VLDB Journal, 3, 517-542 (1994), Ralf Hartmut Güting, Editor
©VLDB

The TV-Tree: An Index Structure for High-Dimensional Data

King-Ip Lin, H.V. Jagadish, and Christos Faloutsos

Received July 12, 1993; accepted May 20, 1994.

Abstract. We propose a file structure to index high-dimensionality data, which are typically points in some feature space. The idea is to use only a few of the features, using additional features only when 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, which saves up to 80% in disk accesses.

Key Words. Spatial index, similarity retrieval, query by content.

1. Introduction

Many applications require enhanced indexing that is capable of performing similarity searching on several, non-traditional (exotic) data types. The target scenario is as follows: given a collection of objects (e.g., 2-D images, 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:* Jagadish (1991) showed how to query for similar shapes, describing each shape by the coordinates of a few rectangles that cover it ($\approx$20 features per shape). Niblack et al., (1993) supported queries on color, shape and texture, using color histograms (64-256 attributes per image) as feature vectors, and using the first 20 moments for shapes.

King-Ip Lin is a graduate student, and Christos Faloutsos, Ph.D., is Associate Professor, Department of Computer Science, University of Maryland, College Park, MD 20742; H.V. Jagadish, Ph.D., is with AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

- *Medical databases,* where 1-D objects (e.g., ECGs), 2-D images (e.g., x-rays), and 3-D images (e.g., MRI brain scans; Arya et al., 1993) are stored. The ability to retrieve quickly past cases with similar symptoms is 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 have appeared in the past. Agrawal et al. (1993) used the co-efficients of the Discrete Fourier Transform (DFT) as features.
- *Multimedia databases,* with audio (voice, music) or video (Narasimhalu and Christodoulakis, 1991). Users might want to retrieve similar music scores or video clips.
- *DNA databases* that contain 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 (Altschul et al., 1990) uses successive, overlapping n-grams for indexing. When using n-grams as features, we need 4^n features or 1,024 features for $n = 5$.
- *Searching for names or addresses,* (e.g., in a customer mailing list), which 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 (Kukich, 1992), and OCR error correction. Given a wrong string, we should search a dictionary to find the closest strings to it. Triplets of letters are often used to assess the similarity of two words (Angell et al., 1983), in which case we have $\approx 26^3 = 17{,}576$ features per word (assuming that words consist exclusively of the 26 English letters, ignoring digits, upper-case letters, etc.).

For all of these applications, we rely on an expert to derive features that adequately describe the objects of interest. As Jagadish (1991) proposed, once objects are mapped into points in some feature space, we can accelerate the search 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 range from 10 to 100. The spatial access methods of the past have mainly concentrated on 2-D and 3-D spaces, such as the R-tree based methods (Guttman, 1984), and the linear-quadtree based methods (e.g., z-ordering; Orenstein and Manola, 1988). Although conceptually they can be extended to higher dimensions, they usually require time and/or space that grows exponentially with the number of dimensions.

In this article, we propose a tree-structure that avoids 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 current level of the tree. 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, using 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.

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

2. Related Work

As mentioned above, feature extraction functions map objects into points in feature space for a variety of applications; these points must be stored in a spatial access method. The prevailing methods form three classes: R^*-trees (Beckmann et al., 1990) and the rest of the R-tree family (Guttman, 1984; Jagadish, 1990); linear quadtrees (Samet, 1989); and grid-files (Nievergelt et al., 1984).

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

- *Exact match queries.* Find whether 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 are within a certain distance from the object. Similarity queries also fall within this category. For example, find all buildings within 2 miles of 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 neighbor queries.* Given a query item, find the item that 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 of each other.
- *Sub-pattern matching.* Instead of looking at the objects as a whole, find a sub-pattern within an object that matches our description. For example, find stock movements that contain a certain pattern; or find all x-ray images that contain tissue with tumor-like texture.

Previous work compared the performance of different spatial data structures. Greene (1989) compared the R-tree, $R+$-tree, K-D-B-tree, and the 2-D Index Sequential Access Method, and concluded that the R-tree and the $R+$-tree give the better performances. Hoel and Samet (1992) compared the *PMR*-quadtree to the R-tree variants for large line segment databases. Their results show that different data structures are suited for different kinds of queries.

Most multidimensional indexing methods, however, explode exponentially with the dimensionality, eventually reducing to sequential scanning. For linear quadtrees, the effort is proportional to the hypersurface bounding the query region (Hunter and Stieglitz, 1979); the hypersurface grows exponentially with the dimensionality. Grid files face similar problems, because 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 focus mainly on nearest-neighbor queries: Voronoi diagrams do not work at all for dimensionalities higher than 3 (Aurenhammer, 1991). The method of Friedman et al. (1975) does almost as much work as linear scanning for dimensionalities > 9. The spiral search method of Bentley et al. (1980) also has a complexity that grows exponentially with the dimensionality.

Relevant to our work is a wide variety of clustering algorithms (e.g., Hartigan, 1975; Salton and Wong, 1978; Murtagh, 1983, for surveys). However, the main goal of these algorithms is 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, several of the target applications require indexing in a high-dimensional feature space. Current spatial access methods suffer from the *dimensionality curse* (i.e., 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 classify objects: for example, in zoology, the species are grouped in a few broad classes first, using a few features (e.g., vertebrates versus invertebrates). As the classification is further refined, more and more features are gradually used (e.g., warm-blooded versus cold-blooded, or lungs versus gills).

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 (i.e., feature vectors) are clustered into leaf nodes of the tree, and the description of their *Minimum Bounding Region* (MBR) is stored in the 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, features.

