Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances

Ada Wai-chee Fu, Polly M. S. Chan, Yin-Ling Cheung, Y.S. Moon

Department of Computer Science and Engineering
The Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

Abstract

For some multimedia applications, it has been found that domain objects cannot be represented as feature vectors in a multidimensional space. Instead, pair-wise distances between data objects are the only input. To support content-based retrieval, one approach maps each object to a k-dimensional (k-d) point and tries to preserve the distances among the points. Then existing spatial access index methods such as the R-trees and KD-trees can support fast searching on the resulting k-d points. However, information loss is inevitable with such an approach since the distances between data objects can only be preserved to a certain extent. Here we investigate the use of distance-based indexing method. In particular we apply the vantage point tree (vp-tree) method. There are two important problems for the vp-tree method that are not studied in depth, the n-nearest neighbors search and the updating mechanisms. We propose an n-nearest neighbors search algorithm, which is shown by experiments to scale up well with the size of the dataset and the desired number of nearest neighbors, n. Experiments also show that the searching in the vp-tree is more efficient than that for the R^*-tree and the M-tree. Next, we propose solutions for the update problem for the vp-tree, and show by experiments that the algorithms are efficient and effective. Finally, we investigate the problem of selecting vantage-point, propose a few alternative methods and study their impact on the number of distance computation.

Index Terms: Content-based retrieval, indexing, multimedia data, nearest neighbor search, updating, similarity metric, pair-wise distances, R^*-tree, M-tree, vp-tree.

1 Introduction

With the advent of large-scale multimedia database systems, there is a need to efficiently answer user queries. *Content-based retrieval* is typically required [18]. One advantage of such an approach is that it bypasses the difficult problem of specifying the desired multimedia objects in terms of formal query languages. A popular form of content-based queries employs the query-by-example paradigm. For example, in a collection of images, users can use existing images as query templates and ask the system for images similar to the query images. This is the so-called "like-this" query. Alternatively, the user can sketch a picture that serves as the query template.

To support content-based retrieval, often we have to rely on feature extraction capabilities to map each domain object into a point in some k-dimensional space where each object is represented by k chosen features. An example feature vector may be color components of an image or shot cuts of a video clip. Hence, processing content-based queries typically requires some measurement of similarity between k-dimensional points. The similarity (or distance) between two objects is measured using some metric distance function over the k-dimensional space. The most common metric distance function is the Euclidean distance. The entire problem is then formulated as storing and retrieving k-dimensional points, for which there are many fine-tuned indexing methods available. In general, these methods are called *multidimensional indexing* or *Spatial Access Methods* (SAMs) [30]. Some of the previous works are [2, 3, 4, 12, 13, 15, 19, 21, 25, 26, 28, 35, 38, 34].

It has been found that the above setting cannot be applied to certain applications. For example, [24] cites the example of typed English words, where the similarity function is defined by the minimum number of insertion, deletions and substitutions to transform one string to another, and the example of of matching digitized voice excerpts, which include the consideration of time warping [31, 29]. The time warping problem occurs also in the similarity search in time series [5, 1]. For other examples, [27] describes a method of measuring the similarity between color images based on a color similarity matrix which takes into account the perceptual distance between different pairs of colors. For shape similarity, a natural definition is the value of "area difference", for which we measure the area where two shapes do not match when one is *placed on top of* the other, e.g. one can obtain a pixel-wise exclusive-or of the two shapes. Another method represents a shape in terms of its boundaries, and a string-edit-distance-based similarity measure is employed based on the number of changes required to transform one to the other [17]. In still other applications, it may be relatively easier for a domain expert to assess the similarity or distance between two objects rather than giving a computable definition of similarity [37]. In all of the above applications, we are given the similarities or distances between pairs of data. In most cases, the distances are *metric*, meaning that the triangle inequality property applies, and this is the assumption we make. Given only the distance information, we still need an indexing method to support query and update.

For this problem, one approach uses the distance information to deduce k-dimensional points for the objects so that we can subsequently make use of Vector Space Model (VSM) [37] multidimensional indexing methods such as the R-tree and its variants. The *FastMap* algorithm [24] and *Multidimensional Scaling* [23] fall in this category. The main challenge for this approach is to preserve distances as well as possible. It is difficult to decide on a value of k and then map each domain object into a k-dimensional point while still accurately representing the similarity between objects using some metric in that k-dimensional space. In this paper, our experimental results show that such an approach can incur a considerable amount of inaccuracy for n-nearest neighbor search.

Therefore we study an alternative approach that uses distance-based indexing, known as Metric Space Model (MSM) indexing. In particular we examine the *Vantage-Point tree* (vp-tree) method [36, 39, 10]. This approach can obviously save the overhead of inferring points in a multidimensional space, and can also avoid the difficulty in preserving distances. Our main contributions are the following.

1. We propose an n-nearest neighbor search algorithm for the vp-tree index method. From our experiments, the n-nearest neighbor search algorithm demonstrates promising performance. In the detailed implementation, we also suggest and implemented a method for the physical clustering of vp-tree nodes. We compared the costs of the n-nearest neighbor search with R^*-tree and M-tree by experiments, and show that the search of vp-tree is considerably more efficient.
2. The update problem has also been left open for the vp-tree and its variants [6]. We propose mechanisms for update operations on the vp-tree. We investigate two alternatives in the insert operation: split-first and redistribute-first techniques; and two alternatives in the delete operation: merge-first and redistribute-first. All of these techniques preserve the balanced-tree property of the vp-tree. The split-first and merge-first strategies can also preserve the original characteristics of the vantage points and are found to be more efficient. We show by experiments that the updating cost is not particularly expensive.
3. Finally we propose some alternative techniques for the vantage point selection which can reduce the number of distance computation. Experiments were carried out to demonstrate that the proposed methods are efficient and effective.

The rest of the paper is organized as follows. Previous related work is described in Section 2. In Section 3, we give experimental results of the distance-preserving approach. Section 4 details the proposed algorithm for n-nearest neighbor search in vp-trees, along with performance results. Update mechanisms for the vp-tree are outlined in Section 5 and comparison of different approaches by experiments is given therein. Section 6 discusses some possible improvements in the selection of vantage points, with experimental results. We conclude the paper in Section 7 with an outline of future work.

2 Related Work

Given only pair-wise distances between objects, there are two major approaches to the indexing methods for content-based retrieval: the distance preserving transformation based methods, and the distance-based indexing methods. We shall briefly describe both approaches.

2.1 Distance Preserving Methods

With the distance preserving approach, we try to deduce for each object a corresponding point in a multi-dimensional space so that the distances among objects are preserved as much as possible. One example is a method from pattern recognition, namely, Multidimensional Scaling (MDS) [23]. Another is the FastMap algorithm proposed by Faloutsos and Lin [24]. As experiments in [24] showed that FastMap achieves dramatic computational savings over known MDS methods, without loss in quality of the results, we shall focus only on the FastMap method. Thus, the details of other MDS methods are omitted for brevity.

The FastMap algorithm assumes that objects are points in some unknown high-dimensional space, and projects these points on k mutually orthogonal directions (k being user-defined), such that objects are mapped to points in this k-dimensional space. One important requirement that FastMap must fulfill is to preserve distances as much as possible such that the Euclidean distances between the points in the k-dimensional space match the pair-wise distances given. If the overall distances are not preserved sufficiently, some of the information that distinguishes the objects cannot be maintained. After the mapping, any one of a number of highly fine-tuned spatial access methods like the R-tree, etc. [2, 4, 19, 25, 28, 33] can be

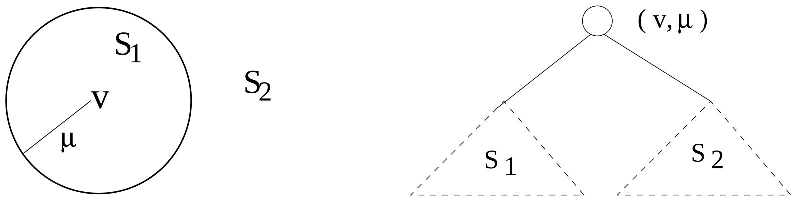


Figure 1: Partitioning mechanism of the vantage-point tree method.

employed to provide fast searching for range queries[1] and n-nearest neighbor queries[2].

FastMap introduces pre-processing costs to both index construction and querying because all domain objects and the query object must first be mapped to the corresponding k-d points before an index structure is built or a query is processed. The mapping of N objects into N k-d points requires k recursive calls to the function called FastMap. For example, if our target is to deduce 2-d points for N objects, FastMap will determine the coordinates of the N objects on one axis in the first recursive call, and those on the other axis in the second recursive call.

2.2 Distance-Based Index Structures

Quite a number of distance-based indexing structures have been proposed. A summary of some of these methods can be found in [6, 7]. Previous work includes techniques suggested in [8], which contains some of the basic ideas for later methods, the *generalized hyperplane tree* (gh-tree) [36], the *vantage point tree* (vp-tree) [36, 39, 10], the *Geometric Near-neighbor Access Tree* (GNAT) [7], the mvp-tree [6] which is a variation of the vp-tree, and the M-tree [11]. In [7] the GNAT is compared to the binary vp-tree in a set of experiments and is found to incur more expensive construction but less number of distance computations in the range querying. The mvp-tree uses pre-computed distance to reduce the number of distance computation, and also uses multiple vantage points in a tree-node. In [6] it is shown by experiments that the mvp-tree would incur less distance computations in range querying compared to the vp-tree. However, n-nearest neighbor search has not been considered for GNAT in [7] or for the vp-tree or its variations, and both [6] and [7] focus only on the number of distance computations for range queries.

The M-tree is a balanced tree which is able to deal with dynamic data. In an M-tree, each internal node has a routing object, and all objects in the subtree under the node are within a certain distance from the routing object. Updating is allowed and may trigger node splitting. Experimental results show that it is a competent access method. Therefore we shall compare our approach with M-tree.

2.2.1 The Vantage-Point Tree Method

Since we make use of the vp-tree structure [39, 10], we describe in more details the vp-tree method. Consider a finite set S of N data points [3]. For each node in the tree, a particular data object is selected to be the vantage point, according to a randomized algorithm given in [39] as shown in Figure 2. Let the point chosen

[1]Example: Find objects that are within distance ϵ from the query object.

[2]Example: Find the n ($n \geq 1$) objects that are closest to the query object.

[3]Since vp-tree uses the term *point* to refer to data objects, we shall use the terms data points and data objects interchangeable.

```
Pick a set of candidate vantage points from the data set;
For each vantage point
  Pick a set of sample objects from the data set;
  Compute the distance values from the vantage point to each of the sample objects;
  Calculate the mean and the standard deviation of these distance values;
Endfor
Choose the candidate vantage point with the maximum standard deviation.
```

Figure 2: Algorithm Choose_Vantage_Point.

for the root node be v. Then, let μ be the *median* of the distance values of all the other points in S with respect to v, S is partitioned into two subsets of approximately equal sizes, S_1 and S_2, defined as:

$$\begin{aligned} S_1 &= \{s \in S \mid d(s,v) < \mu\} \\ S_2 &= \{s \in S \mid d(s,v) \geq \mu\} \end{aligned}$$

where $d(p,q)$ is the distance between points p and q. Figure 1 illustrates the concept. This partitioning procedure is then applied to S_1 and S_2 recursively. Every subset, such as S_1 and S_2, corresponds to one node of the vp-tree. At each node, a distinct vantage point is chosen to partition the data points in the corresponding subset. At the leaf node, we store a number of data points. Eventually, the entire data set is organized as a balanced tree as in other spatial index structures.

The m-ary vp-tree construction is similar to the case of binary vp-trees. The data set S is split into m subsets, S_i, $i = 1$ to m, according to the distance values between the chosen vantage point and other data points. Each of S_i's has roughly the same number of data points. μ_i is used to denote the boundary distance value, so that for all $s \in S_i$, $\mu_{i-1} < d(s,v) \leq \mu_i$. Again, each of S_i's is recursively partitioned into smaller subsets using the same partitioning mechanism.

Note that when we discuss updating on the vp-tree, the branching factor of a vp-tree node can be reduced after deletion, we shall specify a minimum branching factor. The m value in the above is the maximum branching factor. For example, if the minimum branching factor is 2, then we have an 2-to-m-ary vp-tree.

The **multi-vantage point tree** (mvp tree) [6] incorporates two mechanisms on top of the vp-tree to reduce the number of distance computation. The first mechanism is to keep pre-computed distances between the data points and the vantage points. The second mechanism is to use more than one vantage point to partition the space into spherical cuts at each level. We shall adopt the first mechanism in our implementation. We shall study more about the second mechanism in Section 6.

