numbers are comparable to the parameter for the optimal R^*-tree parameters. This number is chosen in order to provide a fair comparison for insertion behavior.

Experiments on 2,000 to 16,000 words are run, with words being randomly drawn from the dictionary. We varied several parameters, such as the number of active dimensions α (from 1 to 4) and the tolerance ϵ of the range query, from $\epsilon = 0$ (exact match) up to 2.

For the exact match queries, we tried 'successful' searches (ie, the query word was found in the dictionary), using half of the database words as query points. Experiments with 'unsuccessful search' gave similar results and are omitted. We also issued range queries with the words randomly drawn from the dictionary, (the number of queries are half of the database words).

We measured both the number of disk accesses (assuming that the root is in core), as well as the number of leaf accesses. The former measure corresponds to an environment with limited buffer space; the latter approximates an environment with so much buffer space that, except for the leaves, the rest of the tree fits in core.

5.2 Results

Analysis for the number of active dimensions The first set of experiments tries to determine a good value for α. Different number of active dimensions of the TV-tree were tried. The results are shown in figures (8) - (10). The page size was 4K bytes and objects of size 100 bytes are used.

We also measured the total number of pages accessed, assuming the whole tree (except the root) is stored on the disk and no buffer for the internal levels is available. The results are similar.

The results indicate that $\alpha = 2$ gives the best results, since the TV-2 tree outperforms the rest. This can be interpreted as an optimization of two conflicting factors : tree size and number of false drops. With a smaller α, fewer dimensions will be available to differentiate

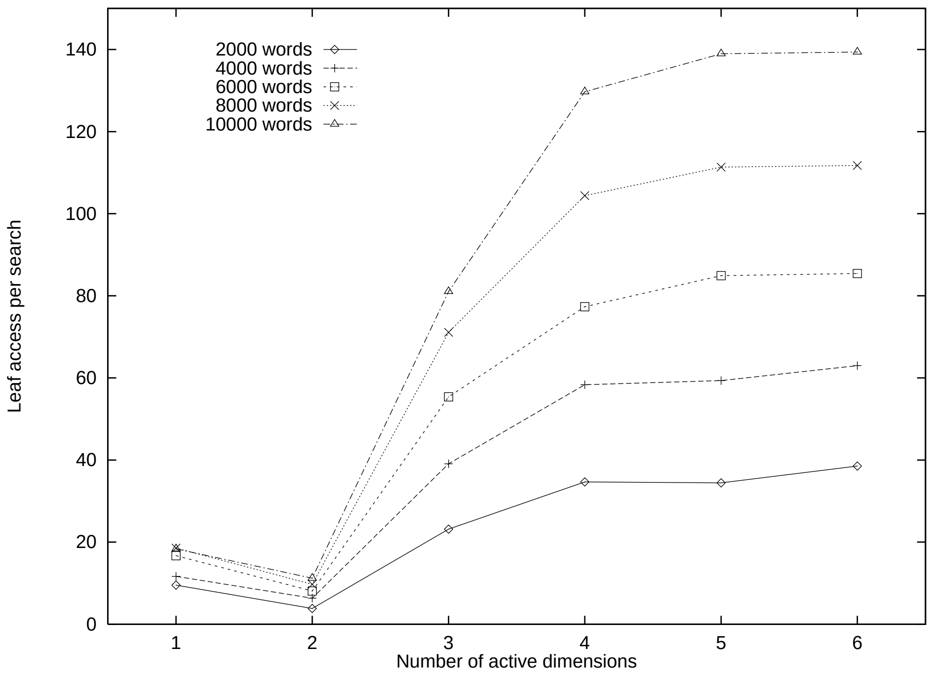


Figure 8: Exact match queries (# leaf accesses vs. α)