As more objects are 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 that describe bounding regions of any shape (cubes, spheres, rectangles, etc.). Also, there is flexibility in the choice of the telescoping function, which selects the features of interest 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 (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 article, we assume that truncation is the telescoping function selected.

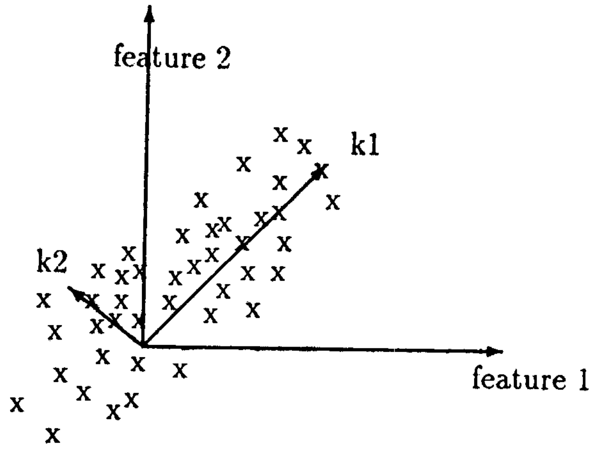
The proposed method treats the features asymmetrically, favoring the first few features over the rest, when truncation is used as the telescoping function. For similarity queries, which are likely to be frequent in the application domains we have in mind, it is intuitive that well ordered features will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

In most applications, transforming the given feature vector will achieve good ordering. Ordering the features on the basis of importance is exactly what the Karhunen Lowe (KL) transform achieves (Fukunaga, 1990): 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 KL transform on the illustrated set of points.

The KL transform is optimal if the set of data is known in advance (i.e., 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, or files with customer mailing lists that are updated in batches. The KL 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 Transform (DCT; Wallace, 1991), the Discrete Fourier Transform (DFT), the Hadamard Transform (Hamming, 1977), and the Wavelet

Figure 1. Illustration of the Karhunen Lowe transform



Transform (Ruskai et al., 1992). Fortunately, many data-independent transforms will perform as well as the KL if the data follow specific statistical models. For example, the DCT is 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 (Wallace, 1991) exactly exploits this phenomenon, effectively ignoring the high-frequency components of the DCT. 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 DFT in time series (Agrawal et al., 1993). 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 (Mandelbrot, 1977; Chatfield, 1984). Birkhoff's theory (Schroeder, 1991) 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 the order of importance. 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 simplest shape 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 (*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. The algorithm for the diamond shape is presented in Appendix A.

Minor modifications are required in the *TV*-tree algorithms to accommodate other popular shapes, such as rectangles or ellipses. Compared to L_p-spheres, these shapes differ only in 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, or which branch to traverse during insertion.

For the rest of this article, we concentrate on L_p-spheres as MBRs.

4. The TV-tree

4.1 Node Structure

Each node in the *TV*-tree represents the MBR (an L_p-sphere) of all of 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 also call the center of the region a telescopic vector (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.

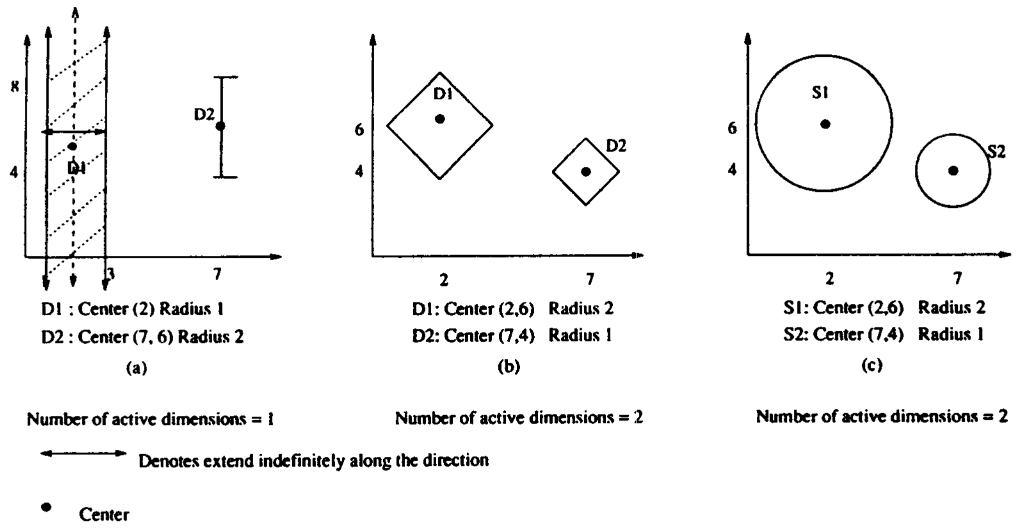
Definition. A telescopic L_p-sphere with center $\vec{c}$ and radius r, with *dimensionality d* and with α *active dimensions* contains the set of points $\vec{y}$ such that

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

and

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

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


8
4
D2
D1
3
7
D1 : Center (2) Radius 1
D2 : Center (7, 6) Radius 2
(a)
Number of active dimensions = 1
6
4
D1
D2
2
7
D1: Center (2,6) Radius 2
D2: Center (7,4) Radius 1
(b)
Number of active dimensions = 2
6
4
S1
S2
2
7
S1: Center (2,6) Radius 2
S2: Center (7,4) Radius 1
(c)
Number of active dimensions = 2
Denotes extend indefinitely along the direction
Center

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).

We need this concept because, 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.

In our presentation, *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 of all its TMBRs.