3 Motivation

The Vector Space Model methods that try to preserve the distances will have a problem in the accuracy, since some distance information may be lost. Here we show with an experiment the inaccuracy that may result from such a method. We implemented FastMap and the R-tree in C under UNIX on a SPARCcenter 2000. We performed tests on 8-nearest neighbor queries relative to points chosen from the dataset. The average is taken over the performance for 15 randomly chosen query points. The R-tree that we implemented is able to handle n-nearest neighbor search. In this experiment, a synthetic dataset is used. We generated a dataset of 1500 points in 10-dimensional space. The points form 10 clusters, with the same number of

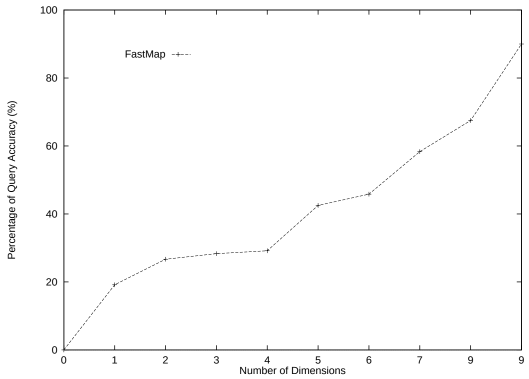


Figure 3: Query accuracy vs. number of dimensions for the dataset of 1500 objects.

points in each cluster. Centers of clusters are uniformly distributed and the distances of the points in each cluster from the centers follow a normal distribution. Our input is the Euclidean distances between such data points.

To study the accuracy of n-nearest neighbor search resulting from the FastMap method as the dimensionality k increases, we varied k from 1 to 9. 8 nearest neighbors are to be found for each of 15 query objects. k-d points were deduced, for $k = 1$ to 9. Each set of points was then organized in an R-tree. It was also necessary to map 15 query objects to k-d space. The resulting k-d queries were submitted to the R-trees whose search algorithm was run to obtain the results.

We expressed the accuracy of an n-nearest neighbor search as the percentage of the n answers that are indeed among the n nearest neighbors of the query object. We reported an average of accuracy over the 15 queries. Figure 3 plots the percentage of query accuracy as a function of the number of dimensions. With FastMap, the lower the dimensionality, the more nearest neighbors have been missed. It is mainly because FastMap may not preserve the actual distances between objects and the preservation problem gets worse when k is getting smaller.

There are enhancements proposed on the R-tree based methods, such as the multi-step k-Nearest Neighbor Search method [32, 22]. However, these are enhancement on the efficiency of the method and would not help in enhancing the accuracy of the method when the data is subjected to the above information loss.

3.1 Discussion

FastMap suffers from the difficulties in preserving actual distances and in determining a proper k value to achieve high accuracy, Therefore, we propose to use methods that are able to locate every nearest neighbor with a fast response. We have chosen the vp-tree approach. The advantages of the vp-tree approach mainly lie in the following:

1. There is no need to infer multidimensional points for domain objects before an index can be built. Instead, we build an index directly based on the distances given. This avoids pre-processing steps. There are two major problems with the pre-processing in the FastMap approach:
   (a) The computation involved in these steps can be costly.
   (b) It is difficult to determine the number of dimensions, k, that can preserve the distances to a

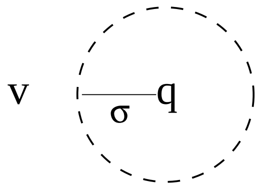


Figure 4: The meaning of the threshold σ.

satisfactory level.

2. The updates on the vp-tree are relatively easier than that for the Fastmap method. For the Fastmap method, after a certain amount of data objects are inserted or deleted, the mapping for the data points will no longer be as distance-preserving as before. There will be a point when Fastmap has to be executed once again for all the objects, and it is not clear how to determine what is a good time for the re-construction. By comparison, the updates for vp-tree are much more straight-forward (see Section 5).

3. A distance-based indexing method such as the vp-tree is flexible: it is not only applicable to multimedia objects given pair-wise distances, but is also able to index objects that are represented as feature vectors of a fixed number of dimensions (the case when feature extraction functions are available). We also show that the performance is much better than R*-tree and M-tree in terms of page accesses for n-nearest neighbor search.

4 n-Nearest Neighbor Search in vp-trees

For content-based retrieval, it is rare to have exact match on multimedia data, so nearest neighbor queries are more desirable. The search methods for the vp-tree presented in [39, 10] locate a single nearest neighbor. In practice, users often ask for a certain number of objects similar to a given query object so that they can select part of the returned collection. Therefore, we aim at finding n nearest neighbors to a query object, where n is usually greater than one. Here we describe an algorithm for n-nearest neighbor search in the vp-tree.

The single-nearest-neighbor search algorithm in [10] relies on a specific threshold, σ, which estimates an upper bound on the distance between a query object and its nearest neighbor. Let $d(p,q)$ be the distance between points p and q. Given a σ value, the algorithm in [10] will look for the single nearest neighbor to q within the range $d(v,q) \pm \sigma$ (see Figure 4). Recall that at each node of the vp-tree, a vantage point determines m subsets S_i by the distances μ_i. If the hypersphere depicted in Figure 4 falls inside the boundary of only one subset S_i, the algorithm only needs to explore that particular subset. Otherwise, multiple subsets (subtrees) need to be explored. A tighter value of σ will ensure that less subtrees are explored, but it also increases the chance that no nearest neighbor is found, in which case the search has to be repeated with a greater value of σ. In the algorithm in [10] there is no specific bound on the number of trials (passes) with increasing values of σ.

We generalize the above mechanism for n-nearest neighbor search. We also use a value of σ in a similar way. The initial value of σ is set to be infinitely large. The value of σ is dynamically improved as we discover shorter distances between the query point and some data points that we come across during the search.

```
Procedure nNN_Search(q,n,root)
Input: the query point q, the number of nearest neighbors requested n,
       the root node of a vp-tree.
Output: the set W of n nearest neighbors to q.

begin
  Local variable σ : to record the distance of q to the n-th nearest neighbor discovered so far.
  Search(q,n,root,σ,W);
  return W;
end
```

Figure 5: Procedure nNN_Search.

For simplicity we focus on the binary partitioning case, but the discussion can easily be generalized to the case of an m-ary vp-tree, for $m > 2$. The construction algorithm for the vp-tree is similar to the original vp-tree construction algorithm in [39]. The pseudocode for the algorithm is given in Figure 5. This algorithm triggers the procedure call Search(q,n,root,∞,ϕ). The procedure Search (Figure 6 [4]) is recursively activated one or more times. During the traversal, σ is dynamically adjusted to the value of the distance between the query object and the current n-th nearest candidate. This σ guarantees that the n nearest neighbors will at most be at distance σ from the query point. Effectively, we perform a depth first search, and let σ be determined by the distances of the actual data objects at the leaf nodes that we encounter in the search.

```
Procedure Search(q,n,node,σ,W)
Input: the query point q, the number of nearest neighbors requested n, a node of the vp-tree,
       σ = the distance of n-th nearest points discovered so far
       and W is a set of n nearest neighbors obtained so far.
Output: updated values of σ and W.
begin
  if node is leaf
    for each element v in node,
      if d(v,q) ≤ σ
        insert v to proper position in W (W always sorted in ascending order of d(w,q));
      if |W| = n then σ := d(w_n,q);
  else
    dist := d(node↑.v,q);
    if dist < node↑.mu
      if {dist < nodes.mu + σ} then Search(q,n,node↑.left, σ,W);
      if {dist ≥ nodes.mu − σ} then Search(q,n,node↑.right, σ,W);
    else
      if {dist ≥ nodes.mu − σ} then Search(q,n,node↑.right, σ,W);
      if {dist < nodes.mu + σ} then Search(q,n,node↑.left, σ,W);
end
```

Figure 6: Procedure Search.

[4] In Figure 6, node↑.v is the vantage point at the node, node↑.mu is the median value at the node, node↑.left is the pointer to the left child node, and node↑.right is the pointer to the right child node.

We have also tried some other alternative ways of setting an initial σ that is not infinite but our experimental results show that setting the infinite initial value can achieve better performance. Note that instead of a depth first traversal, we could adopt the method that is proposed in [11] where the best traversed node and its subtree is searched in each iteration. This could lead to better pruning power. However, the current method is simpler and our experimental results show that it is already superior to the R^*-tree and the M-tree.

4.1 Performance Evaluation

To study the performance of our proposed n-nearest neighbor search algorithm for the vp-tree, we implemented the vp-tree and the search algorithms in C under UNIX on an UltraSPARC. In the experiment, we compared the vp-tree with the M-tree and the R^*-tree. We implemented the algorithms for the R^*-tree by Berchtold, Keim and Kriegel [4], which is able to support n-nearest neighbor queries, and enhanced it with the method proposed in [9]. We also implemented the n-nearest neighbor search algorithms for the M-tree. Next we describe the setup, as well as our results and observations.

4.1.1 Experimental Setup

We used 2 synthetic datasets and 1 real dataset. The synthetic datasets are similar to the ones used in [37] and the details of them are:

- Clustered 10,20,30,40,50D: sets of 10000, 20000, ..., 100,000 vectors, each consisting of 100 clusters of equal size. Each cluster was centered on a point chosen from a uniform distribution in the interval [0,1] on each dimension and each point in the cluster was uniformly distributed in the interval $[-0.1, +0.1]$ relative to the cluster center in each dimension.
- Uniform 5,10,15,20,25D: sets of 10000, 20000, ..., 100,000 uniformly distributed vectors in the interval [0,1] on each dimension.

The real dataset was provided by Berchtold, Keim and Kriegel [4] and contains about 70 Mbyte of Fourier points of variable dimensionality, representing shapes of polygons. We randomly extracted groups (in sizes of 10000, 20000, ..., 100,000) of points in dimensions of 2, 4, 6, 8, 10, 12, 14 and 16 out of the entire dataset.

Here we provide the details of our disk-based implementation of the vp-tree. In every internal node, we store one vantage point, $m-1$ boundary distance values and m child pointers, where m denotes the branching factor. In a leaf node we keep the actual data objects (feature vectors).

We have carried out some experiments and found that the branching factor affects the performance in such a way that it should neither be too high nor too low. We also note that for a low branching factor (say below 10), the size of a vp-tree node is very small: we need only to keep the vantage point, which is either an object id or a feature vector, a median distance value, and the pointers to the child nodes. This is much smaller than that for a node of an R^*-tree for the same dimensionality of data points. Therefore we can fit a number of nodes in a vp-tree in a page (a disk block). We suggest to do the following: starting from the root, and scanning down the tree, group the next l levels of the subtree under it into a page. This is repeated recursively with the lowest level nodes stored in the page. Suppose we have a branching factor of 5, and l is 1, then we have altogether 5 vantage points to be kept in the page, and the number of child node pointers at the bottom of this subtree in the page is $5^2 = 25$. Essentially, we can treat the content of the page as a supernode which has a branching factor of 25 (see Figure 7). The dotted box in Figure 7 indicates the contents in a page. Since a search down the vp-tree typically traverses from a parent node to one or

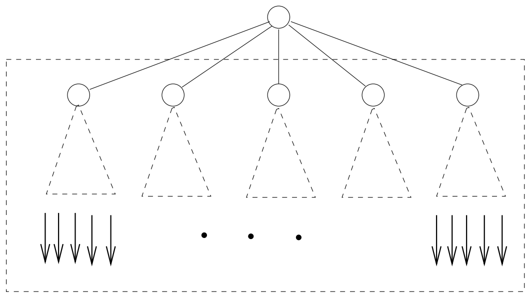


Figure 7: The clustering of vp-tree nodes in a page.

more of its child nodes, the above clustering of the vp-tree nodes can help us achieve a page access rate of only a fraction of that of the node access rate. Some preliminary experiments show that this set up can achieve good performance in terms of the amount of page accesses in n-nearest neighbor search, compared to some other strategies.

Given a value of l, the maximum branching factor of internal nodes and the maximum number of data objects contained in a leaf are determined by the page size. Tables 1 and 2 list the parameters that we use in the calculations. A 4K page size is used, and we assume vantage points and data objects are represented as feature vectors in dimensions of D, each dimension occupying a 4-byte float. We have done some experiments to estimate the optimal number of branching for internal nodes, and found that the best results occurs for the greatest branching factor possible with the restriction that the l levels of subtree (such as that shown in Figure 7) can be held within one disk page. We have chosen to store only one level of a subtree under a node in a page, meaning that the value of l in the above is 1. The branching factor of internal nodes is calculated by finding the value of m that satisfies

$$m = \left\lfloor \frac{page - header}{inode} \right\rfloor$$

Parameters	Descriptions
page	size of a page (4K bytes)
flag	indicator of an internal or a leaf node (1 byte)
no_of_entries	number of internal nodes stored in a page (4 bytes)
header	$flag + no\_of\_entries$ (5 bytes)
vantage_point	represented as a feature vector (4×D bytes)
sigma_factor	used in the *sigma_factor* algorithm (4 bytes)
mu	boundary distance value for partitioning (4 bytes)
pointer	pointer to child node (4 bytes)
inode	size of one internal node, $= vantage\_point + sigma\_factor + (m-1) \times mu + m \times pointer$
m, branching factor of internal nodes	

Table 1: Parameters for calculating the branching factor of internal nodes.