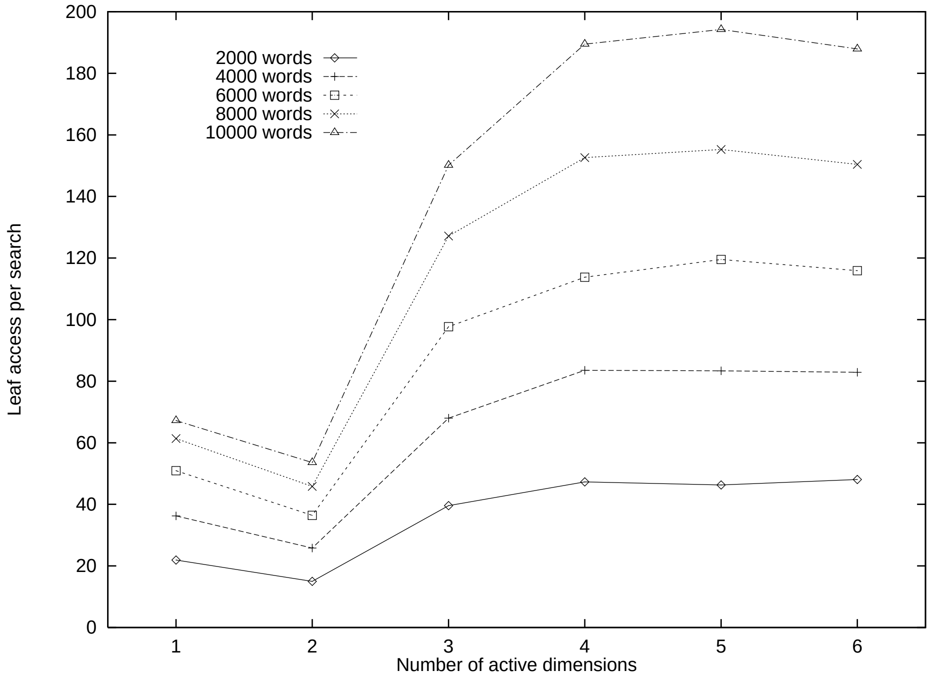


Figure 9: Range queries (tolerance=1) (# of leaf accesses vs. α)

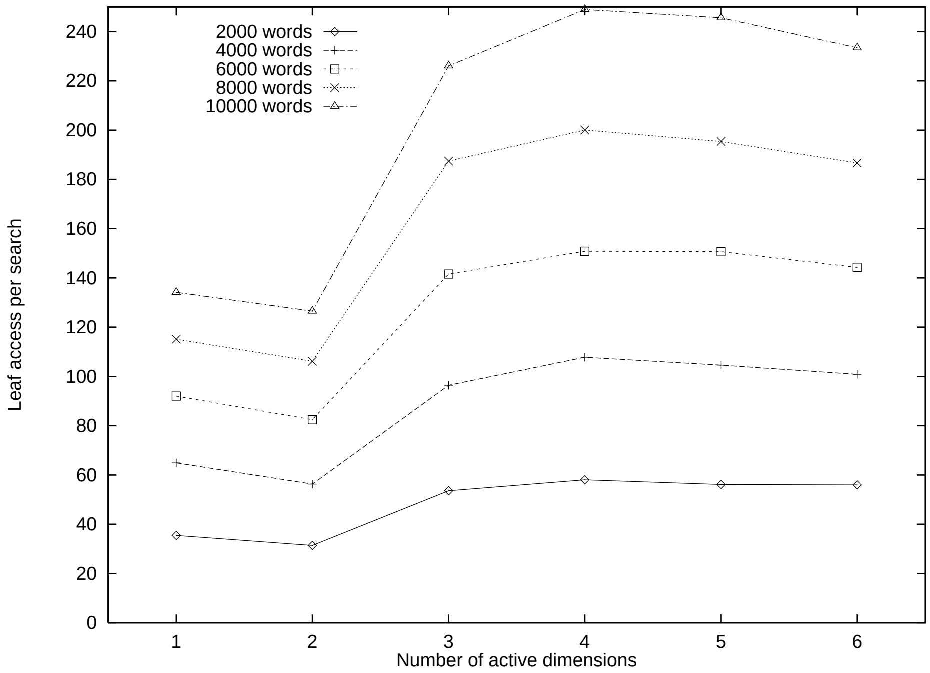


Figure 10: Range queries (tolerance=2) (# of disk accesses vs. α)

among the entries, thus more branches will have to be searched. However, a larger α will lead to a decrease of fanout per node, making it necessary for more branches to be retrieved when the search space is large. Moreover, effectively clustering objects in higher dimensions is also more difficult, giving the constraints in shapes allowed. (In 1-d, one can always sort the numbers and order it; but this method breaks down in higher dimensions). In the experiments we ran , $\alpha = 2$ is the best compromise.