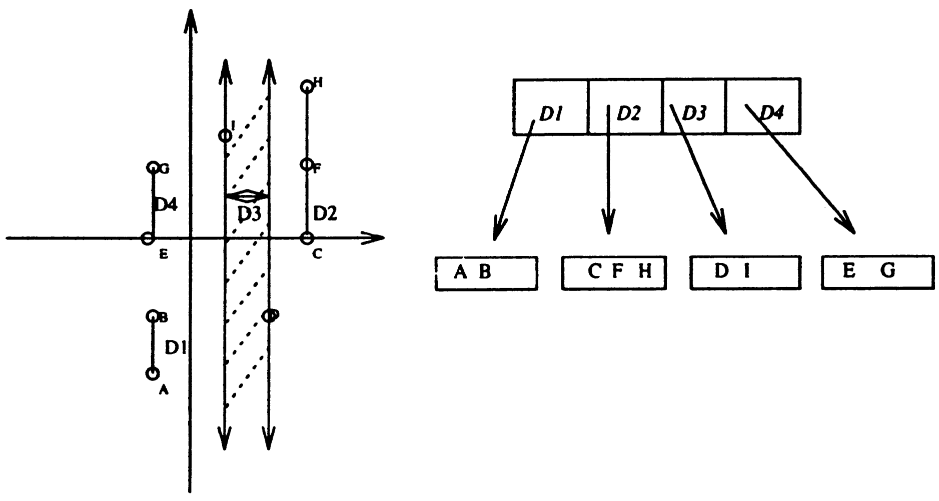
The notation TV-1 denotes a *TV*-tree with α=1; Figure 2 shows the TMBRs of TV-1 and TV-2 trees. The discriminatory power of the tree is determined by α. Whenever more discriminatory power is needed, new dimensions are introduced to ensure that the number of active dimensions remains the same.

The data structure for a 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 3. Example of a TV-1 tree (with diamonds)



4.2 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; and each node occupies exactly one disk page.

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

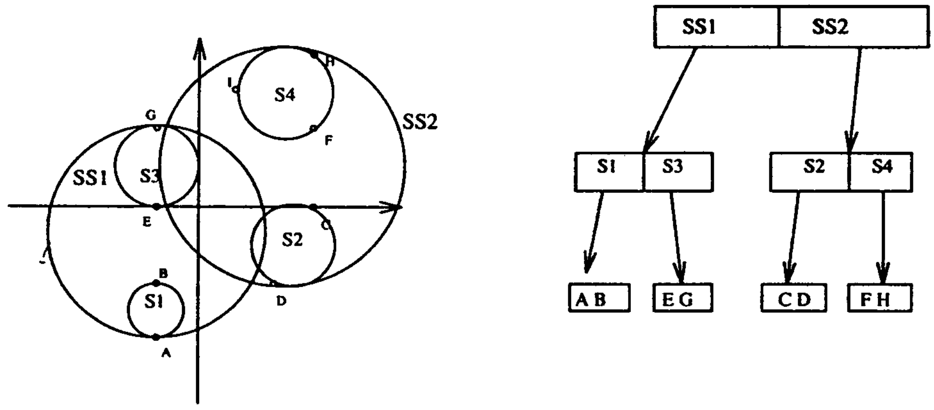
In the TV-1 tree, the number of active dimensions is 1, thus the diamonds extend only along 1 dimension at any time. As a result, the shapes are straight lines or rectangular blocks (extended infinitely). In the TV-2 case, the TMBR resembles two dimensional L_p-circles.

At each stage, the number of active dimensions is exactly as specified. Sometimes, more than one level of the tree may using the same active dimensions. Figure 4 is an example; 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 is the case in Figure 3 when D3 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.

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



Spatial join can be handled as well. Recall that such a query requires all pairs of points that are close to each other (i.e., 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 (Brinkhoff et al., 1993).

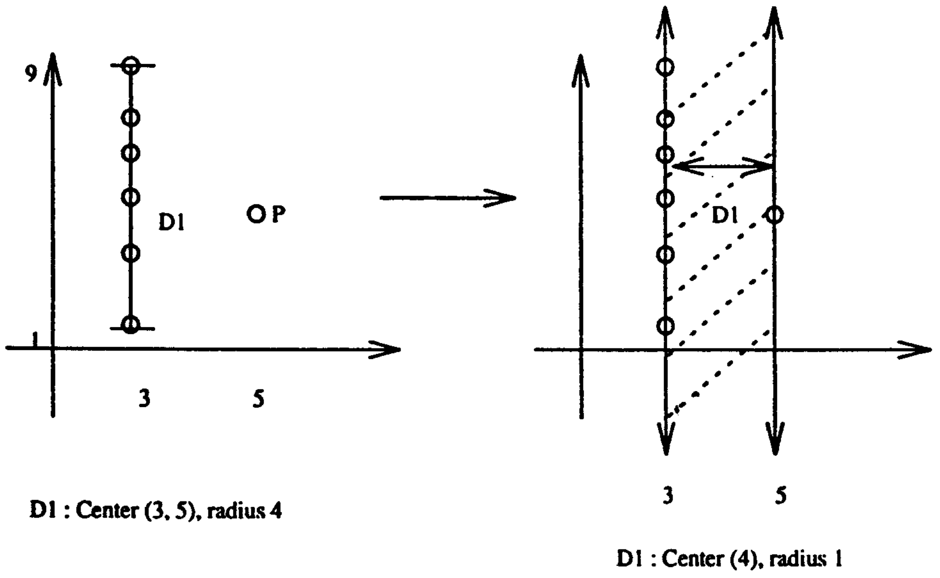
Similarly, nearest-neighbor queries can be handled with a branch-and-bound algorithm (Fukunaga and Narendra, 1975). The algorithm works as follows: given a (query)(query) point, examine the top-level branches, and compute upper and lower bounds for the distance; descend the most promising branch, disregarding branches that are too far away.

Insertion. To insert a new object, we traverse the tree, choosing the branch at each stage 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 decrease its dimensionality (i.e., contract the telescopic vector of the center), to accommodate the new object (Figure 5).