Note that each child pointer that is stored in an internal vp-tree node is a page pointer since l is 1. If l is greater than 1, some pointers may be offsets within the same page. The leaf nodes are different in structure from the internal nodes of the vp-tree, since they keep only the data vectors. A number of data vectors are stored in each leaf node. We maximize the number of data vectors that can be stored when we use one page for each leaf node of the vp-tree. Therefore each leaf node of the vp-tree corresponds to a disk page.

Parameters	Descriptions
$page$	size of a page (4K bytes)
$flag$	indicator of an internal or a leaf node (1 byte)
$no\_of\_entries$	number of data objects stored in a page (4 bytes)
$header$	$flag + no\_of\_entries$ (5 bytes)
$data\_object$	represented as a feature vector (4×D bytes)
max. number of data objects = $\lfloor \frac{page-header}{data\_object} \rfloor$	

Table 2: Parameters for calculating the maximum number of data objects stored in a leaf node.

For the R^* tree, one internal node corresponds to one disk page and we maximize the branching factor according to the page size. For the M-tree, the internal node keeps an data object as the center of the spherical space corresponding to the node, and also a radius of the space. The nodes are also arranged as in the vp-tree, so that all child nodes of a node are stored in a disk page, and the branching factor is maximized according to this arrangement.

We measured the total number of pages accessed per search, assuming the whole tree (except the root) is stored on the disk. All results were averaged over 100 query points that were randomly chosen from the test dataset.

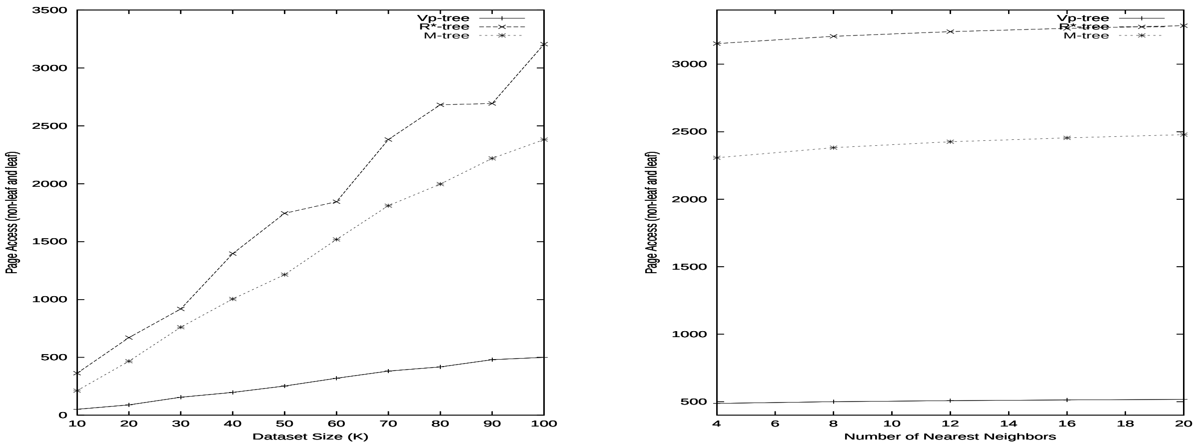
4.1.2 Comparison with the R^*-tree and the M-tree

The partitioning strategies adopted in vp-trees and R^*-trees are different. The vp-tree partitions the search space based on the distances between objects (distance-based indexing), whereas the R^*-tree uses the absolute coordinate values of a multidimensional vector space (feature-based indexing). However, we believe a comparison between the two is significant because distance-based index methods can be applied to both the distance-based case and the vector space case. While R^*-tree is popular, it cannot be used for distance-based indexing. Therefore, if we can show that the vp-tree has comparatively good performance in searching, we can argue that vp-tree is a good choice for content-based retrieval indexing.

We also compare vp-tree with M-tree. M-tree is a distance-based indexing. It is shown to have good n-nearest neighbor search performance and is capable of handling dynamic updates. The node structure of M-tree has similarity with R^*-tree in that the hyperspace occupied by different nodes in the same level of the tree can overlap. This contrasts with vp-tree in which the hyperspace of sibling nodes are disjoint.

All results were obtained by averaging the results of 100 runs of n-nearest neighbor queries. We first tested on the synthetic clustered datasets. Figures 8-9 show the performance of the vp-tree, the M-tree and the R^*-tree (in terms of page accesses) as a function of the dataset size, the number of nearest neighbors, and the dimensionality, respectively. As seen from the figures, the vp-tree consistently outperforms the M-tree and the R^*-tree.

Figure 8(a) plots the results of experiments in which we fixed the dimensionality at 30 and made 8-nearest neighbor queries on varying sizes of datasets. The gap between vp-tree and the other two trees increases with the size of the datasets, indicating that the vp-tree method scales up better. For different numbers of nearest neighbors, we see that the vp-tree makes around 80% fewer page accesses than the M-tree and the R^*-tree. Notice that the increase in the number of nearest neighbors leads to only a small increase in the search effort for the three methods. On the other hand, the dimensionality has much greater impact on the performance. Again the vp-tree has significantly better performance than the M-tree and the R^*-tree. The result also provide further support for the findings made by previous work that R^*-trees stop being efficient for dimensionalities greater than 20 [14, 20, 6].



(a) #nearest neighbors=8 (b) dataset size=30000

Figure 8: Comparison with the M-tree and R^*-tree on synthetic clustered data, dimension=30

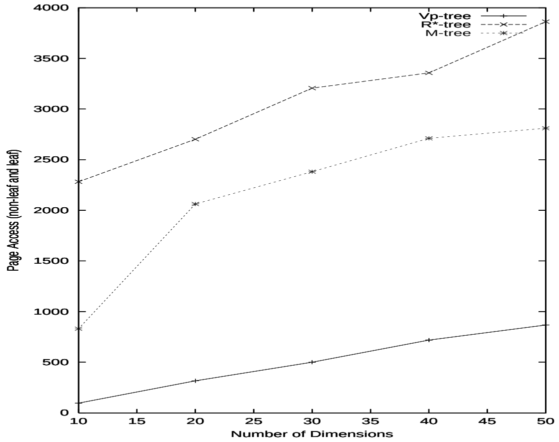
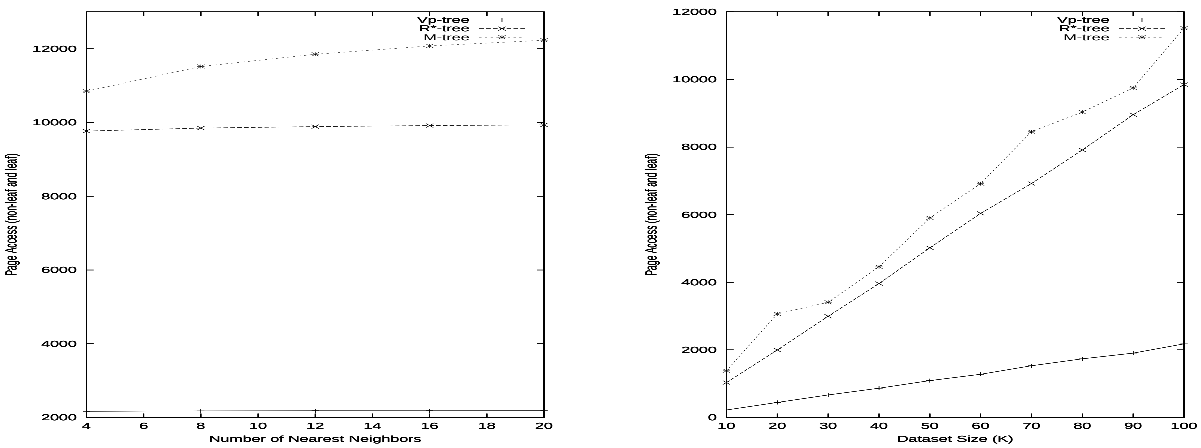


Figure 9: Comparison with M-tree and R^*-tree on synthetic clustered data, dataset size=30000, #nearest neighbors=8.

The reasons why the vp-tree achieves better performance than the R^*-tree are discussed below. (a) The partitioning methods of the vp-tree and the R^*-tree belong to two entirely different approaches. Their n-nearest neighbor search algorithms should accordingly have certain specific properties that make them perform differently. In particular, the R^*-tree has the undesirable property that the tree nodes in the same level of the tree can overlap with each other, and a data point that lies in the overlapping area of two tree nodes can be grouped in either one node. This property triggers many backtracking searches especially for high-dimensional data where the overlapping problem becomes more serious. The vp-tree does not introduce overlapping area among tree nodes in the same level. This can explain why vp-tree is better in performance and the gap in performance increases with the number of dimensions. (b) Since in an R^*-tree two limits for a closed bounded interval are stored on each dimension, the size of an internal node of the R^*-tree is larger than that of the vp-tree. This leads to a lower fanout and a larger tree size, resulting in more cost on querying. (c) The disk blocks used by the vp-tree are highly utilized. As the R^*-tree implementation focuses on other issues, such as the reduction of overlap between bounding boxes, the utilization rate is typically not as high as the vp-tree's.



(a) #nearest neighbors=8

(b) dataset size = 30000

Figure 10: Comparison with M-tree and R^*-tree on synthetic uniform data, dimension=20.

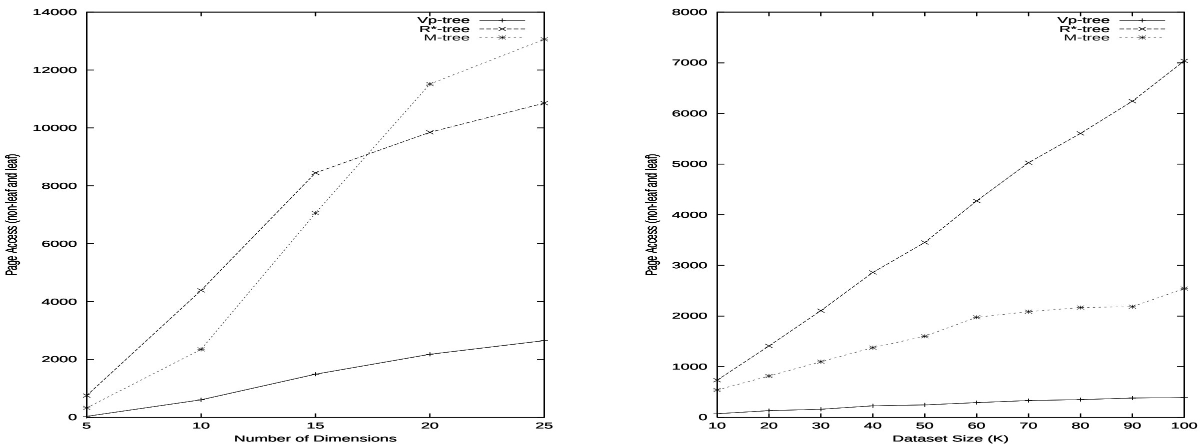
For the M-tree, the disadvantages of having large internal tree nodes is avoided since like the vp-tree, it stores only a center and a distance value to indicate the boundary of a node. However, similar to the R^* tree, it has a problem of overlapping node spaces. This can explain why vp-tree is also much superior to M-tree in searching performance.

Next, our test dataset was the synthetic uniform one. Recall that the dimensions of this set of data are varied from 5 to 25, much lower than those of the clustered dataset. Figures 10-11 show the results. Compared to the results for the clustered dataset, the curves in these figures display considerable similarity in terms of the general trend; that is, the vp-tree outperforms the other two tree structures, and all three structures do not respond strongly to the increase of the number of nearest neighbors. The better performance achieved by the vp-tree can also be explained by the reasons we have mentioned before. However, we can see that the performance is not as good as that for the clustered datasets. This is due to the fact that the data objects in these uniform datasets are distant from each other, making it harder to filter out non-qualifying objects for the n-nearest neighbor search.

The curves for uniform data exhibit a slightly different trend in that the R^*-tree can sometimes perform better than the M-tree. We believe that for clustered data, the shape of clusters are more spherical rather than rectangular, and hence the spherical nodes of the M-tree are superior in handling these clusters. However, this advantage may not hold out for uniformly distributed data.

