INDEXING METHODS FOR MULTIMEDIA DATA OBJECTS GIVEN PAIR-WISE DISTANCES

By

CHAN Mei Shuen Polly

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science and Engineering

The Chinese University of Hong Hong

1997

Abstract

By means of feature extraction, multimedia data can be mapped into points in k-dimensional space, and thus, any feature-based indexing method can be used to organize and efficiently search the k-d points. For some multimedia applications, however, it has been found that domain objects cannot be represented as feature vectors in a multidimensional space. Instead, pair-wise distances between the data objects are the only input. To support content-based retrieval, one approach transforms each object to a k-d point from some unknown high-dimensional space and tries to preserve the distances between the points. Then existing feature-based indexing methods such as R-trees and kd-trees, etc, can support fast searching on the resulting k-d points. Information loss is inevitable with such an approach since the distances between data objects can only be preserved to some extent. We show by experiments that the distance preserving approach introduces considerable inaccuracy for n-nearest neighbor search. In this thesis we investigate the use of distance-based indexing methods. In particular we apply the Vantage-Point tree (vp-tree) method. This approach allows for an index construction directly based on the distance information. Previous work on the vp-tree has not explored algorithms for n-nearest neighbor search. We propose three n-nearest neighbor search algorithms, which are shown by experiments to scale up well with the size of datasets and the dimensionality. In addition, we propose a solution to the update problem for the vp-tree, which has been left open in previous work. We also study various methods for minimizing the distance computations involved in the vp-tree for handling queries. Our methods are shown by experiments to perform better than some previous methods.

Acknowledgement

Foremost, I would like to express my eternal gratitude to my supervisors, Professor Ada Fu and Professor Yiu Sang Moon, for their guidance, support and patience throughout the work on this thesis. I would also like to extend my gratitude to the other members of my examining committee, Professor Chin Lu, Professor Man Hon Wong and Professor Siu Cheung Chau, for useful input which helped shaping the final version of this thesis.

I sincerely thank S. Berchtold, D. A. Keim and Hans-Peter Kriegel for giving me the data I used in my experiments. My thanks too to my fellow classmates, Kingly Cheung and Terence Wong, for enduring me in the past two years. Finally, I am grateful to my dearest Frank, my parents and my younger sister and brothers for their love and encouragement.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

With the advent of large-scale multimedia database systems, there is a need to efficiently answer users' queries. *Content-based retrieval* is typically required. 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 query 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, 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. The resulting k-dimensional points are called *feature vectors*. Examples of feature vectors are color components of an image [25], shot cuts of a video clip [11], shape descriptors [17], Fourier vectors [33], etc. Besides the capability to extract key features from data objects, we also need the ability to capture, what we humans perceive as, a similarity between two objects. Hence, processing content-based queries typically requires some measurement of similarity between the 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, although other metrics such as the city-block distance can also be used (definitions will be given

later). The entire problem of content-based retrieval 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 *feature-based indexing methods.*

However, it has been found that the above setting cannot be applied to certain applications. Consider the application in genetics, finding similar DNA or protein sequences from a genetics database would be a commonly-asked query. In information retrieval, we need to find sentences semantically similar to a user's query in a large database of documents. We would like to match digitized voice excerpts in voice recognition. In these applications, selecting a suitable set of features to accurately represent objects is not always an easy task since objects like strings and patterns cannot be easily represented as vectors. When the feature elements are complex and domain dependent, the process of feature extraction is complicated. Fortunately, it is relatively easier for a domain expert to assess the similarity or distance between two objects [34]. Given only the distance information, feature-based indexing methods cannot offer the required access mechanism. Nevertheless, we need some index structures to facilitate query and update operations. This is precisely the motivation behind this work.

For this problem, one approach uses the distance information to deduce k-dimensional points for the data objects so that we can subsequently make use of any readily available feature-based indexing methods such as the R-tree. The *FastMap* algorithm [22] and *Multidimensional Scaling* [21] fall into this category. The main challenge for this approach is to preserve the distances between objects as much as possible. Because of the discrepancy between the actual distances and the transformed distances over the k-d space, errors result in finding all the required nearest neighbors to a query object. Our experimental results show that such an approach can incur a considerable amount of inaccuracies in doing n-nearest neighbor search.