The routine *PickBranch(Node N, element e)* examines the branches of the node N and returns the branch 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 (i.e., choose the TMBR such that after update, the number of *new* pairs of overlapping TMBR is minimized within the node introduced; Figure 6*a*).
2. Minimum decrease in dimensionality (i.e., choose the TMBR with which the new object can agree on as many coordinates as possible, so that it can

Figure 5. Decrease in dimensionality during insertion



accommodate the new object by contracting its center as little as possible. For example, in Figure 6*b*, *R*1 is chosen to avoid contracting *R*2.

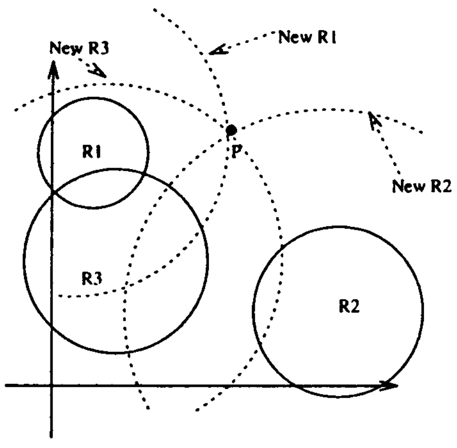
3. Minimum increase in radius (Figure 6*c*).
4. Minimum distance from the center of the TMBR to the point (in case the previous two criteria tie; Figure 6*d*).

Handling overflowing nodes is another important aspect of 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 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:

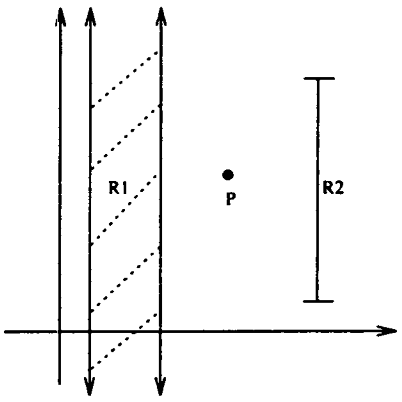
- For a leaf node, a pre-determined percentage (p_{ri}) of the leaf contents will be reinserted if it is the first time a leaf node overflows during the current insertion. Otherwise, the leaf node is split in two. Once again, different policies can be used to choose the elements to be reinserted. Here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

Figure 6. Illustration of choose-branch criteria



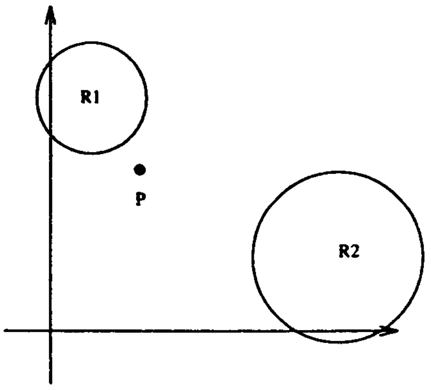
(a)

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



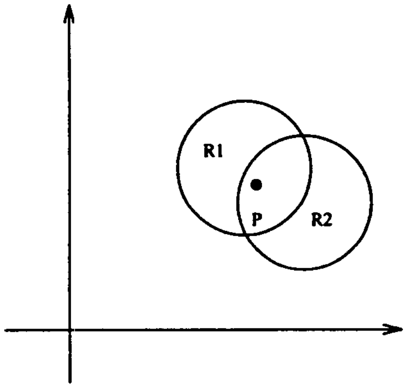
(b)

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

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 farthest 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 to facilitate future operations and provide high space utilization. There are several ways to do the split. One way is to use a clustering technique that groups vectors so that similar ones will reside in the same TMBR.

Algorithm 2. Splitting by clustering

```
begin
    /* assume $N$ is an 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 dissimilar TMBRs
       (i.e., the two with the smallest common prefix in their centers; on tie, pick
       the pair with the largest distance between their centers). Let $R1$ and $R2$ be
       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 (i.e., the centers of the TMBRs) are ordered in some way and the 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 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, such as a form of space-filling curves (e.g., the Hilbert curve; Kamel and Faloutsos, 1993) can also be used.

Algorithm 3. Splitting by ordering

```
begin
    /* assume N is an 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 their
       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 small ($\leq$ min_fill bytes); (b) among
       the remaining cases, choose the break-point such that the sum of the radius
       of the TMBRs of the two sets is the smallest. 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).

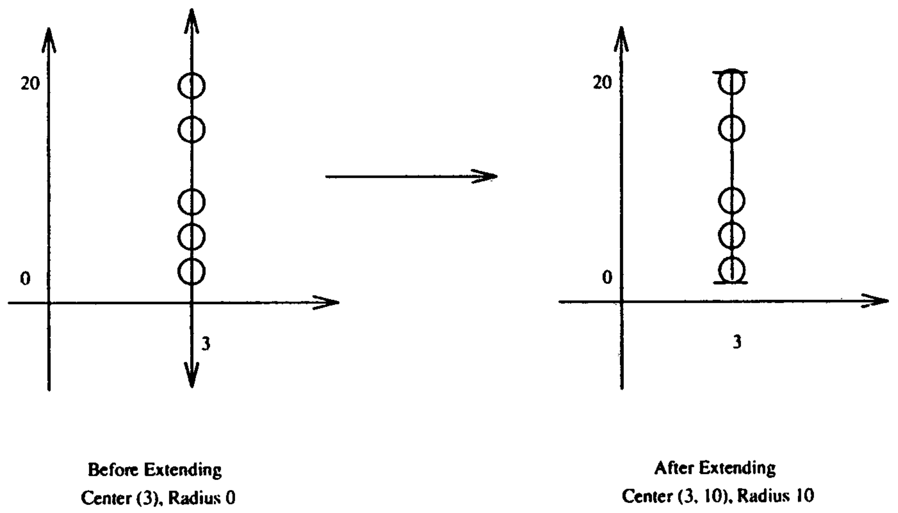
Deletion. Deletion is straightforward, unless it causes an underflow. In this case, the remaining branches of the node are deleted and re-inserted. The underflow may propagate upwards.