For the real data, we first studied the dependency on the dataset size. We used datasets of sizes (10000, 20000, ..., 100000) and fixed the maximum dimensionality at 16. Figure 11(b) presents the number of page accesses versus the dataset size. The vp-tree performs much better than the R^*-tree and the M-tree. The gap seems to open up as the dataset size increases. Figure 12(a) gives the performance results for varying numbers of nearest neighbors. The test dataset contained 30000 16-dimensional objects. The vp-tree outperforms both the M-tree and the R^*-tree. The same dataset size (30000 objects) was chosen to experiment on the impact of the dimensionality of data. The result is shown in Figure 12(b). Again the vp-tree performs much better and especially for higher dimensions.

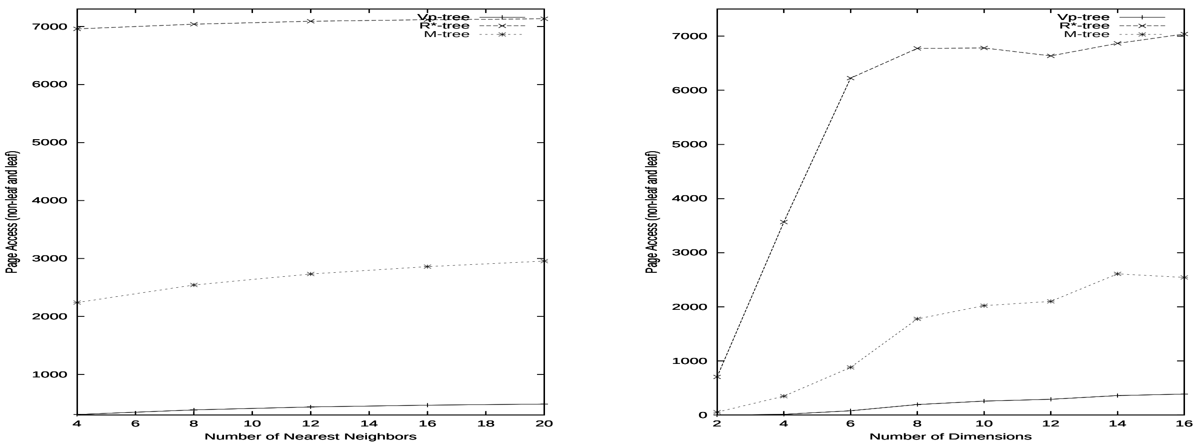
By observing the actual number of page accesses being reported in Figures 11-12, it is clearly seen that the query cost required for the real data is larger than that for the clustered data, but smaller than that for the uniform data. As mentioned before, objects in uniform datasets are distant from each other,



(a) synthetic uniform data, dataset size=30000

(b) real data, number of dimensions = 16

Figure 11: Comparison with M-tree and R^*-tree, dataset size=30000, #nearest neighbors=8.



(a) dimension=16

(b) #nearest neighbor=8

Figure 12: Comparison with M-tree and R^*-tree on real data, dataset size=30000.

that explains why the cost involved in searching through the uniform data is the most. We believe that the original set of real data provided by Berchtold, Keim and Kriegel [4], like most of the real datasets, is correlated or clustered. However, because we randomly selected only a small part from the whole set (containing about 1.3 million objects), the clustering effect could not be fully maintained. Therefore, the search effort for the real data corresponds to somewhere between what the clustered and uniform datasets require.

Distance Computation

One may see that the vp-tree and M-tree search requires distance computation which is expensive. In fact, each node access is accompanied by a distance computation (unless some strategy as discussed in Section 6 is adopted). However, the nearest neighbor search algorithm for R^*-tree also requires the computation of distance (MINDIST, which is roughly the shortest distance between a query point and a minimum bounding box of a tree node) in order to achieve effective hueristic pruning. This computation is also used for each node in the tree. For the R^*-tree, we have used a page for storing each node. For the vp-tree and the M-tree, we used the physical clustering as described in Section 4.1.1, which determines the branching factor. We have used an improved version of the M-tree implementation in that the data-object of a node representing the center of the spherical space is stored as an object-id rather than the entire feature vector. In this way, the branching factor of the M-tree is independent of the number of dimensions and it is always 29 in our experiments. This should enhance the performance of the origianl M-tree method.

We compare the branching factors determined by these arrangements for the R^*-tree, the M-tree and the vp-tree in Table 3. The branching factor of the R^*-tree is the maximum number of minimum bounding rectangles stored in a node or a page, which also determines the number of distance computations needed for each page access. For the vp-tree or M-tree, a node may store a number of vantage points or routing objects up to the branching factor, however, only a fraction of these need to be considered since only the nodes that are within the search range need to be traversed. In our experiments, since the number of page access in the vp-tree search is much smaller compared to R^*-tree and the M-tree, the number of distance computation would also be smaller.

# Dimensions	10	20	30	40	50
R-tree	50	25	16	12	10
vp-tree	20	18	16	14	13

Table 3: Maximum Branching factor for R-tree and vp-tree.

5 Update Operations on VP-trees

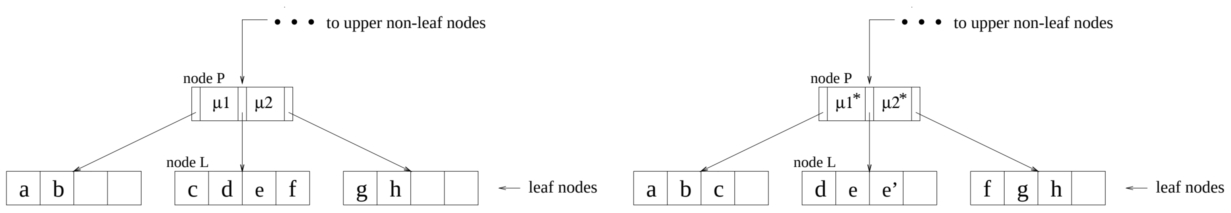
Because of the top-down partitioning strategy, updates of the vp-tree are complex to manage and a global reorganization of the structure may result. Unlike the B-tree and its variants, we cannot conveniently split a node and simply propagate the splitting up the vp-tree. This is because the partitioning at a parent node affects the partitioning at the child nodes, so the effect of a split needs to be propagated downwards. Similar problems also apply to the merging of nodes on deletion. As such, handling update operations that can maintain a balanced tree without substantial restructuring has been left as an open problem.

Here we propose algorithms for doing insertions and deletions on the vp-tree. The maximum branching factor of non-leaf nodes and the maximum number of objects contained in a leaf are determined by the page size as described in Section 4.1.1. For a non-leaf and non-root node, the number of child nodes varies between a minimum value and a maximum value m, as in a B-tree. The minimum value should not be greater than $\lceil m/2 \rceil$, and m should be at least 2. The root can either be a leaf node or have at least 2 and at most m child nodes.

5.1 Insert

To insert a new object, at each level of the vp-tree, we pick a node, the distance d between the associated vantage point and the new object is first computed. We then traverse the tree, choosing the subtree S_i whose distance range covers d, i.e., $\mu_{i-1} < d \leq \mu_i$, until a leaf node L is found. If there is room in L, we insert the new object and the insertion is done. If L is full, we employ the following strategy. Examples are given in Figures 13 to 18.

1. If any sibling leaf node of L is not full, redistribute all objects under P among the leaf nodes (Figure 13).



(a) A new object e' needs to be inserted into L and sibling leaf nodes are not full.

(b) All objects under P have been redistributed, e' gets inserted and boundary distances have been updated

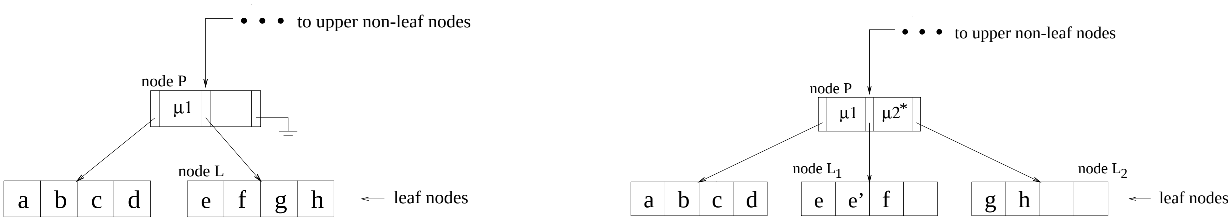
Figure 13: Redistribution among leaf nodes.

Let F be the number of leaf nodes under P. Retrieve all objects stored in the F leaf nodes, and let S be the set of objects retrieved plus the new object being inserted. Order the objects in S with respect to their distances from P's vantage point v. Divide S into F groups of equal cardinality, and let $\mathsf{SS_i}$ be the F subsets, for i=1,2,...,F. Finally update the boundary distance values and pointers stored in P as below.

for $\mathsf{i} = 1$ *to* $\mathsf{F}-1$
    $P\uparrow.\mathsf{mu_i} := (\max\{\mathsf{d}(\mathsf{v},\mathsf{s_j}) \mid \forall \mathsf{s_j} \in \mathsf{SS_i}\} + \min\{\mathsf{d}(\mathsf{v},\mathsf{s_j}) \mid \forall \mathsf{s_j} \in \mathsf{SS_{i+1}}\}) \div 2;$
for $\mathsf{i} = 1$ *to* F
    $P\uparrow.\mathsf{child_i}$:= the leaf node containing $\mathsf{SS_i}$.

2. Else if L has a parent node P and P has room for one more child, split the leaf node L (Figure 14).

   Assume L is the k-th child of P. Retrieve all objects stored in L, and let S be the set of objects retrieved plus the new object. Order the objects in S with respect to their distances from P's vantage point v. Divide S into 2 groups of equal cardinality, and let $\mathsf{SS_1}$ and $\mathsf{SS_2}$ be the two subsets in order. Again, F denotes the number of leaf nodes rooted at P. Then the following pseudocode describes how we shift the boundary distances and pointers of P so as to make room for a new leaf node split from L.



(a) e' needs to be inserted into L and all siblings are full, but the parent node P has room for one more child.

(b) L has been split into nodes L1 and L2, and e' gets inserted.

Figure 14: Splitting of leaf node.

for $\mathsf{i} = \mathsf{k}$ *to* $\mathsf{F}-1$

$\quad P\uparrow.\mathsf{mu}_{\mathsf{i}+1} := P\uparrow.\mathsf{mu}_{\mathsf{i}}$;

$P\uparrow.\mathsf{mu}_{\mathsf{k}} := (\max\{\mathsf{d}(\mathsf{v},\mathsf{s}_\mathsf{j}) \mid \forall \mathsf{s}_\mathsf{j} \in \mathsf{SS}_1\} + \min\{\mathsf{d}(\mathsf{v},\mathsf{s}_\mathsf{j}) \mid \forall \mathsf{s}_\mathsf{j} \in \mathsf{SS}_2\}) \div 2$;

for $\mathsf{i} = \mathsf{k}+1$ *to* F

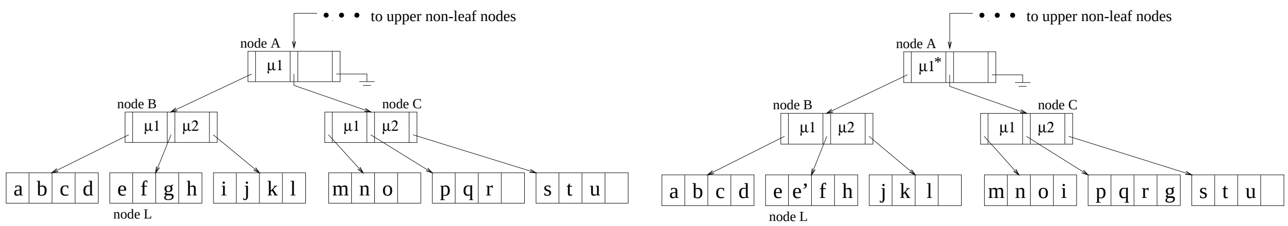
$\quad P\uparrow.\mathsf{child}_{\mathsf{i}+1} := P\uparrow.\mathsf{child}_{\mathsf{i}}$;

$P\uparrow.\mathsf{child}_{\mathsf{k}} :=$ the leaf node containing SS_1;

$P\uparrow.\mathsf{child}_{\mathsf{k}+1} :=$ the leaf node containing SS_2.

3. Else, if we can find a nearest ancestor A of L that is not full. Let B be the immediate child node of A, which is also an ancestor of L.

   (a) If any sibling subtree of B is not full, locate the nearest not-full sibling C and redistribute the objects among the subtrees between B and C inclusively (Figure 15).



(a) e' needs to be inserted into L, the entire B subtree is full, since the sibling subtree C has room, we choose to redistribute objects among B and C.

(b) Assume objects g and i are the farthest with respect to A's vantage point. After redistribution they have been moved to the C subtree and e' gets inserted.

Figure 15: Redistribution among subtrees.