Comparison with R^*-tree

Index creation We measured the number of disk accesses (read + write) needed for building the indices. We assume every update of the index will be reflected on the disk. We found out that in general the insertion cost is cheaper in the TV-tree. This is due to the fact that the TV-tree is usually shallower than the corresponding R^*-tree, thus fewer nodes need to be retrieved and fewer potential updates need to be written back to disk. The following table show the result for object size 100 bytes with a 4K page size.

The big jump between 4000 and 8000 for the TV-2 tree is because of an addition level introduced. However, the TV-2 tree still has one fewer level then the R^*-tree. Thus the increase in disk access for the TV-2 tree is slower afterwards.

Table 1: Disk access per insertion – object size 100 bytes

Dictionary size	Disk access per insertion	
	R^*-tree	TV-2 tree
4000	5.25	4.75
8000	5.51	5.21
12000	6.19	5.28
16000	6.50	5.35

Search The next set of experiments compare the proposed TV-tree against the R^*-tree. Figures (11)-(13) show the number of disk/leaf accesses as a function of the database size (number of records). The number of leaf accesses is the lower curve, in each set. A 4K page size is used The following results are for objects of size 100 bytes.

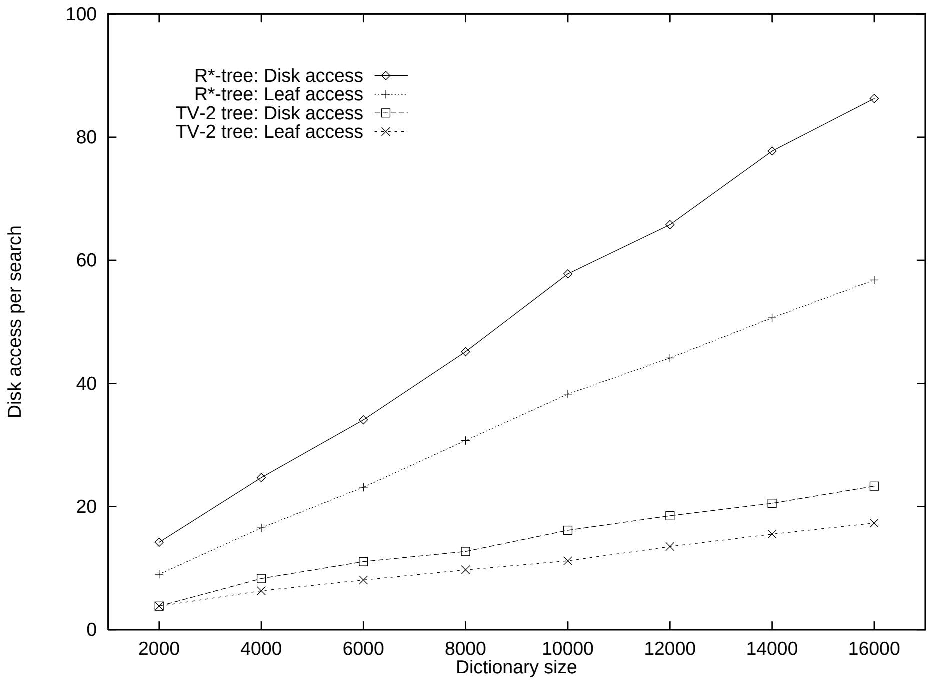


Figure 11: Disk/leaf accesses vs. db size - exact match queries

As seen from the figures, the TV-2 tree consistently outperforms the R^*-tree, with up to 67-73% savings in total disk accesses for exact matches and similar savings in leaf accesses. The savings for range queries are also high ($\approx$ 40% for large dictionary size).