This thesis attempts to solve the problem with an alternative approach, *distance-based indexing.* In particular we apply the *Vantage-Point tree* (vp-tree) method [16, 36, 8]. The partitioning mechanism of distance-based indexing methods allows us to construct an index structure for domain objects directly based on the distance

information provided. This approach can obviously save the overhead of inferring points in a multidimensional space, and can also avoid the difficulty in preserving distances so that the correctness of search results can be guaranteed. Besides, this approach can be applied not only to the distance case but also to the vector case where data objects are well represented by feature vectors, once the distance function has been defined. In fact, some recent work has proposed distance-based indexing methods as the solution to the problems arising from indexing high-dimensional vectors [32].

There has been a wealth of previous work on distance-based indexing for multidimensional data. However, as far as we know, none of the previous work explored the problem of n-nearest neighbor search, for $n > 1$. We propose three n-nearest neighbor search algorithms for the vp-tree method. In our experiments, the search algorithms demonstrate promising performance. We also note that the update problem has been left open for the vp-tree and its variants [32]. In this work we propose mechanisms for these operations on the vp-tree. The distance calculations involved in distance-based index structures contribute largely to the computation cost. Some techniques in reducing the number of these calculations will be investigated.

1.1 Definitions

Content-based retrieval of multimedia data relies on similarity measures to assess the distance between data objects. Metric distance functions are one prevalent form of similarity measures although some applications may use similarity matrix. Below we describe various distance functions and state their common properties.

Similarity Measures Most multimedia data objects are represented by a k-dimensional feature vector, and as such are represented as points in a k-dimensional space. 'Distance' between a pair of such points represents the dissimilarity between those objects. The farther the points are from each other, the more dissimilar the objects are and vice versa. The distance function defined below can be mapped into the range [0,1] to represent dissimilarity, which when subtracted from unity yields the similarity measure [18].

Minkowski r-metric refers to a class of metric distance functions defined as follows:

$$d_r(x,y) = \left[\sum_{i=1}^{k} |x_i - y_i|^r\right]^{1/r}, \quad r \geq 1 \tag{1.1}$$

$$d_\infty(x,y) = \max_i |x_i - y_i| \tag{1.2}$$

where x and y are two points in a k-dimensional space with components, x_i, y_i, i =1,2,...,k. For $r = 2$, it is the Euclidean metric, for $r = 1$, it is the city-block metric, and for $r = \infty$, it is the dominance metric.

The Euclidean metric is the most common metric used in multimedia applications. However, there are other complicated distance functions specifically designed for certain applications. In medical databases where X-ray images and brain scans are stored, the distance functions must involve some warping of the two images to make sure the anatomical structures are properly aligned, before the differences can be assessed [2]. In DNA and string databases, the distance function is typically the editing distance which refers to the minimum number of insertions, deletions or substitutions that are needed to transform one string to the other [22]. These distance functions must exhibit the properties of a metric distance function.

By definition, a metric distance function, $d(x,y)$, has three properties [18, 9]:

1. Symmetry: $d(x,y) = d(y,x)$;
2. Positivity: $0 < d(x,y) < \infty, x \neq y$ and $d(x,x) = 0$;
3. Triangle inequality: $d(x,y) \leq d(x,z) + d(z,y)$.

Query types Having realized the similarity between data objects, we are ready to answer users' queries. There are various kinds of queries on multimedia objects. The most typical ones are listed below:

1. Exact match queries. Find if a given query object is in the database.
2. Nearest neighbor queries. Find the first n ($n \geq 1$) objects that are closest to the query object.
3. Range queries. Find objects that are within distance ϵ from the query object. When $\epsilon = 0$ the query corresponds to an exact match query.