For simplicity, we focus on a case where two adjacent subtrees take part in the redistribution, but the discussion can easily be generalized to cases where any number of subtrees are involved. We shall redistribute the objects kept in the k-th and the (k+1)-st subtrees. B can be either the k-th or the (k+1)-st subtree. Let Num(k) and Num(k+1) be the number of objects stored in the k-th and the (k+1)-st subtrees respectively. We first calculate the average number of objects stored in the two subtrees. If it is found that the k-th subtree holds more objects than the average, those (in the k-th subtree) farthest from A's vantage point will be moved to the (k+1)-st subtree, so that both subtrees will eventually hold the same number of objects. The boundary distances and pointers involved in the subtrees will be updated accordingly. On the other hand, if we find that the (k+1)-st subtree holds more objects, its objects that are the closest to A's vantage point will be moved to the k-th subtree. Figure 17 gives the pseudocode description of the redistribution of objects between two adjacent subtrees. An example is also shown in Figure 16.

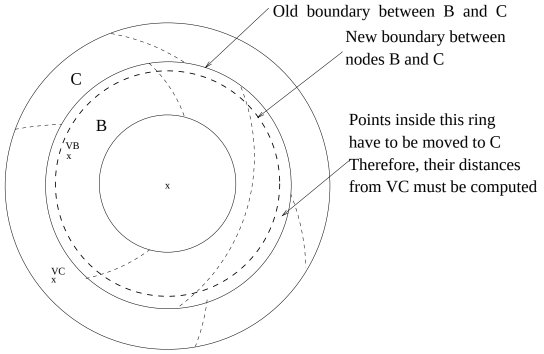


Figure 16: Redistribution among non-leaf nodes

(b) Else since A is not full, it has room for one more child, we split the non-leaf node B (Figure 18 and 19).

Assume B is the k-th child of A. Retrieve all objects stored in the subtree rooted at B, and let S be the set of objects retrieved plus the new object. Order the objects in S with respect to their distances from A's vantage point v. Divide S into 2 groups of equal cardinality, and let $\mathsf{SS_1}$ and $\mathsf{SS_2}$ be the two subsets in order. Let F be the number of subtrees rooted at A. To make room for the new subtree that is split from B, we shift the boundary distances and pointers of A in the way as follows. Note that make_vp_tree is the procedure for vp-tree construction, which will be called to construct two subtrees on the sets $\mathsf{SS_1}$ and $\mathsf{SS_2}$ respectively.

for $\mathsf{i} = \mathsf{k}$ *to* $\mathsf{F}-1$
    $A\uparrow.\mathsf{mu_{i+1}} := A\uparrow.\mathsf{mu_i}$;
$A\uparrow.\mathsf{mu_k} := (\max\{d(v,s_j) \mid \forall s_j \in SS_1\} + \min\{d(v,s_j) \mid \forall s_j \in SS_2\}) \div 2$;
for $\mathsf{i} = \mathsf{k}+1$ *to* F
    $A\uparrow.\mathsf{child_{i+1}} := A\uparrow.\mathsf{child_i}$;
$A\uparrow.\mathsf{child_k} := \mathsf{make\_vp\_tree}(SS_1)$;
$A\uparrow.\mathsf{child_{k+1}} := \mathsf{make\_vp\_tree}(SS_2)$.

4. Else, either L is the root node or we cannot find any ancestor node of L that is not full. Then we split the root node into two new nodes s_1 and s_2, and insert the new data point according to the strategy as discussed above for splitting either a leaf node or a non-leaf node. A new root node is created with s_1 and s_2 as the child nodes.

The insertion algorithm described above is based on a redistribute-first strategy, that is, we prefer redistribution to node splitting whenever both choices are allowed. We can certainly adopt a split-first strategy in which case node splitting has a higher order of preference. We shall compare the two strategies in our performance study.

begin
  average := $\lfloor(\mathsf{Num(k)} + \mathsf{Num(k+1)}) \div 2\rfloor$;
  if Num(k) > Num(k+1) *then*
    Let S be the set of objects stored in the k-th subtree plus the new object;
    Order the objects in S with respect to their distances from A's vantage point v;
    Let w be the number of data objects that will be moved from k-th subtree to (k+1)-th subtree;
    w := Num(k) − average;

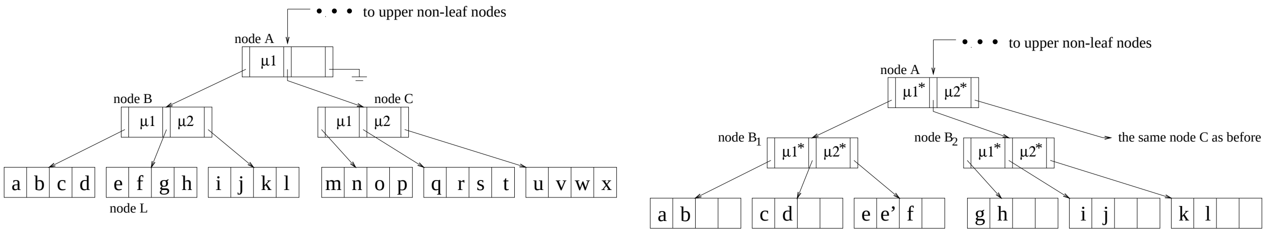
    Divide S into 2 subsets, SS_1 and SS_2 in order, where
      $SS_1 = \{S_1, S_2, ..., S_{Num(k)-w}\}$ and
      $SS_2 = \{S_{Num(k)-w+1}, S_{Num(k)-w+2}, ..., S_{Num(k)}\}$;

    for all $s_i \in SS_2$, delete s_i from the k-th subtree;
    $A \uparrow.mu_k := (\max\{d(v, s_j) \mid \forall s_j \in SS_1\} + \min\{d(v, s_j) \mid \forall s_j \in SS_2\}) \div 2$;
    for all $s_i \in SS_2$, reinsert s_i to the (k+1)-st subtree;
  else
    Let S be the set of objects stored in the (k+1)-st subtree plus the new object;
    Order the objects in S with respect to their distances from A's vantage point v;
    Let w be the number of data objects that will be moved from (k+1)-st subtree to k-th subtree;
    w := Num(k+1) − average;

    Divide S into 2 subsets, SS_1 and SS_2 in order, where
      $SS_1 = \{S_1, S_2, ..., S_w\}$ and $SS_2 = \{S_{w+1}, S_{w+2}, ..., S_{Num(k+1)}\}$;

    for all $s_i \in SS_1$, delete s_i from the (k+1)-st subtree;
    $A \uparrow.mu_k := (\max\{d(v, s_j) \mid \forall s_j \in SS_1\} + \min\{d(v, s_j) \mid \forall s_j \in SS_2\}) \div 2$;
    for all $s_i \in SS_1$, reinsert s_i to the k-th subtree;
  endif
end

Figure 17: Algorithm for redistributing objects between two adjacent subtrees.



(a) e' needs to be inserted into L, the B subtree is full and so are the siblings, but ancestor A still has room for one more child. We do node splitting at B.

(b) B has been split into B1 and B2, and e' gets inserted. Note that C remains unchanged.

Figure 18: Splitting of non-leaf node.

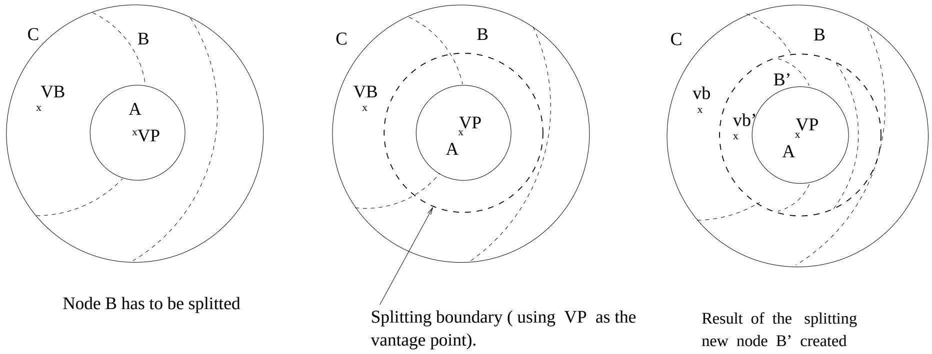


Figure 19: Splitting a non-leaf node

5.2 Delete

Traverse the tree in the same way as described in the insertion case until a leaf node L is found. Remove the object from the leaf node and see if the node underflows. If not, the task is done.

Let level(E) denote the level of node E. If E is a leaf node, level(E)=0. Let MIN_{leaf} be the minimum number of objects that should be stored in a leaf node, MIN_{fan} be the minimum number of subtrees that a non-leaf node should have. Then $\text{MIN}_{\text{data}}(E)$ denotes the minimum number of objects that should be stored in the subtree rooted at node E, and is defined as:

$$\text{MIN}_{\text{data}}(E) = \text{MIN}_{\text{leaf}} \times (\text{MIN}_{\text{fan}})^{\text{level}(E)}$$

Here we define that a leaf node underflows if the number of objects it stores is less than MIN_{leaf}, and that a subtree at node E underflows if the number of objects stored in that subtree is less than $\text{MIN}_{\text{data}}(E)$.

If the leaf node L underflows we choose the following scheme:

1. If L has a parent node P and P does not underflow, let F be the number of leaf nodes under P and we do either of the following.
   (a) If the total spare room of L's siblings can hold all of the objects in L, redistribute the objects under P among $\mathsf{F}-1$ nodes, i.e., L is to be merged with its siblings.
   (b) Else, when the total spare room is not enough to hold all of the objects in L, redistribute the objects under P among F nodes.
2. Else, locate a nearest ancestor A of L that does not underflow. Let B be the immediate child node of A, and B is also the ancestor of L. Assume B is the k-th subtree under A.
   (a) If either of the following three conditions is satisfied, we perform a merge. Note that the merge involves only adjacent subtrees. Figure 21 describes such a merge. We show a diagram of merging two nodes in Figure 20, in which we show that the vantage point VM can be new one or an old one, in our implementation, we choose to use an old one.

   Case 1: if the (k+1)-st subtree has enough room to hold all the objects in B, we move the objects in B to the (k+1)-th subtree and delete B.

   Case 2: if the (k-1)-th subtree has enough room to hold all the objects in B, we move the objects in B to the (k-1)-th subtree and delete B.

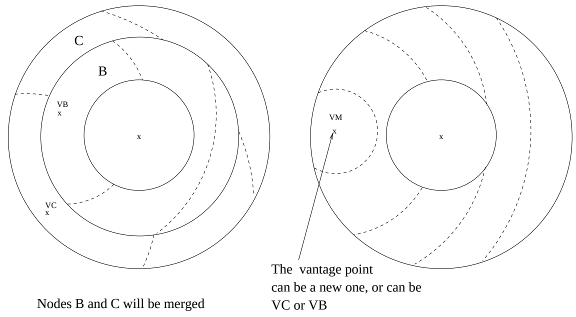


Figure 20: Merging 2 nodes

Case 3: if the total spare room of the (k+1)-st and the (k-1)-th subtrees can hold all the objects in B, we first calculate the number of objects that should be moved to the (k-1)-th subtree (denoted by the variable mid in Figure 21). We then move mid objects from B to the (k-1)-th subtree and the rest to the (k+1)-st subtree, and delete B.

(b) Else, when none of the above three conditions is satisfied, redistribute all the objects under A among its child subtrees. We apply the same method as in step 3(a) for insertion.

3. Else, either L is the root node or all ancestors of L underflow. In this case, if the root node has at least two child nodes, we merge two child nodes of the root with the deletion of the given data point. The merging strategy in the above can be used. The original root node is deleted and the merged node becomes the new root node. If the root node is a leaf node, it means that L is the root node and the deletion is carried out in the root node, we simply delete the data object from L.

The check on adjacency that the three cases outlined in step 2(a) have emphasized is to make sure that the re-insertion involved in the merge of subtrees will not cause redistributions. Similar to the insert algorithm, there can be redistribute-first and merge-first strategies of doing deletes. The above procedure is the merge-first strategy where redistributions will take place only when adjacent nodes do not have enough room to allow for a merge. On the other hand, in the redistribute-first strategy merges will occur only when all sibling nodes of the underflowing one are at the minimum size (that is, $\mathrm{MIN}_{\mathrm{leaf}}$ for leaf nodes or $\mathrm{MIN}_{\mathrm{data}}$ for non-leaf nodes).

5.3 Performance Evaluation

We conducted a number of experiments to demonstrate the performance of our insertion and deletion algorithms for the vp-tree. The algorithms were implemented in C under UNIX on an UltraSPARC.

We used three samples of data (clustered 30D, uniform 20D and real 16D), each containing 100,000 objects. A separate vp-tree was constructed to organize the objects of each of the three samples[5]. The branching factor of the tree is determined by the physical clustering strategy described in Section 4.1.1.