Extending and Contracting. As previously mentioned, extending and contracting of 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, extension will occur (i.e., new active dimensions will be introduced and those on which all the objects agree will be rendered inactive).

An example of extending diamonds is given in Figure 7. After extension, the diamond extends only along the y-dimension.

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

Figure 7. Extending a TMBR (diamond), with $\alpha = 1$



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. To find the closest matches in the presence of typing errors, the queries were exact match and range queries. 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 edit 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 matrix is defined as follows:

1. UNIX is a registered Trademark of Novell, Inc.

2. Actually, we are using the 32-dimension Hadamard Transform matrix (Hamming, 1977) and padding extra 0s to the feature vectors.

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

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

The *TV*-trees in the experiment used 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 was chosen in order to provide a fair comparison for insertion behavior.

Experiments on 2,000 to 16,000 words were 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 (i.e., the query word was found in the dictionary), using half of the database words as query points. Experiments with unsuccessful searches gave similar results and are omitted. We also issued range queries with the words randomly drawn from the dictionary, (the number of queries is 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 enough 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 tried to determine a good value for α. Different numbers of active dimensions of the TV-tree were tried. The results are shown in Figures 8 through 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 that the whole tree (except the root) was stored on the disk and no buffer for the internal levels was available. The results are similar.

The results indicate that $\alpha = 2$ gives the best results, because 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 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, given 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.

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

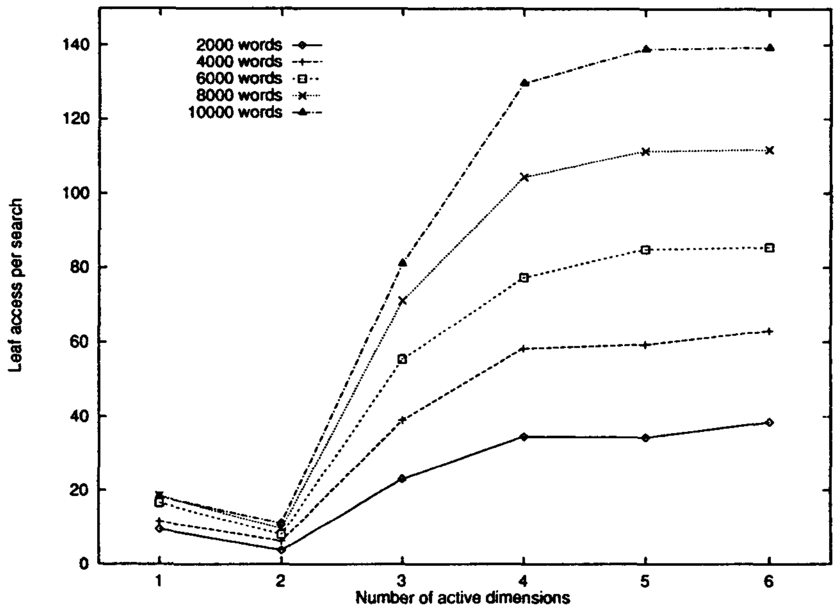


Figure 9. Range queries (tolerance=1)(# of leaf accesses vs. α)

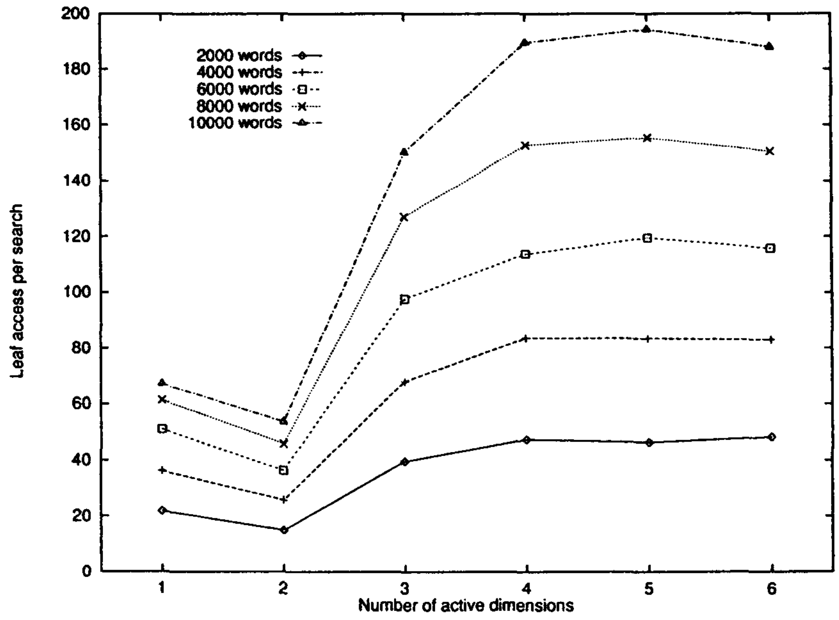


Figure 10. Range queries (tolerance=2)(# of disk accesses vs. α)