1.2 Thesis Overview

The rest of the thesis is organized as follows. Previous work on multimedia indexing and in particular the vp-tree method are described in Chapter 2. In Chapter 3 we analyze the problem of the distance preserving transformation approach in handling nearest neighbor queries, and demonstrate the applicability of the approach we propose. Chapter 4 introduces the concept behind n-nearest neighbor search and details our three algorithms for n-nearest neighbor search in vp-trees. In addition to the performance studies on the proposed algorithms, we compare the vp-tree with the R^*-tree which is one well-known feature-based indexing method. Update mechanisms for the vp-tree are presented in Chapter 5, along with performance results. We provide in Chapter 6 some methods for minimizing distance computations. Finally, we conclude the thesis in Chapter 7 with an outline of future work.

Chapter 2

Background and Related Work

The state of the art in multimedia indexing is based on feature extraction [17, 20]. With proper feature extraction functions, domain objects are represented as feature vectors (points in a multidimensional space). Feature-based index structures are then used as a major filtering mechanism. In the more complex case where objects cannot be mapped to points in the multidimensional space, and we only have an expert-defined distance function that computes the distance (dissimilarity) between objects, we have two main approaches to tackle the indexing problem. One approach is to use distance preserving methods to deduce multidimensional points for the objects from the given distance function and then to apply feature-based index structures. Another approach is to use distance-based index structures. Note that distance-based index structures can be also used to index feature vectors once the distance function has been determined. This review will briefly describe various choices of feature-based index structures, distance preserving methods and distance-based index structures.

2.1 Feature-Based Index Structures

Feature-based index structures are conventional indexing techniques for multimedia datasets that can be described by means of feature vectors. Since feature vectors are multidimensional, feature-based index structures are also called multidimensional indexing methods. A large amount of work has been done on this subject. Many

structures, such as grid-files [15, 26] and linear quadtrees [12, 1, 30], do not scale well to high dimensions[1] whereas structures based on the kd-tree [4] (kdB-trees [28] and hB-trees [24]) and structures based on the R-tree [13] are methods that can extend to higher dimensions. Among these, the R-tree and its most successful variant, the R*-tree [3], have been the most popular structures for indexing high dimensional data [34]. Experiments in [10, 14] show that R*-trees work well for up to 20 dimensions.

The R-tree can be imagined as an extension of the B-tree for multidimensional objects [27]. Every object is represented by its *Minimum Bounding Rectangle* (MBR). Entries in non-leaf nodes are of the form (*R, ptr*) where *R* is the MBR that covers all rectangles in the child node pointed to by *ptr*. Leaf nodes contain entries of the form (*R, object-id*) where *R* is the MBR that encloses the data object pointed to by *object-id*. The tree grows in a bottom-up fashion. Extensions, variations and improvements to the original R-tree structure include the packed R-tree [29], the R^+-tree [31], the R*-tree [3], the Hilbert R-tree [19], etc.

The TV-tree [23], the SS-tree [35] and the X-tree [5] are recent methods proposed specifically for indexing high dimensional data. Both TV-trees and SS-trees performed better than the R*-tree. The X-tree was compared with the R*-tree as well as the TV-tree and was shown to be superior to either method. The idea of the TV-tree is to use only the features needed to distinguish between data objects at the top levels of the tree, and to store more and more features in the nodes that are closer to the leaves. This leads to smaller internal nodes and a higher fanout, resulting in a better query performance [5]. The SS-tree uses ellipsoid bounding regions, instead of rectangular shapes as in the R-tree structure, to enclose data objects. The R-tree family, TV-trees and SS-trees suffer from the overlap problem of bounding boxes [34, 5]. Berchtold et. al. [5] addressed this problem and introduced the X-tree that uses an overlap-minimizing split algorithm and extended variable size internal nodes (so-called supernodes) to avoid or eliminate overlap between search regions.

[1] This is the well-known "dimensionality curse" problem, which means that the performance of indexing methods degrades with the dimensionality, eventually reducing to that of sequential scanning.