Moreover, the savings increase with the size of the database, indicating that our proposed

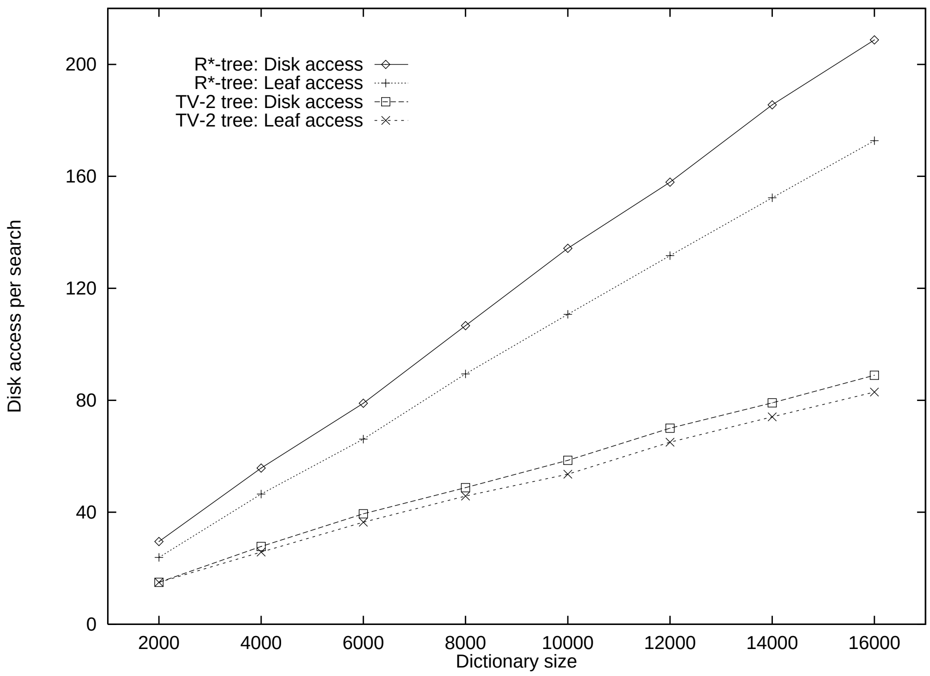


Figure 12: Disk/leaf accesses vs. db size - range queries (tolerance=1)

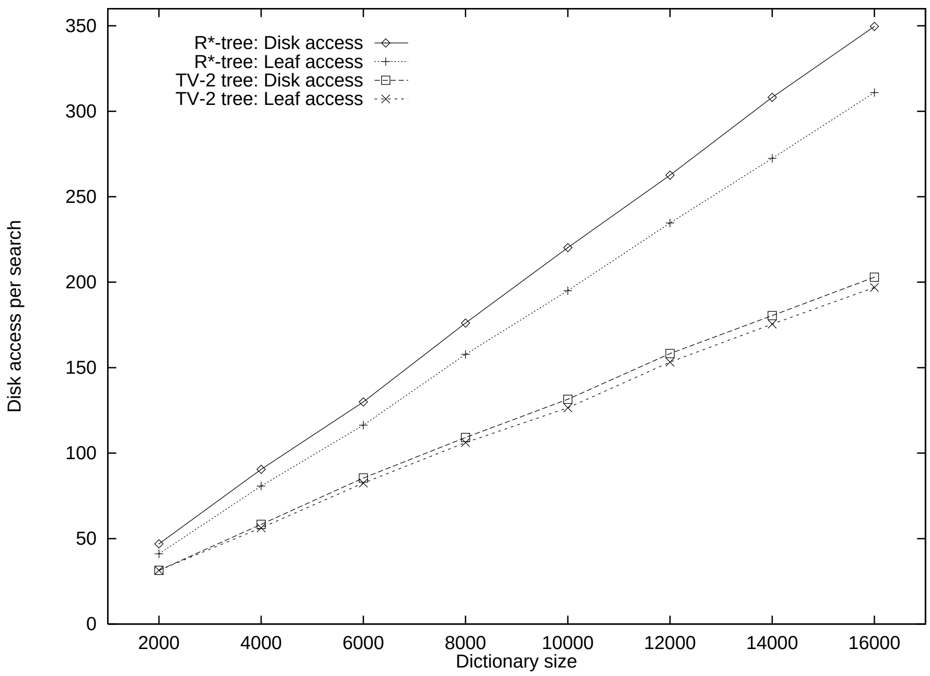


Figure 13: Disk/leaf accesses vs. db size - range queries (tolerance=2)

method scales up well: As the database size grows from 2,000 to 16,000 elements, the savings in the number of leaf accesses increased consistently: from 67% to 73% for exact match queries, from 50% to 58% for range queries with tolerance ϵ=1 and from 33% to 42% for range queries with ϵ=2.