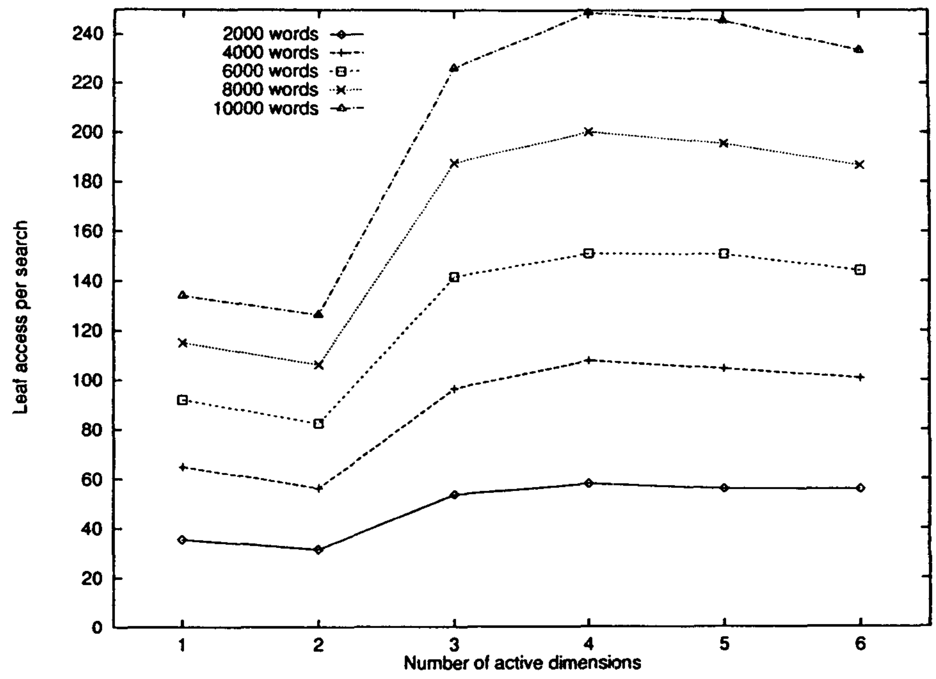


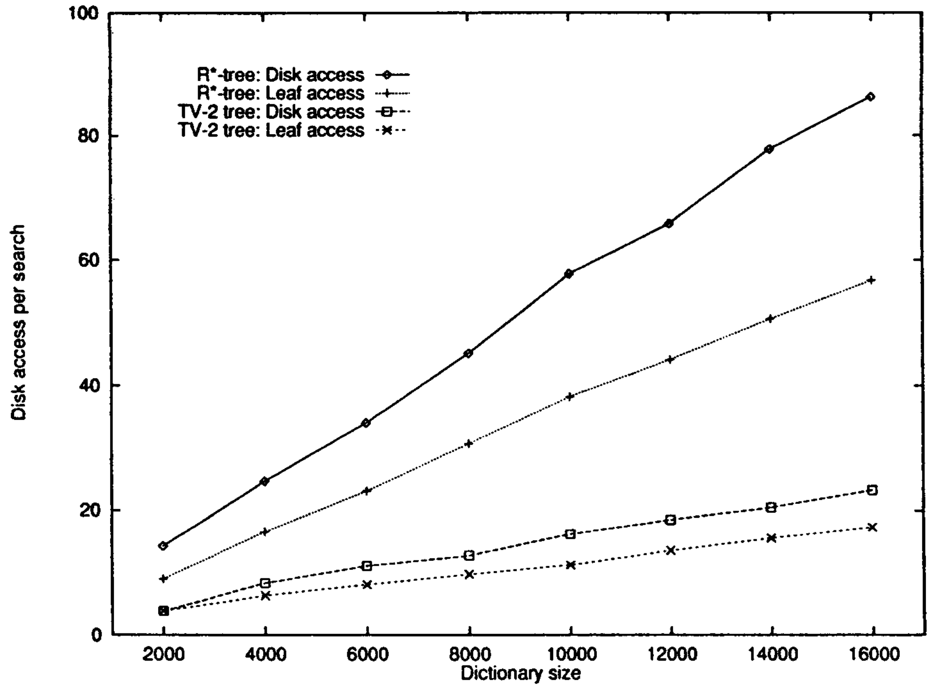
Table 1. Disk access per insertion – object size 100 bytes

Dictionary size	Disk access per insertion	
	R^*-tree	TV-2 tree
4,000	5.25	4.75
8,000	5.51	5.21
12,000	6.19	5.28
16,000	6.50	5.35

5.3 Comparison with R^*-Tree

Index Creation. We measured the number of disk accesses (read + write) needed to build the indexes. We assumed that every update of the index would be reflected on the disk. We found 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 and, thus, fewer nodes need to be retrieved and fewer potential updates need to be written back to disk. Table 1 shows the result for object size 100 bytes with a 4K page size.

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



The big jump between 4,000 and 8,000 for the TV-2 tree is because of an introduced addition level. However, the TV-2 tree still has one level fewer than the R^*-tree. Thus, the increase in disk access for the TV-2 tree is slower after the introduced level.

Search. The next set of experiments compared the proposed *TV*-tree with the R^*-tree. Figures 11 through 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 4,000 page size was used. The following results are for objects of size 100 bytes.

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 increased with the size of the database, indicating that our proposed method scales up well. As the database size increased 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 $\epsilon=1$; and from 33% to 42% for range queries with $\epsilon=2$.