2.2 Distance Preserving Methods

In the distance preserving approach, we try to deduce for each object a corresponding point in a multidimensional space so that the distances between objects are preserved as much as possible. One example is an old method from pattern recognition, namely, Multidimensional Scaling (MDS) [21]. Another is the FastMap algorithm proposed by Faloutsos and Lin [22]. As experiments in [22] showed that FastMap achieves dramatic time savings over MDS, without loss in quality of the results, we shall focus only on the FastMap method. The details of MDS 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 the distances between objects as much as possible such that the Euclidean distances between the points in the resulting k-dimensional space match the pair-wise distances given.

Faloutsos and Lin [22] claimed that FastMap can accelerate the search time for queries. This is mainly because a number of highly fine-tuned feature-based indexing methods like the R-tree [13, 3, 31] can be employed to provide fast searching for range queries and n-nearest neighbor queries. However, FastMap inevitably introduces pre-processing costs to both index construction and querying since all domain objects and the query objects must first be mapped to corresponding k-d points before an index structure is built or queries are processed.

The mapping of N objects into N k-d points requires k recursive calls to the FastMap algorithm. 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. The first coordinate of an object i (for $i = 1$ to N) is computed according to the given distance function $D()$. To determine the second coordinate, the object i has to be projected on another axis. Let object i' stand for the projection of object i. Then, there is a need for another distance function $D'()$ which measures the distances between the projections of all the N objects. $D'()$ can be transformed from the original distance

function $D()$ as follows:

$$(D'(i',j'))^2 = (D(i,j))^2 - [\frac{(D(a,i))^2 - (D(b,i))^2 - (D(a,j))^2 + (D(b,j))^2}{2 \times (D(a,b))}], \quad i,j = 1,..,N \quad (2.1)$$

where objects a and b are called *pivot objects.* The line that passes through them is the axis for the projections of all other objects. The above procedure can be generalized to the case of k dimensions, for $k \geq 1$. In that case, there should be k axes and thus, k pairs of pivot objects, and the computation will be repeated k times.

2.3 Distance-Based Index Structures

Quite a number of distance-based index structures have been proposed. A good summary of these methods can be found in [32, 6]. Previous work in [7] contains some of the basic ideas for later methods, namely the *generalized hyperplane tree* (gh-tree) [16], the *Geometric Near-neighbor Access Tree* (GNAT) [6], the *vantage point tree* (vp-tree) [36, 8], and the *multi vantage point tree* (mvp-tree) [32] which is a variation of the vp-tree.

A gh-tree partitions a dataset by first choosing two reference objects at the root level, and dividing the remaining objects based on which of the two reference objects they are closer to. Then the two branches are constructed recursively in the same way. One weakness of the gh-tree is that two distance computations are required at each node and its branching factor can only be two [6].

The GNAT generalizes the idea of the gh-tree to the case of m-ary trees. In GNAT each node stores m *split* objects. At the root level, the dataset is divided into m groups and every data object is assigned to one of the m groups according to which split object it is closest to. Similarly, each of the m groups is partitioned recursively. In [6] the GNAT was compared to the gh-tree and the binary vp-tree in a set of experiments and was found to incur more expensive construction cost but fewer distance computations in doing range queries. The experiments also showed that the gh-tree performs the worst.

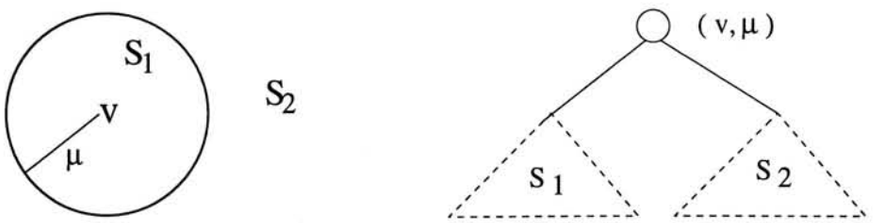


Figure 2.1: Partitioning mechanism of the vantage-point tree method.