Even if we assume only the leaves being stored in the disk (while all the non-leaf levels are being read into memory buffer beforehand), the TV-2 tree still outperform the R^*-tree significantly (around 60-70% for exact match and 25-35% for range queries with ϵ=2).

We also experimented with various size of database objects. Our method will show more significant improvement when object size is small. As object size increases, the leaf fan-out decreases, making the TV-tree grow faster, offsetting some of the advantages of it. However, even with object size 200, we still have improvement of around 60% for exact match and 40% for range queries with ϵ=2.

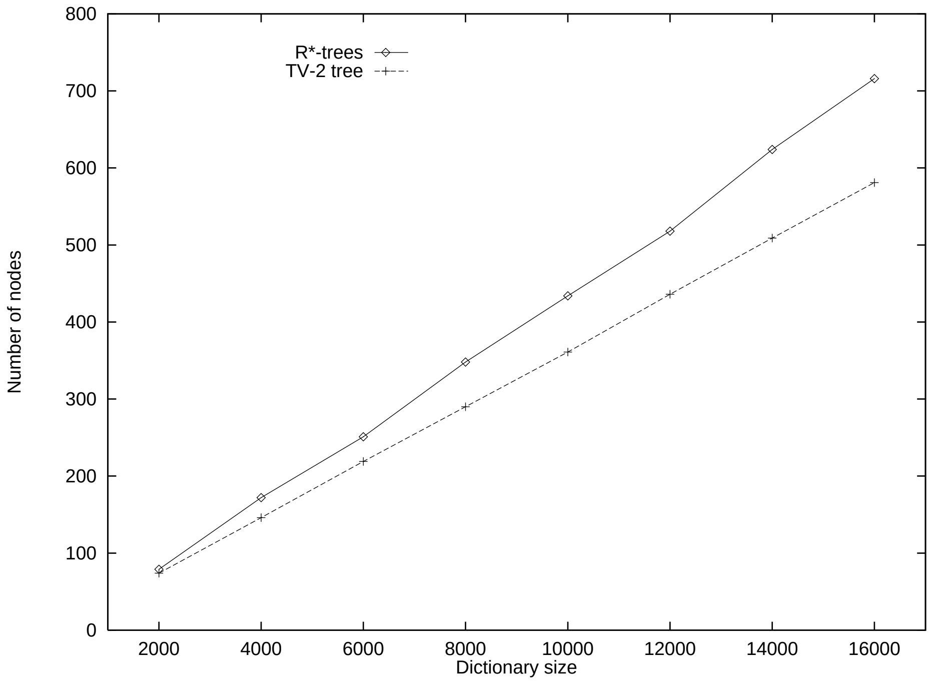


Figure 14: Comparison of space requirements

Comparison of space requirements Figure (14) shows the number of nodes (= pages) in the trees. The TV-tree requires fewer number of nodes (and thus less space). The savings are 15-20%.

Since the object size is the same for both indices, the number of leaf nodes are also very similar (In fact, they will be identical when the utilization is the same). This implies all the savings in the TV-tree are from internal nodes, which means that the non-leaf levels will require a smaller buffer, which can be significant when buffer space is limited.

6 Discussion - Conclusions

In this paper we proposed the TV-tree as a method for indexing high dimensional objects. The novelty lies in its ability to adapt dynamically and use a variable (typically, small) number of dimensions, that are needed to distinguish between objects or groups of objects. Since this number of required dimensions will usually be small, the method saves space and leads to a larger fan-out. As a result, the tree is more compact and shallower, requiring fewer disk accesses.