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

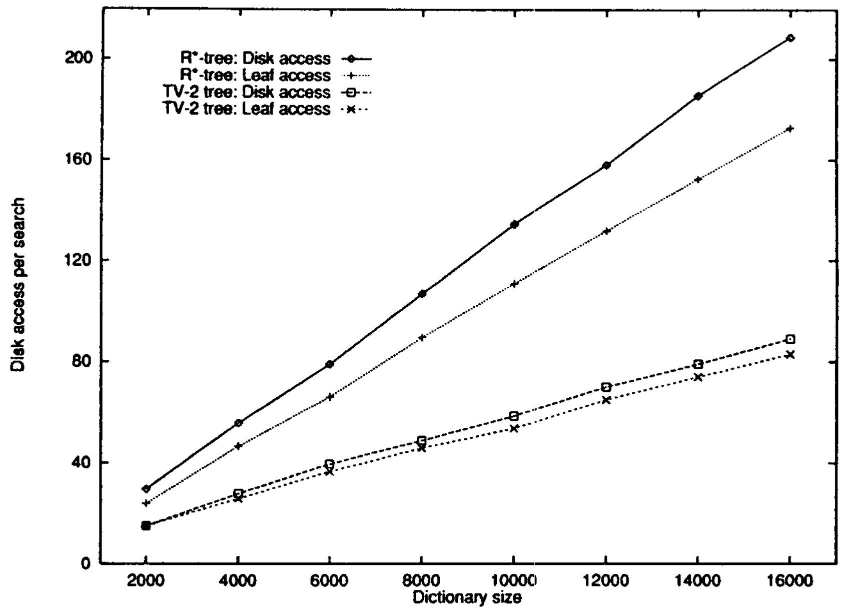


Figure 13. Disk/leaf accesses vs. db size–range queries (tolerance=2)

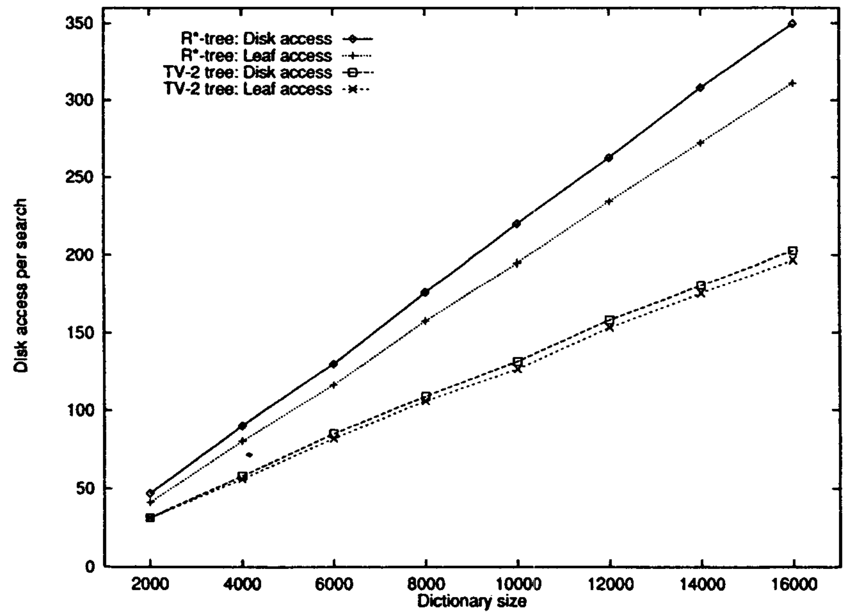
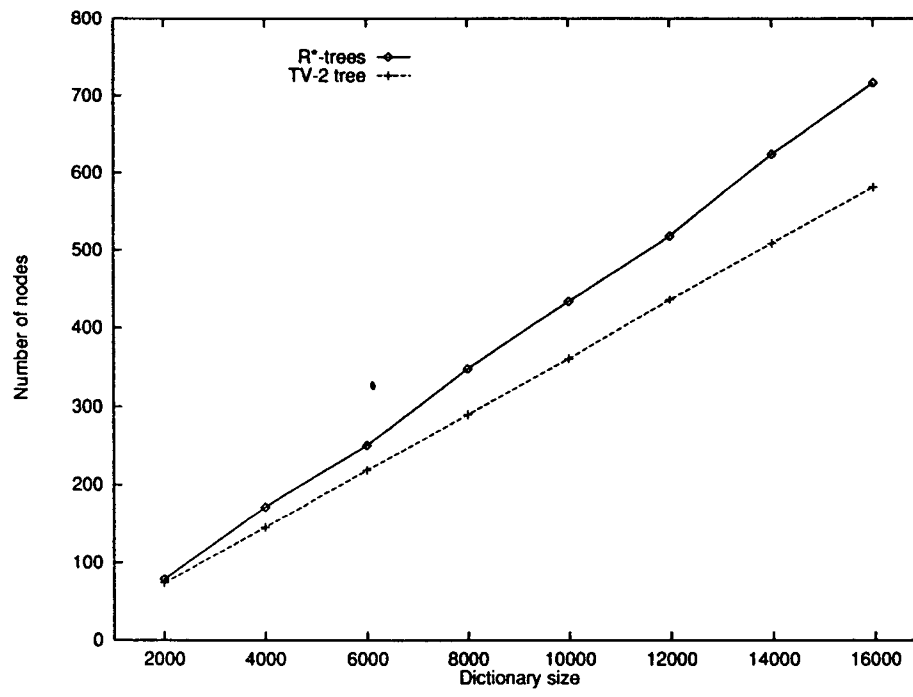


Figure 14. Comparison of space requirements



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

We also experimented with various sizes of database objects. Our method showed more significant improvement when object size is small. As object size increases, the leaf fan-out decreases, making the *TV*-tree grow faster, and offsetting some of its advantages. However, even with object size 200, we still have improvement of around 60% over R^*-trees for exact match and 40% for range queries with $\epsilon=2$.

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

6. Conclusions

In this article, we proposed the *TV*-tree as a method for indexing high dimensional objects. The benefit lies in its ability to adapt dynamically and use a variable number of dimensions to distinguish between objects or groups of objects. Since this number of required dimensions is usually 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 for choosing the design parameters (e.g., optimal active dimension $\alpha = 2$, minimum fill factor $= 45\%$). 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%, 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

This research was partially funded by the Institute for Systems Research, and by the National Science Foundation under grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software, Inc. and Thinking Machines, Inc. The authors thank Alexios Delis and Ibrahim M. Kamel for their help.