The mvp-tree is a variation of the vp-tree. Similar to the vp-tree, the mvp-tree partitions the dataset with respect to vantage points. The key difference between vp-trees and mvp-trees is that the vp-tree uses only one vantage point at each node whereas the mvp-tree uses more than one. The mvp-tree also keeps pre-computed (at construction time) distances between the data objects and the vantage points in the leaf nodes for effective filtering of non-qualifying objects during search operations [32]. It was shown by experiments in [32] that the mvp-tree introduces fewer distance computations in range querying compared to the vp-tree. However, n-nearest neighbor search has not been considered for the GNAT as well as for the vp-tree and its variants, and the performance studies in both [32] and [6] concentrated only on the number of distance computations for range queries. In this thesis, we focus on the vp-tree structure, and thus, we shall describe it in detail next.

2.3.1 The Vantage-Point Tree Method

Consider a finite set S of N data points[2]. In the vp-tree method [36, 8], a particular data point is chosen as the first *vantage point*, 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:

$$S_1 = \{s \in S - \{v\} \mid d(s, v) < \mu\}$$
$$S_2 = \{s \in S - \{v\} \mid d(s, v) \geq \mu\}$$

[2]Since vp-trees use the term *point* to refer to data objects, we shall use the terms data points and data objects interchangeably.

Algorithm 2.1 The Select_vantage_point Algorithm.

```
Pick a set of candidate vantage points from the dataset;
For each vantage point
  Pick a set of sample objects from the dataset;
  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.
```

where $d(p, q)$ is the distance between points p and q. Figure 2.1 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. The tree is constructed in a top-down fashion. Eventually, the entire dataset is organized as a balanced tree as in other index structures.

The m-ary vp-tree construction is similar to the case of binary vp-trees. The dataset 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, $i = 1$ to $m-1$, 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.

A particular data object is selected to be the vantage point based on a randomized algorithm given in [36] as shown in Algorithm 2.1.

Chapter 3

The Problem of Distance Preserving Methods in Querying

The problem that distance preserving methods attempt to solve is defined as follows. Given N objects, their pair-wise distances and the desirable dimensionality k, distance preserving methods are required to map objects into points in k-dimensional space, so that the distances between the objects are preserved as much as possible. Basically, these methods have to overcome two main difficulties. First of all, the distances between objects must be well preserved. If the overall distances are not preserved sufficiently, some of the information that distinguishes the objects cannot be maintained. Secondly, the choice of k plays an important role in the accuracy of the mapping, but it is not always easy to determine an appropriate value for it. For most distance preserving methods, the larger the value of k, the more precisely the methods can deduce the points. Furthermore, in most multimedia datasets the number of features (or dimensions) per object is often of the order of 10 or 100 [25, 11]. As such, a large value of k should be used to help visualize the distribution of objects into some appropriately chosen space, which however implies high-dimensional vectors will be generated. Notice that indexing high-dimensional data with most multidimensional index structures usually leads to performance degradation of the structures. [22] implicitly assumes that users are responsible for the choosing of k. In that case, users

must have certain specific knowledge on the domain. However, we believe that there should be some efficient methods for determining an appropriate value of such an important parameter, k.

To handle queries, distance preserving methods cannot directly provide the indexing facility. They must rely on some conventional multidimensional index structures, such as the R-tree. The dimensionality of the generated k-d points must therefore be carefully chosen. Alternatively, our approach is to adopt distance-based index structures to offer an efficient access mechanism to answer queries. The vp-tree method is chosen for this purpose.

Next, we shall provide some experimental results that can illustrate the problem of inferring multidimensional points for objects in answering queries. As experiments in [22] showed that the FastMap algorithm is superior to previous related methods, the discussion will focus on FastMap.