Then we inserted 10,000 new objects into each tree with the redistribute-first strategy as well as the split-first strategy. For every 1000 insertions, we measured the average page accesses required for all the insertions so far. Figure 22 plots the results for the clustered sample. We also counted the times that node splitting and redistribution had occurred at each interval of 1000 insertions. Tables 4 and 5 show the count and the associated cost in page accesses for the redistribute-first and split-first strategies respectively.

[5]Details of the data samples have been given in Section 4.1.1.

begin
  Retrieve all objects stored in the subtree B, and let S be the set of objects retrieved;
  Let F be the number of subtrees rooted at A;

  if Case 1 *then*
    for $\mathsf{i} = \mathsf{k}$ *to* $\mathsf{F}-2$, $A\uparrow.\mathsf{mu_i} := A\uparrow.\mathsf{mu_{i+1}}$;
    for all $\mathsf{s_i} \in \mathsf{S}$, insert $\mathsf{s_i}$ to the (k+1)-st subtree;

  elseif Case 2 *then*
    for $\mathsf{i} = \mathsf{k}-1$ *to* $\mathsf{F}-2$, $A\uparrow.\mathsf{mu_i} := A\uparrow.\mathsf{mu_{i+1}}$;
    for all $\mathsf{s_i} \in \mathsf{S}$, insert $\mathsf{s_i}$ to the (k-1)-th subtree;

  elseif Case 3 *then*
    Let Num(i) denote the number of objects stored in the i-th subtree;
    mid := $\{$Num(k-1) + Num(k) + Num(k+1)$\}$ $\div$ 2 $-$ Num(k-1);
    Order the objects in S with respect to their distances from A's vantage point v;

    Divide S into 2 subsets, $\mathsf{SS_1}$ and $\mathsf{SS_2}$ in order, where
      $\mathsf{SS_1} = \{\mathsf{S_1}, \mathsf{S_2}, \ldots, \mathsf{S_{mid}}\}$ and $\mathsf{SS_2} = \{\mathsf{S_{mid+1}}, \mathsf{S_{mid+2}}, \ldots, \mathsf{S_{Num(k)}}\}$;

    $A\uparrow.\mathsf{mu_{k-1}} := (\max\{d(v, s_j) \mid \forall s_j \in \mathsf{SS_1}\} + \min\{d(v, s_j) \mid \forall s_j \in \mathsf{SS_2}\}) \div 2$;
    for $\mathsf{i} = \mathsf{k}$ *to* $\mathsf{F}-2$, $A\uparrow.\mathsf{mu_i} := A\uparrow.\mathsf{mu_{i+1}}$;
    for all $\mathsf{s_i} \in \mathsf{SS_1}$, insert $\mathsf{s_i}$ to the (k-1)-th subtree;
    for all $\mathsf{s_i} \in \mathsf{SS_2}$, insert $\mathsf{s_i}$ to the (k+1)-st subtree;
  endif

  for $\mathsf{i} = \mathsf{k}$ *to* $\mathsf{F}-1$, $A\uparrow.\mathsf{child_i} := A\uparrow.\mathsf{child_{i+1}}$;
end

Figure 21: Algorithm for merging adjacent subtrees.

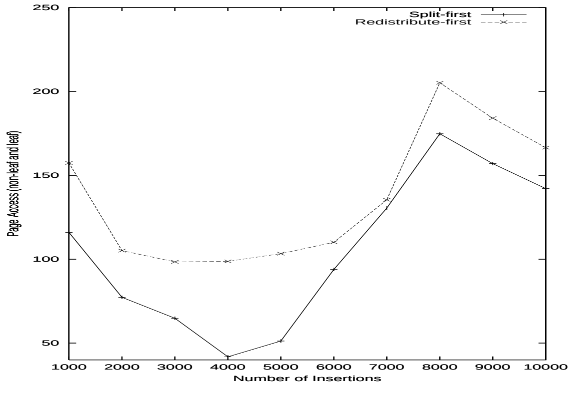


Figure 22: Insertion: synthetic clustered dataset of 100,000 objects.

Intervals of insertions	Node splitting		Redistribution	
	occurrence	avg. cost (pages)	occurrence	avg. cost (pages)
1 - 1000	13	67.54	6175	70.01
1001 - 2000	1	34.00	543	85.48
2001 - 3000	0	0	1100	90.34
3001 - 4000	0	0	1257	90.13
4001 - 5000	1	34.00	1357	89.53
5001 - 6000	0	38.00	1325	93.66
6001 - 7000	0	0	2753	84.33
7001 - 8000	0	0	5960	98.60
8001 - 9000	1	7012.00	0	0
9001 - 10000	0	0	2	25.00

Table 4: Access cost of splits and redistributions at non-leaf nodes with the redistribute-first strategy – synthetic clustered sample.

Intervals of insertions	Node splitting		Redistribution	
	occurrence	avg. cost (pages)	occurrence	avg. cost (pages)
1 - 1000	15	63.07	8820	67.36
1001 - 2000	0	0	302	71.07
2001 - 3000	0	0	315	71.37
3001 - 4000	0	0	415	73.05
4001 - 5000	3	38.00	900	118.84
5001 - 6000	5	38.00	3707	130.11
6001 - 7000	0	0	4804	127.95
7001 - 8000	0	0	4914	126.45
8001 - 9000	1	7012.00	0	0
9001 - 10000	0	0	0	0

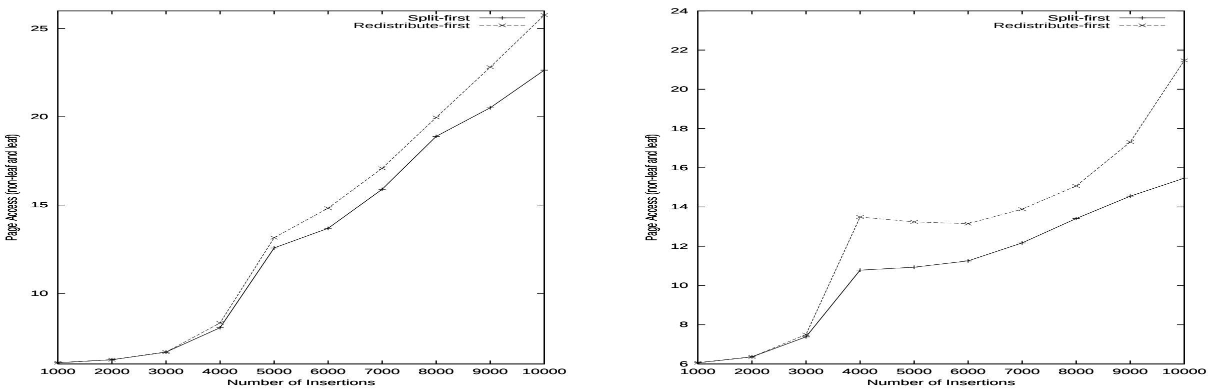
Table 5: Access cost of splits and redistributions at non-leaf nodes with the split-first strategy - synthetic clustered sample.

Note that we only focused on those which occurred at non-leaf nodes because the cost for splitting and redistribution among leaves is comparatively low.

Both of the tables show that the splitting and redistribution of non-leaf nodes occurs in a pulsate manner. During some intervals of insertions, the cost is entirely due to splits and redistributions among leaves. As such operations are not costly, the curves in Figure 22 for both strategies show a lower page access rates. Whereas at some other intervals, the splitting and redistribution of non-leaf nodes were more frequent. This is due to the phenomenon that nodes get fuller under more insertion and up to a point, most insertions triggers non-leaf node splitting or redistributions. At around 4000 insertions, we recognize that prior insertions have moved the nodes close to saturation. Hence, more and more redistributions were required, and each of these redistributions is costly because a large number of subtrees are involved. When no more redistribution could be done, node splitting became necessary.

Having done the split followed by considerable redistributions, the utilization of nodes has been averaged out, leading to a slight drop of the total page accesses after 6000 insertions. Soon after that, nodes became fuller and fuller due to the insertion of another 2000 objects. The trend there resembles the one at the beginning.

With the split-first strategy, choosing splitting rather than redistribution creates much room well before nodes become saturated. This strategy is able to avoid frequent subtree-based redistributions. Therefore,



(a) Synthetic uniform data

(b) real dataset

Figure 23: Insertion: Comparison with datasets of 100000 objects.

the overall access cost made by the split-first strategy is lower than the redistribute-first.

The results for the uniform and real data are shown in Figures 23(a) and 23(b) respectively. We can see from the figures that the two strategies exhibit a similar trend in that the split-first strategy is better.

Note that there is a big difference in the performance between the clustered data and the uniform and real data. This is in fact due to differences in the fullness of the nodes in the trees. For the clustered data, there are 30 dimensions, so the leaf node can hold up to 32 data objects, and the maximum branching factor is 15. With a tree of 4 levels, the maximum number of data objects is $32 \times 15^3 = 108,000$. We can see that this is very close to the number of data objects in the database, which is 100,000. Therefore, most of the nodes are fully packed in the beginning.

For the uniform data, there are 48 dimensions, the leaf node can hold up to 48 data objects, and the maximum branching factor is 17. A tree of 4 levels is needed to hold 100,000 data objects. With a tree of 4 levels, the maximum number of data objects is $48 \times 17^3 = 235,824$. Hence most of the leaf nodes in the tree are not full.

For the real data, there are 16 dimensions, the leaf node can hold up to 60 data objects, and the maximum branching factor is 18. Again a tree of 4 levels is needed to hold 100,000 data objects. With a tree of 4 levels, the maximum number of data objects is $60 \times 18^3 = 349,920$. Again most of the leaf nodes in the tree are not full.

If the leaf nodes are mostly full, we can expect a lot of redistribution and splitting operations, and this is seen in the case of the clustered dataset. If the leaf nodes are not full, as in the cases of the uniform dataset and the real dataset, insertion will not trigger much redistribution and splitting. Therefore the page access will also be very minimal. In fact, we have carried out another set of experiment for the clustered data, but limited the maximal page utilization to about 70%. The resulting number of page accesses for the split-first/merge-first strategies is around 8.

For measuring delete performance, we removed 10,000 objects from each of the vp-trees built for the three data samples using the redistribute-first strategy and the merge-first strategy. We set MIN_{fan} to 2 and MIN_{leaf} to be 50% of the maximum number of objects contained in a leaf.

For every 1000 deletions, we measured the average page accesses required for all the objects deleted so far. Figures 24-25 give the results for the clustered, uniform and real samples. All the curves show a similar

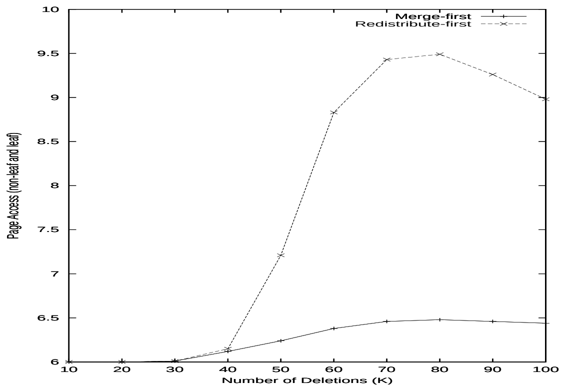
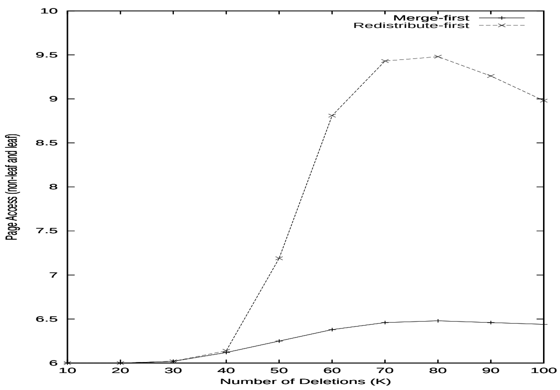
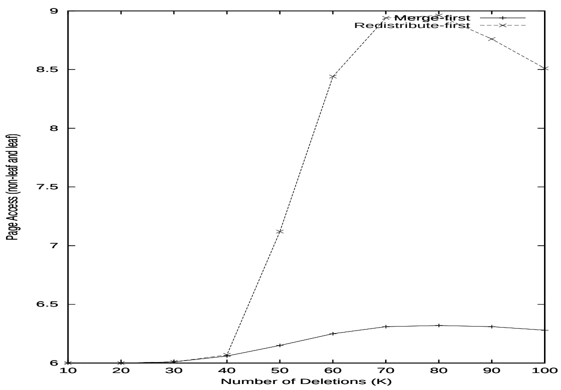


Figure 24: Deletion: synthetic clustered dataset of 100000 objects.