References

Agrawal, R., Faloutsos, C., and Swami, A. Efficient similarity search in sequence databases. *FODO Conference,* Evanston, IL, 1993.

Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. A basic local alignment tool. *Journal of Molecular Biology,* 215(13):403-410, 1990.

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

Arya, M., Cody, W., Faloutsos, C., Richardson, J., and Toga, A. Qbism: A prototype 3-D medical image database system. *IEEE Data Engineering Bulletin,* 16(1):38-42, 1993.

Aurenhammer, F. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys,* 23(3):345-405, 1991.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD,* Atlantic City, NJ, 1990.

Bentley, J.L., Weide, B.W., and Yao, A.C. Optimal expected-time algorithms for closest-point problems. *ACM Transactions on Mathematical Software,* 6(4):563-580, 1980.

Brinkhoff, T., Kriegel, H.-P., and Seeger, B. Efficient processing of spatial joins using *R*-trees. *Proceedings of the ACM SIGMOD,* Washington, DC, 1993.

Chatfield, C. *The Analysis of Time Series: An Introduction.* London: Chapman and Hall, 1984. Third edition.

Friedman, J.H., Baskett, F., and Shustek, L.H. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers,* C-24(10):1000-1006, 1975.

Fukunaga, K. *Introduction to Statistical Pattern Recognition.* New York: Academic Press, 1990.

Fukunaga, K. and Narendra, P.M. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers,* C-24(7):750-753, 1975.

Greene, D. An implementation and performance analysis of spatial data access methods. *Proceedings of Data Engineering,* Boston, MA, 1989.

Guttman, A. R-trees: A dynamic index structure for spatial searching. *Proceedings of the ACM SIGMOD,* 1984.

Hamming, R.W. *Digital Filters.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

Hartigan, J.A. *Clustering algorithms.* New York: John Wiley & Sons, 1975.

Hoel, E.G. and Samet, H. A qualitative comparison study of data structures for large line segment databases. *Proceedings of the ACM SIGMOD Conference,* San Diego, CA, 1992.

Hunter, G.M. and Steiglitz, K. Operations on images using quad trees. *IEEE Transactions on PAMI,* 1(2):145-153 (1979).

Jagadish, H.V. Spatial search with polyhedra. *Proceedings of the Sixth IEEE International Conference on Data Engineering,* Los Angeles, CA, 1990.

Jagadish, H.V. A retrieval technique for similar shapes. *Proceedings of the ACM SIGMOD Conference,* Denver, CO, 1991.

Kamel, I. and Faloutsos, C. Hilbert *R*-tree: An improved *R*-tree using fractals. Systems Research Center (SRC) TR-93-19, University of Maryland, College Park, MD, 1993.

Kukich, K. Techniques for automatically correcting words in text. *ACM Computing Surveys,* 24(4):377-440, 1992.

Mandelbrot, B. *Fractal Geometry of Nature.* New York: W.H. Freeman, 1977.

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

Narasimhalu, A.D. and Christodoulakis, S. Multimedia information systems: The unfolding of a reality. *IEEE Computer,* 24(10):6-8, 1991.

Niblack, W., Barber, R., Equitz, W., Flickner, M., Glasman, E., Petkovic, D., Yanker, P., Faloutsos, C., and Taubin, G. The qbic project: Querying images by content using color, texture, and shape. *SPIE 1993 International Symposium on Electronic Imaging: Science and Technology Conference 1908, Storage and Retrieval for Image and Video Databases,* San Jose, CA, 1993. Also available as IBM Research Report RJ 9203 (81511), 1993.

Nievergelt, J., Hinterberger, H., and Sevcik, K.C. The grid file: An adaptable, symmetric, multikey file structure. *ACM TODS,* 9(1):38-71, 1984.

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

Ruskai, M.B., Beylkin, G., Coifman, R., Daubechies, I., Mallat, S., Meyer, Y., and Raphael, L. *Wavelets and Their Applications.* Boston: Jones and Bartlett Publishers, 1992.

Salton, G. and Wong, A. Generation and search of clustered files. *ACM TODS,* 3(4):321-346, 1978.

Samet, H. *The Design and Analysis of Spatial Data Structures.* Reading, MA: Addison-Wesley, 1989.

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

Wallace, G.K. The jpeg still picture compression standard. *CACM,* 34(4):31-44, 1991.

Appendix

A. 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 dimensions. This means that all points inside the diamond will have coordinates that start with (1,0,8,7,. . .).

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.

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

Algorithm 4. Finding the MBD

```
begin
    /* $\alpha$ is the number of active dimensions */
    Proc TMBD(Array of Diamonds $D$, integer $\alpha$);
    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 $Startproject \leftarrow m + 1$
         Set $Startproject \leftarrow min - \alpha + 1$ /* special case when some diamonds have
         small dimensionality. This step is to ensure that there will be $\alpha$ active
         dimensions */
    4. Project each diamond to dimensions $Startproject \ldots Startproject + \alpha - 1$,
       setting the radius to 0 if none of the projected dimension is active,
       otherwise retain the original radius.
    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 has boundary (0.5, 8) along the first dimension, and (2, 7.5) along the second. Its center is (4.25, 4.75). The radius required to cover all three diamonds is 6. Thus, the final TMBR has center (1, 0, 4,25, 4.75) and radius 6.

B. Telescoping Without Truncation

Given a feature vector of length n, its contraction to length m is achieved through multiplication by the matrixA_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), (2 m - 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 1s 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.