3.1 Some Experimental Results

We implemented the vp-tree and FastMap in C and UNIX on an UltraSPARC. As FastMap must work in conjunction with some multidimensional indexing method to provide the search facility and the R-tree family is one popular approach, we used the original implementation of the R^*-tree by Berchtold, Keim and Kriegel [5]. For the experiment, we generated a dataset of 2000 points in a 10-dimensional space. The points form 10 clusters, with the same number of 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. The Euclidean distances between such data points are the only input for both the vp-tree and FastMap. We performed tests on 8-nearest neighbor queries relative to points chosen from the dataset. The results were averaged over the performance for 15 randomly chosen query points. Since the search algorithm presented for the original vp-tree can locate only one nearest neighbor, we adopted one of the three n-nearest neighbor search algorithms that we shall propose in Chapter 4. The R^*-tree that we used is able to handle n-nearest neighbor search.

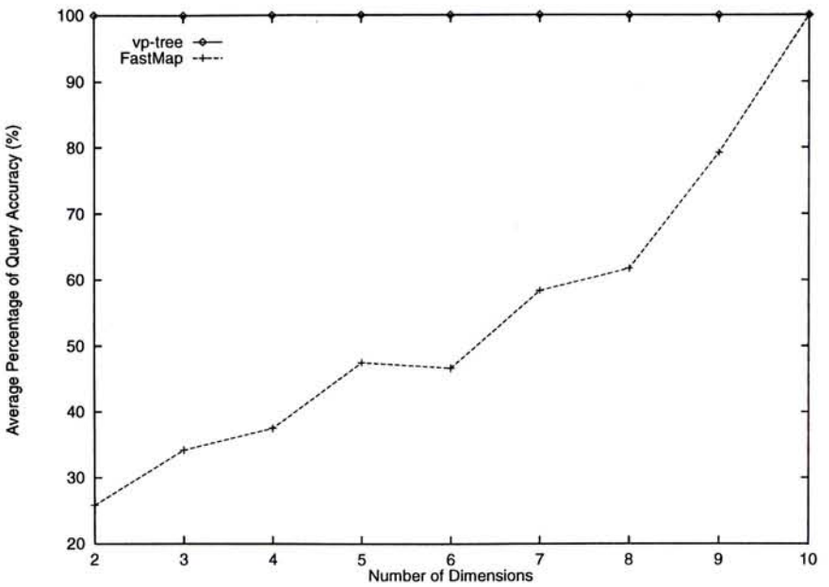


Figure 3.1: Query accuracy vs. number of dimensions.

To study the accuracy of doing n-nearest neighbor queries as the dimensionality k increases, we varied k from 2 to 10. The vp-tree and the FastMap were both required to find 8 nearest neighbors for each of the 15 query objects. In our approach a vp-tree holding 2000 data objects was constructed, then we ran our search algorithm[1] to obtain the results. Note that the value of k does not affect our method. In FastMap 9 sets of 2000 k-d points were generated, for $k = 2$ to 10. Each set of points was then organized in a distinct R^*-tree. FastMap was also required to map the 15 query objects to the corresponding k-d space, such that they were submitted to the R^*-tree whose search algorithm was run to obtain the results.

We expressed the accuracy of an 8-nearest neighbor search as a percentage of how many answers out of 8 are indeed the true nearest neighbors of the query object. We obtained the 8 true nearest neighbors by sequential scanning. We reported an average percentage over the 15 queries. Figure 3.1 plots the average percentage of query accuracy as a function of the number of dimensions. The number of dimensions is not a relevant parameter for the vp-tree method, we plotted the accuracy for it only for comparison. As seen from the figure, the vp-tree method guarantees 100% accuracy in all 15 queries. With FastMap, the lower the dimensionality, the more

[1] We used the single-pass algorithm. Details of it will be discussed in Chapter 4.

nearest neighbors have been missed. This is mainly because FastMap cannot preserve the actual distances between objects while k is getting smaller.

As every data point in the given synthetic dataset is 10-dimensional, we had expected that the set of 10-d FastMap-generated vectors would allow for a 100% query accuracy. Such an anticipation is valid only because the dimensionality of the given dataset is known in advance. In real cases where the only input would be the pairwise distances between data objects, the dimensionality k must be carefully estimated. Theoretically, k should be set as large as possible, this would however introduce significant cost in the storage requirement as well as in the search performance for most multidimensional index structures.