(a) synthetic uniform data



(b) real dataset

Figure 25: Deletion: Comparison on datasets of 100000 objects.

trend. Given that the height of the trees for clustered and uniform samples is only 6, we observe that the first 2000 deletions for both samples required only the corresponding minimum cost of deletes with both merge-first and redistribute-first strategies. This is also true for the first 3000 deletions for the real data sample. In other words, deleting such amounts of objects has not caused any underflows.

However, as more and more objects were removed, redistributions or merges occurred more often. Consequently, the average number of page accesses increased steadily with the number of deletions, as shown in all three figures. The reason that the merge-first strategy needs fewer page accesses than the redistribute-first strategy is because merging of nodes reduces the total number of nodes, and in turn increases the average utilization of nodes. Thus, underflows did not occur as frequently as in the case of redistribute-first.

Next we changed the minimum fan-out factor of the vp-tree to $\lceil m/2 \rceil$, where m is the maximum fan-out. We repeated similar experiments for deletion. This time we deleted the data until the tree became empty. The results are shown in Figures 26 and 27. Although the greater minimum fan-out would introduce more restructuring of the tree, it would also help to increase page utilization and shorten the tree. We see that the average number of page access has increased compared to the previous set of experiments, this is because more redistributions and merges occurred. However, the deletion process is still quite efficient.

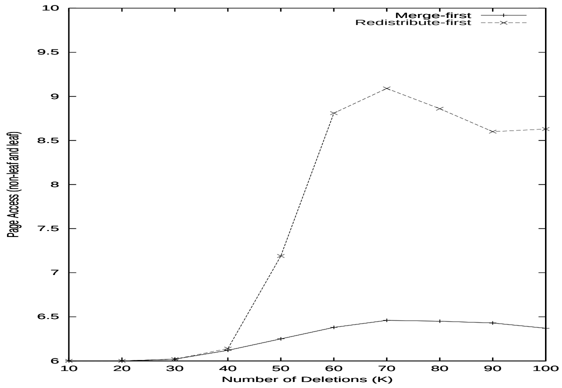
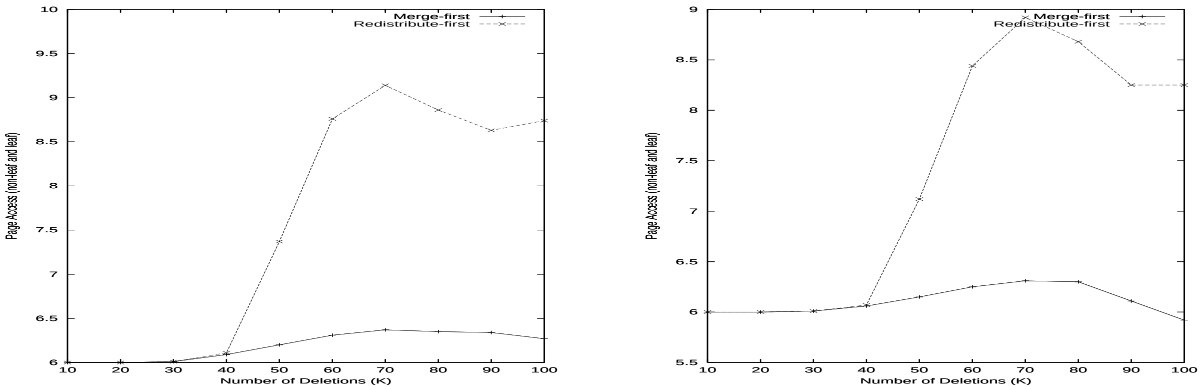


Figure 26: Deletion: synthetic clustered dataset of 100000 objects with minimum fan-out = $\lceil m/2 \rceil$.



(a) synthetic uniform data

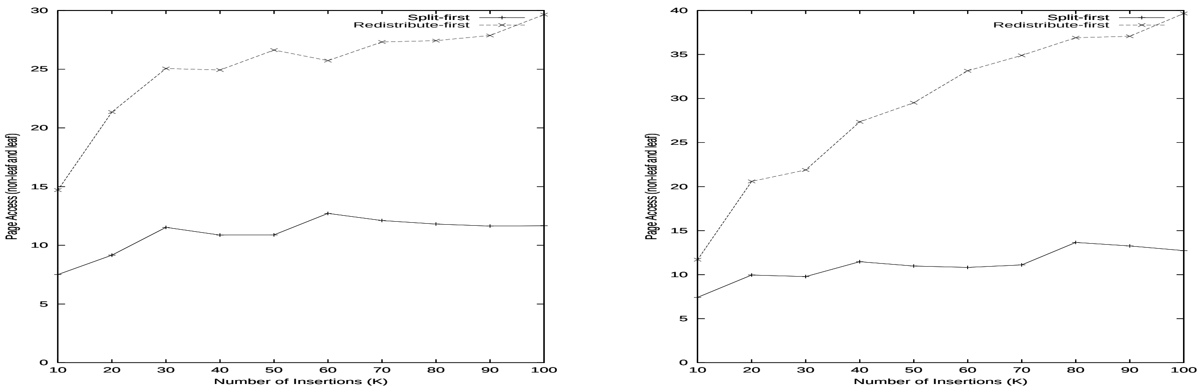
(b) real dataset

Figure 27: Deletion: Comparison on datasets of 100000 objects with minimum fan-out=$\lceil m/2 \rceil$.

5.3.1 Other performance evaluation

In the previous subsections, updates have been performed on top of an existing database. Next we investigate the case where we begin with an empty database. We incrementally updated the database and the results are shown in Figures 28 and 29. The average page access is lower for the split-first strategy. The performance is slightly worse than that for inserting on an existing database, however, it is still acceptable as the number of page accesses is below 15 for the split-first case for all the cases considered.

We carried out another set of experiments interleaving insertion and update operations and the results in performance are similar to pure insertion/deletion.



(a) synthetic uniform data (b) real dataset

Figure 28: Starting with an empty tree: Comparison on inserting datasets of 100000 objects.

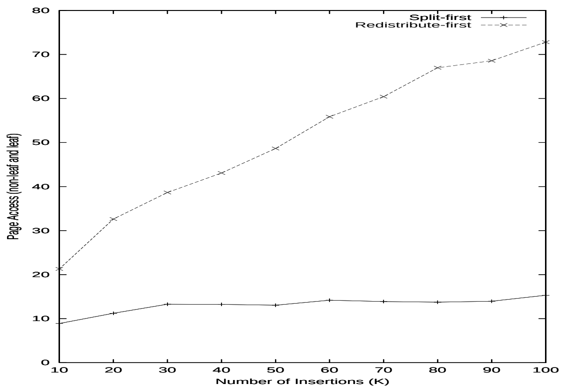
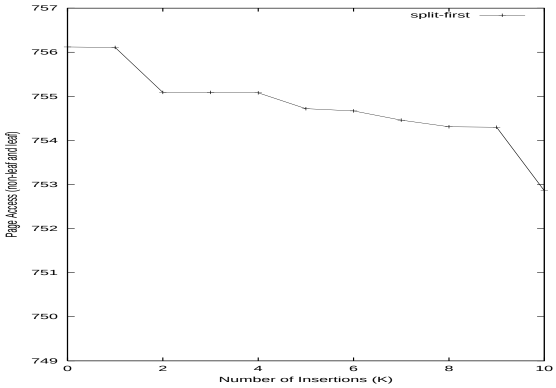
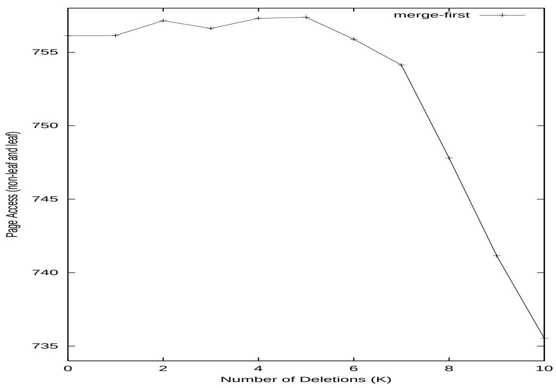


Figure 29: Starting with an empty tree: Comparison on inserting a synthetic clustered dataset of 100000 objects.

We have also performed a set of experiments to evaluate the effect of update operations on n-nearest neighbor search. Instead of using the maximal page utilization of 100%, we use a maximal page utilization of around 70% for the vp-tree, because with node splitting, the average utilization should be less than maximal. We chose this amount of utilization because it has been found typical in the B^+-tree indexing. Starting with a dataset of size 100,000, we performed insert or delete operations and also 8-nearest neighbor query. We measured the average 8-nearest neighbor query performance in terms of page accesses after each 1000 insertion or deletion. The results for the clustered data are shown in the Figure 30. Note that the query performances before the updating is slightly worse than those reported in Section 4.1 since here we began with a node utilization of 70%. The search performance has not been much affected by the updating. In fact, after a certain amount of insertion or deletion, the nodes probably became even better organized than the initial 70% utilization through the re-organization of the tree during insertion or deletion. This is reflected in the drop in the number of page access of the n-nearest neighbor search. The results show that the proposed insertion and deletion algorithms are effective in maintaining the desired goodness of the tree structures.



(a) insert



(b) delete

Figure 30: Updating and Querying: Comparison on datasets of 100000 objects

6 Vantage Point Selection

In a high-dimensional space, the distance calculations between data objects are expected to be computationally expensive. As such, a major concern on distance-based indexing is to minimize the number of distance computations in order to aim at efficient query processing. The mvp-tree [6] is one recent example. The mvp-tree uses two vantage points in every node. In binary mvp-trees, the first vantage point divides the space into two parts, and the second vantage point divides each of these partitions into two, making the fanout of a node in a binary mvp-tree four. As seen from Figure 31, each node of the mvp-tree can be viewed as two levels of a vp-tree, but involving fewer vantage points[6]. Because of using more than one vantage point in a node, the mvp-tree has fewer vantage points compared to a vp-tree. For query processing, most of the distance computations made are between the query point and the vantage points. The mvp-tree structure can therefore reduce a certain amount of distance computations. The mvp-tree approach will be compared with the two alternatives we shall present, with a number of experiments.

6.1 A Single Vantage Point per Level

In vp-trees, every node of the tree is associated with a distinct vantage point. When the search operation traverses multiple branches, we have to make a different distance computation at the root of each branch. Conversely, if we use a single vantage point to partition the regions associated with the nodes of the same

[6]In Figure 31, v1,v2,v3 denote the different vantage points used in the nodes.

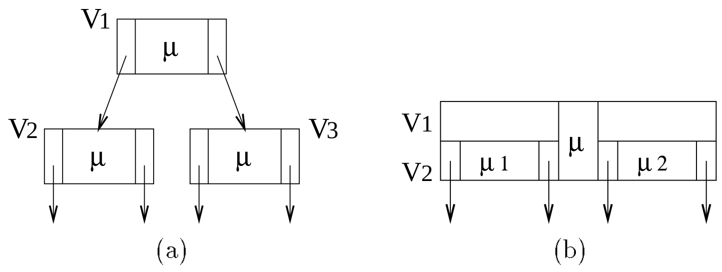


Figure 31: Node structures for (a) a binary vp-tree and (b) a binary mvp-tree.

level, only one distance computation will be involved at each non-leaf level. This is the idea behind our first method for minimizing distance computations.

At the root level, we choose the first vantage point with the method depicted in Figure 2 (also the method used in the original vp-tree [39, 10]). Then we choose the second vantage point for the next level to be one of the farthest points from the first vantage point; the third vantage point to be the farthest from both of the previous two vantage points; and so forth. The reason why we require the vantage points to be far apart is to ensure a relatively effective partitioning of the dataset.

Since there is only a single vantage point for each level, in a search operation, the number of distance computations at non-leaf nodes is equivalent to the number of non-leaf levels of the tree, which can be assumed to be a small number. Because of the small quantity, we can keep the vantage points outside the tree and keep only pointers to them in the tree. This allows a higher fanout at the non-leaf nodes and a smaller tree size, and consequently can enhance the performance on querying.

6.2 Reuse of Vantage Points

The main drawback of using a single vantage point for each level lies in the deviation from the original partitioning strategy of the vp-tree. In the original method, every chosen vantage point (by the algorithm in Figure 2) should suit its associated region to a certain extent. Although we attempt to maintain a good partitioning as in the original method by choosing vantage points that are distant from each other, the one chosen vantage point may not be appropriate for each of the nodes at the corresponding level. Our second method tries to achieve a balance between a favorable partitioning of the dataset and a reduction of distance computations.