We presented the manipulation algorithms in detail, as well as guidelines on how to choose the design parameters (optimal active dimension α=2, minimum fill factor = 45% etc). We implemented the method, and we reported performance experiments, comparing our method to the R^*-tree. The TV-tree achieved access cost savings of up to 80%, while at the same time resulting in a reduction in the size of the tree, and hence its storage cost. Moreover, the savings seem to increase with the size of the database, indicating that our method will scale well.

In short, we believe that the TV-tree should be the method of choice for high dimensional indexing.

Acknowledgments: The authors would like to thank Alexios Delis and Ibrahim M. Kamel for their help during the write-up.

A Appendix: Calculation of the Telescopic Minimum Bounding Diamond (TMBD)

To find the TMBD of a given set of points or diamonds, we first find the largest m such that all the TVECTORS (centers of the diamond or vectors corresponding to data points) agree in the first m dimensions. Then we project the next α dimensions, where α is the number of active dimensions of the TV-tree. Thus the projected diamonds will reside in a α-dimensional space. An example is given in table 2, assuming the diamonds are from a TV-2 tree.

In table 2, m is 2 (and α is 2 by definition of the TV-2 tree). Note that the projected second diamond has a radius of 0 because the third and fourth dimensions are not active

Table 2: Example of diamond projection in a TV-2 tree

Original diamond		Projected diamond	
Center	Radius	Center	Radius
$(1,0,3,4)$	2	$(3,4)$	2
$(1,0,8,7,5,6)$	4	$(8,7)$	0
$(1,0,2,6)$	1.5	$(2,6)$	1.5

dimensions. That means all points inside this diamond will have coordinates that start with $(1,0,8,7,\ldots)$.

From there we find the minimum bounding diamond of the projected diamonds, and use its center as the active dimensions of the final MBD. The non-active dimensions will be the common m dimensions we first found.

Finding the minimum bounding diamond of these projected diamonds can be formulated as a linear programming problem. However, we decided to use a faster approximation algorithm to find the approximate MBD. The algorithm first calculates the bounding (hyper)rectangle of the projected diamonds, and then use its center as the diamond center. The smallest radius that is needed to cover all the diamonds is then calculated.

Algorithm 4 Finding the MBD

begin

/ α is the number of active dimensions */*

Proc TMBD(Array of Diamonds D, integer α);

1) *Find min, the minimum dimensionality among all diamonds in D.*

2) *Find the maximum m such that all the diamonds have the same first m dimensions.*

3) *If $m + \alpha \leq min$*

   Set $Startproject \leftarrow m + 1$

   else

   Set $Startproject \leftarrow min - \alpha + 1$ / special case when some diamonds have small dimensionality. This step is to ensure there will be α active dimensions */*

4) *Project each diamond to dimensions $Startproject \ldots Startproject + \alpha - 1$, setting the radius to 0 if non of the projected dimension is active, and retain the original radius otherwise.*

5) *Find the minimum bounding rectangle of the projected diamonds. Let $c \leftarrow$ center.*

6) *Set center of the result diamond* $\leftarrow$ *the* m *common dimensions of the diamonds concatenated with* c

7) *Find the minimum distance that is needed to contain all diamonds, and set this as the radius.*

end

Continuing the example from table 2, the bounding rectangle for the projected diamonds have boundary (0.5, 8) along the first dimension and (2, 7.5) for the second. Its center is (4.25, 4.75). The radius required to cover all the three diamonds is 6. Thus the final TMBR has center (1, 0, 4,25, 4.75) and radius 6.

B Appendix: Telescoping Without Truncation

Given a feature vector of length n, its contraction to length m is achieved through multiplication by the matrix A_m. Here we present an example of a simple, summation-based, telescoping function that does not involve truncation. The required series of matrices A_m are:

If $n \leq 2m$, A_m has a 1 in position (1,1), (2,2), ..., $(2m-n,\ 2m-n)$, $(2m-n+1,\ 2m-n+1)$, $(2m-n+2, 2m-n+1)$, $(2m-n+3, 2m-n+2)$, $(2m-n+4, 2m-n+2)$, ..., $(n,\ m)$, and a 0 everywhere else.

In other words, the first $2m-n$ rows have a single 1 each on the "diagonal", and the remaining $n-m$ rows have two 1's each, in pairs, in a stretched out continuation of the diagonal. Call this the *halving step.*

If $n/4 \leq m < n/2$, obtain the matrix A_p, where $p = ceiling(n/2)$, using the halving step, and then apply the halving step once more to the p length vector to create an m length vector. A_m is obtained as the product of the two matrices for each application of the halving step.

Similarly, for any value of m, enough applications of the halving step produce the required contraction. The contraction for $m = 1$ is simply the summation of all elements, induced by a matrix A_m which is a vector of all 1's.

References

[1] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *FODO Conference*, Evanston, Illinois, October 1993. to appear.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 1990.

[3] R.C. Angell, G.E. Freund, and P. Willet. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management*, 19(4):255–261, 1983.

[4] Manish Arya, William Cody, Christos Faloutsos, Joel Richardson, and Arthur Toga. Qbism: a prototype 3-d medical image database system. *IEEE Data Engineering Bulletin*, March 1993.

[5] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD*, pages 322–331, May 1990.

[7] Jon Louis Bentley, Bruce W. Weide, and Andrew C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. on Mathematical Software (TOMS)*, 6(4):563–580, December 1980.

[8] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *Proc. of ACM SIGMOD*, pages 237–246, May 1993.

[9] Christopher Chatfield. *The Analysis of Time Series: an Introduction.* Chapman and Hall, London & New York, 1984. Third Edition.

[10] Jerome H. Friedman, Forest Baskett, and Leonard H. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. on Computers (TOC)*, C-24:1000–1006, October 1975.

[11] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 1990. 2nd Edition.

[12] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. on Computers (TOC)*, C-24(7):750–753, July 1975.

[13] D. Greene. An implementation and performance analysis of spatial data access methods. *Proc. of Data Engineering*, pages 606–615, 1989.

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[15] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, 1975.

[16] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. *Proc. of ACM SIGMOD Conf.*, pages 205–214, June 1992.

[17] G.M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Trans. on PAMI*, PAMI-1(2):145–153, April 1979.

[18] H. V. Jagadish. Spatial search with polyhedra. *Proc. Sixth IEEE Int'l Conf. on Data Engineering*, February 1990.

[19] H.V. Jagadish. A retrieval technique for similar shapes. *Proc. ACM SIGMOD Conf.*, pages 208–217, May 1991.

[20] Mark A. Jones, Guy A. Story, and Bruce W. Ballard. Integrating multiple knowledge sources in a bayesian ocr post-processor. In *First International Conference on Document Analysis and Recognition*, Saint-Malo, France, September 1991. to appear.

[21] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: an improved r-tree using fractals. Systems Research Center (SRC) TR-93-19, Univ. of Maryland, College Park, 1993.

[22] Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–440, December 1992.

[23] B. Mandelbrot. *Fractal Geometry of Nature.* W.H. Freeman, New York, 1977.

[24] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[25] A. Desai Narasimhalu and Stavros Christodoulakis. Multimedia information systems: the unfolding of a reality. *IEEE Computer*, 24(10):6–8, October 1991.

[26] Wayne Niblack, Ron Barber, Will Equitz, Myron Flickner, Eduardo Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. The qbic project: Querying images by content using color, texture and shape. *SPIE 1993 Intl. Symposium on Electronic Imaging: Science and Technology, Conf. 1908, Storage and Retrieval for Image and Video Databases*, February 1993. Also available as IBM Research Report RJ 9203 (81511), Feb. 1, 1993, Computer Science.

[27] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[28] J.A. Orenstein and F.A. Manola. Probe spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Engineering*, 14(5):611–629, May 1988.

[29] Mary Beth Ruskai, Gregory Beylkin, Ronald Coifman, Ingrid Daubechies, Stephane Mallat, Yves Meyer, and Louise Raphael. *Wavelets and Their Applications*. Jones and Bartlett Publishers, Boston, MA, 1992.

[30] G. Salton and A. Wong. Generation and search of clustered files. *ACM TODS*, 3(4):321–346, December 1978.

[31] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[32] Manfred Schroeder. *Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise*. W.H. Freeman and Company, New York, 1991.

[33] Gregory K. Wallace. The jpeg still picture compression standard. *CACM*, 34(4):31–44, April 1991.

Contents