Unlike the previous approach, nodes at the same level may have different vantage points. However, not every such vantage point will be unique. Some of them may in fact be the same, because the vantage points are reused. Before building the vp-tree we fix a number p to be the maximum number of vantage points that we shall use in total. The selection of these p vantage points is the same as in the previous approach: the first vantage point is selected based on the algorithm in Algorithm 2, and all of the p points are chosen to be the farthest from each other. Then, we construct the vp-tree using such pre-selected vantage points. In other words, the set of pre-selected points act as the ‘candidate vantage points’ described in Algorithm 2. By increasing the number p, our method provides more choices of vantage points for the partitioning at each node. Clearly, the number of distance computations at non-leaf nodes for query processing is bounded by p. If the number p is of a manageable amount such that keeping them in the main memory is not costly, we can keep the p vantage points outside the vp-tree as in the previous approach. This can reduce the storage size of the tree and increase the fanout of the non-leaf nodes, in particular if the vantage points are

Dataset size	Number of distance computations			
	reuse	single	mvpt	original
10000	492.31	502.18	498.33	509.66
20000	1096.85	1106.77	1105.02	1125.46
30000	1812.58	1816.85	1829.67	1876.81
40000	2237.00	2239.46	2236.00	2320.76
50000	2743.43	2754.07	2744.59	2832.18

Table 6: 8 nearest neighbor search for clustered data of dimension 30.

high-dimensional feature vectors.

It is a good idea to keep p small so that we can store all the vantage points outside the tree and use less time to select the p distant points out of the dataset and to determine the best vantage point for each non-leaf node. But a minimal p may offset good partitioning of the dataset. We believe that a good value of p can be found by trial and error for a given dataset.

6.3 Performance Evaluation

To compare our methods with the mvp-tree approach, we implemented the disk-based model of the mvp-tree and extended it with the n-nearest neighbor search algorithm. Our original implementation of the vp-tree was modified according to the two methods we proposed. The mvp-tree and the vp-tree were both implemented in C on an UltraSPARC.

For each data point x in the leaves of an mvp-tree, the tree keeps the pre-computed (at construction time) distances between the data point x and the first b vantage points along the path from the root to the leaf node that keeps x. These distances are used for effective filtering of non-qualifying objects during search operations. The experiments in [6] have proved the competence of such a technique. All of the vp-trees (including the original version) and mvp-trees we built for this performance study employed this technique. We set b to 3, i.e., three extra distances were stored for each data point in the leaves. For the method that reuses a fixed number p of vantage points, we set the value of p to be a reasonably small number, 20. For only the 'single vantage point per level' approach, we kept the vantage points outside the vp-trees.

Two performance metrics were used: the number of distance computations and page accesses. We counted the number of distance computations and page accesses required for 8-nearest neighbor queries by each method. All results were averaged over 100 such queries. We used five sets of synthetic clustered data, each containing a different amount of data points in dimensions of 30. The amounts vary from 10000 to 50000. The details of these datasets have been given in Section 4.5.1.

We present the results in Tables 6 and 7. In these tables, the column labeled 'reuse' refers to the method that reuses a fixed number of vantage points, 'single' refers to the method that associates only a single vantage point with each non-leaf level, 'mvpt' refers to the method adopted by the mvp-tree, and 'original' refers to the original vp-tree structure. Note that the results made by the original vp-tree are provided for reference. Table 6 reports the number of distance computations for various dataset sizes. As seen from the table, reusing vantage points achieves the best results, and the mvp-tree approach is better than the 'single' method. This indicates that choosing only a single vantage point for all the nodes at the same level has certain negative effects on the partitioning at these nodes, which leads to more multiple-path searching, and in turn more leaf accesses. As a result, more distance computations between the query point and the data points are involved.

Dataset size	Page accesses			
	reuse	single	mvpt	original
10000	29.23	22.76	31.91	31.06
20000	56.04	55.70	56.79	60.18
30000	68.57	65.45	82.92	90.07
40000	101.86	100.83	100.66	120.15
50000	117.12	116.90	117.99	141.79

Table 7: Page accesses per search for 8 nearest neighbor search for clustered data of dimension 30.

Besides the better performance it offers, the 'reuse' method has two other advantages over the mvp-tree method. Firstly, as its method for selecting vantage points is straightforward and the selection process is completed well before tree construction, it makes the construction easier and less time is required. Secondly, we can reduce the size of the tree by storing all of the vantage points in use outside the tree, when the total number of them is small enough (such as 20 in our experiments).

We also measured the page accesses to see how the three methods affect the access cost of the vp-tree. Table 7 displays the results. All three methods in general make fewer page accesses than the original vp-tree structure. The 'single' method needs the least number of page accesses. This is merely because the trees constructed based on this 'single vantage point per level' approach are the smallest compared to the others. With a smaller tree size, the access cost for search operations is inevitably lower. In summary, our methods are comparable with the mvp-tree approach in terms of the number of distance computations and additionally provide better performance in page accesses. From the performance in searching, we believe that the reuse of vantage points would not affect the goodness of the tree structure. Moreover, they may facilitate the updating procedures as described in the previous section. Reusing vantage points can lead to less work in updating since pre-computed distances of data objects to reused vantage points can be used.

7 Conclusions and Future Work

We tackle the problem of content based retrieval for multimedia data objects given only pair-wise distances between data objects. One approach to solve this problem is to first infer a feature vector for every data object from the distances provided, preserving as much as possible the distances between objects. We ran some experiments to show that this approach may incur considerable inaccuracies in the query results. Then we examine another approach which is to make use of a distance-based indexing structure. Such an approach provides a number of advantages: First, pre-processing steps involved in inferring feature vectors can be eliminated. Second, the difficulty in preserving distances is avoided. Third, updating can be handled much more efficiently. Lastly, the method can be applied to data represented as multi-dimensional vectors as well. A promising structure for this approach as shown in [6] is the vp-tree.

We examine two of the important problems for the vp-tree and its variants: n-nearest neighbor search and updating. Also, we propose some methods of vantage point selection to reduce the number of distance computation. We proposed an n-nearest neighbor search algorithm for the vp-tree. We show by experiments that the algorithm scales up well with size of the data sets and the value of n, for a set of synthetic clustered data, uniform data and a set of real data. It also scales up well with the dimensionality. We compare the result with the R^*-tree and the M-tree and show that the vp-tree outperforms both trees significantly for n-nearest neighbor search for synthetic clustered data, uniform data and real data by experiments.

We propose some solutions to the update problem for the vp-tree and its variants. The insertion and deletion procedures involve nodes redistribution, splitting and merging, and always preserve the balanced structure of the vp-tree. There are two alternatives of split-first or redistribute-first for insertion, as well as two alternatives of merge-first or redistribute-first for deletion. We study the performance of these methods and show that the overhead is quite acceptable, we also show that the split-first and merge-first strategy is superior to the redistribute-first strategy.

We propose alternatives to the vantage point selection, by making use of the idea of reusing vantage points. This was the basic idea in [6]. We design different methods to reuse vantage points. It turns out that the results in terms of reduction in distance computation are similar for the different methods.

For future work, we may investigate some other methods of vantage point selection. The major criticism of [7] about vp-trees is that *the region inside the median sphere and the region outside the median sphere are extremely asymmetric, and since volume grows rapidly as the radius of a sphere increases, the outside of the sphere tends to be very thin.* Let us consider the convex hull enclosing all data points to be our *search space*. If the vantage point creates a boundary that is a hypersphere surface mostly within the search space, then the above argument of [7] holds. However, if the vantage point is far from the search space, (theoretically it can be at an infinite distance from all points) then the boundary created by the median in the search space will be closer to a hyperplane that divides the search space into two regions. The regions would then be more *symmetric*. Therefore, we may want to look for vantage points that are far away from the center of the search space. We can locate such points efficiently by a method such as the heuristic introduced in [24] for choosing two distant objects. We note that the space near the spherical boundary created by the median of a vantage point is a critical factor for the performance of query evaluation, since a query that falls under such a region will likely lead to a search on both sides of the boundary. A method adopted in our experiments and in previous work make use of the standard deviation to try to ensure a small number of data points around the boundary. We therefore look for a vantage point that is far from the center of the search space of the current node and which also has a high standard deviation. However, it is possible that the standard deviation method would often automatically introduce vantage points that are at the edge of the search space (see Figure 1 in [39]), and it is left for further investigation to see if this is true.

Finally, we would like to investigate improvements on the vp-tree in the future. There have been enhancements on the R^*-tree such as introducing redundancy [16] and the introduction of some flavor of linear search as in the X-tree [4]. We can try to capture the ideas behind such enhancements and apply them to the vp-tree.

Acknowledgments

We are very thankful to the authors of [4] for providing us with the set of real data from their experimental setup. We would also like to thank the authors of [6] for sending us a pre-published version of their paper. We thank Kelvin, Kam Wing Chu for his enhanced version of the M-tree implementation, and Chun Hing Cai for the implementation of the enhanced n-nearest neighbor search of the R^*-tree. We are very grateful to the anonymous referees who gave very thoughtful and useful comments to enhance the paper.

References

[1] W. A. Ainsworth. *Speech Recognition by Machine.* London: Peter Peregrinus Ltd, 1988.

[2] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 322–331, May 1990.

[3] S. Berchtold, C. Bohm, B. Braunmuller, D. A. Keim, and H.P. Kriegel. Fast Parallel Similarity Search in Multimedia Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1997.

[4] S. Berchtold, D. A. Keim, and H.P. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on VLDB*, 1996.

[5] D. J. Berndt and J. Clifford. Finding Patterns in Time Series: A Dynamic Programming Approach. In *Advances in Knowledge Discovery and Data Mining, Edited by Fayyad U.M. and Piatestsky-Shapiro G. and Smyth P. and Uthurusamy R.* AAAI/MIT Press, 1996.

[6] T. Bozkaya and M. Ozoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1997.

[7] S. Brin. Near neighbor search in large metric space. In *Proceedings of the 21st International Conference on VLDB*, pages 574–584, 1995.

[8] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[9] K. Cheung and A. Fu. Enhanced nearest neighbour search on the r-tree. In *ACM SIGMOD Record*, volume 27:3, September 1998.

[10] T.C. Chiueh. Content-based image indexing. In *Proceedings of the 20th VLDB Conference*, pages 582–593, 1994.

[11] T. Ciaccia, Patella M., and Zezula P. M-tree: an efficient access method for similarity search in metic. In *In Proceedings of the 23rd Inernational Conference on Very Large Data Bases*, pages 426–435, August 1997.

[12] G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{II}-tree: a Modified bB-tree supporting concurrency, recovery and node consolidation. In *Proceedings of the 21st International Conference on VLDB*, pages 551–561, 1995.

[13] G. Evangelidis, D. Lomet, and B. Salzberg. The hBpi-tree: a multi-attribute index supporting concurrency, recovery and node consolidation. *the VLDB Journal*, 6(1):1–25, 1997.

[14] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, July 1994.

[15] J.H. Friedman, J.H. Bentley, and Finkel A. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, Sept. 1977.

[16] A. Fu and K.L. Cheung. Enhancements on the R-tree to support efficient similarity search in high-dimensional space. In *Manuscript*, 1997.

[17] W.I. Grosky and R. Mehrotra. Index-Based Object Recognition in Pictorial Data Management. *Computer Vision*, 52(3):416–436, 1990.

[18] V.N. Gudivada and V.V. Raghavan. Content-based image retrieval systems. *IEEE Computer*, 28(9):18–22, September 1995.

[19] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 47–57, June 1984.

[20] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, July 1995.

[21] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1997.

[22] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. Technical Report CS-TR-3613, University of Maryland, March 1996.

[23] J. B. Kruskal and M. Wish. *Multidimensional scaling.* SAGE publication, Beverly Hills, 1978.

[24] K.I. Lin and C. Faloutsos. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of ACM SIGMOD*, 1995.

[25] K.I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree - an index structure for high-dimensional data. *VLDB Journal*, 3:517–542, October 1994.

[26] D. B. Lomet and B. Salzberg. The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[27] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: querying images by content using color, texture and shape. In *Proceedings of SPIE: Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, February 1993.

[28] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 71–79, June 1995.

[29] H. Sakoe and S. Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. In *Readings in Speech Recognition, Edited by Waibel, A., and Lee K.* Morgan Kaufmann Publishers, 1990.

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

[31] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparisons.* Addison-Wesley, 1983.

[32] T. Seidl and H.P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 154–165, 1998.

[33] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multidimensional objects. In *Proceedings of the 13th International Conference on VLDB*, pages 507–518, 1987.

[34] Y. Theodoridis and T. Sellis. A Model for Prediction of R-tree Performance. In *Proceedings of the ACM Principles of Database Systems, PODS*, pages 161–170, 1996.

[35] S. Thomas and H. Kriegel. Efficient User-Adaptable Similarity Search in Large Multimedia Databases. In *Proceedings of the 23th International Conference on VLDB*, 1997.

[36] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:4:175–179, November 1991.

[37] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, University of California, San Diego, July 1996.

[38] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, pages 516–523, February 1996.

[39] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.