3.2 Discussion

While FastMap suffers from the difficulties in preserving the distances between objects and in determining a proper k value to achieve high accuracy, our method is able to locate every nearest neighbor with a fast response. 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 computations 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 satisfactory level.

2. It avoids the difficulty in preserving the actual distances between objects as faced by the Fastmap method.

3. The updates on the vp-tree are relatively easier than that for the Fastmap method. For Fastmap, 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 when is a good time for the reconstruction. By comparison, the updates for the vp-tree are much more straightforward (see Chapter 5).

4. 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). In this latter case, we use the Euclidean distances between the feature vectors.

Chapter 4

Nearest Neighbor Search in VP-trees

For content-based retrieval, it is rare to have an exact match on multimedia data, nearest neighbor queries are more desirable. The search methods for the vp-tree presented in [36, 8] locate a single nearest neighbor. We believe that finding a single nearest neighbor usually cannot satisfy users' needs. 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 to issue other queries or to do further processing. Therefore, we aim at finding n nearest neighbors to a query object, where n is usually greater than one.

The single-nearest-neighbor search algorithm in [8] relies on a specific threshold, σ, which estimates an upper bound on the distance between a query object and its nearest neighbor. If it turns out that the distance between a query object and its nearest neighbor is greater than σ, the nearest neighbor is meaningless and therefore not interesting. Let $d(p, q)$ be the distance between points p and q. Given a σ value, the algorithm in [8] will look for the single nearest neighbor to q within the range $d(v, q) \pm \sigma$ (see Figure 4.1). Recall that at each node of a vp-tree, a vantage point determines m subsets, S_i (for $i = 1$ to m), according to $m-1$ boundary distances donated by μ_i, $i = 1$ to $m-1$. If the hypersphere depicted in Figure 4.1 falls inside the

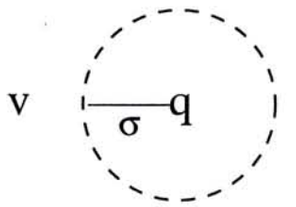


Figure 4.1: The meaning of the threshold σ.

boundary of only one subset S_i, i.e., when $\mu_{i-1} + \sigma < d(v, q) \leq \mu_i - \sigma$, the algorithm only needs to explore that particular subset. Otherwise, multiple subsets need to be explored.

The choice of σ represents the tradeoff between the likelihood of locating the nearest neighbor and the searching efforts. A tighter value of σ ensures that fewer subtrees will be explored, but it also increases the chance that no nearest neighbor is found. Clearly, the question is how the value of σ is chosen. For n-nearest neighbor search we even have to take the value of n into account. In that sense, within the hypersphere in Figure 4.1 there must exist at least n nearest neighbors.

We propose two approaches to solving the problem of n-nearest neighbor search. In the first approach, we have two different methods for estimating a σ value that **must guarantee** the presence of n nearest neighbors. Then we perform a range search with the estimated σ value. The two σ estimation methods are incorporated into two separate n-nearest neighbor search algorithms, namely the *sigma_factor* algorithm and the constant-α algorithm. In the second approach, we set the σ value to be infinitely large and dynamically optimize σ whenever we encounter a candidate answer in the search. The single-pass algorithm uses such an approach.

In the following sections we describe our methods. 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 $m > 2$.

4.1 The *sigma_factor* Algorithm

Intuitively, the value of σ should be determined based on the distribution of the distance values with respect to the vantage point. Thus, there should be a different

σ value associated with each vantage point within a vp-tree. As every vantage point must be selected during the construction of a vp-tree, we attempt to estimate the corresponding σ value in the same period of time.

Our construction algorithm for the vp-tree is similar to the original vp-tree construction algorithm in [36] except that we compute a *sigma_factor* for each node. Suppose S is the set of data objects in a node and v is the vantage point for the node, *sigma_factor* of S is given by

$$\frac{\text{distance of furthest } s \in S \text{ from } v - \text{distance of closest } s \in S \text{ from } v}{|S| - 1} \tag{4.1}$$

Therefore, *sigma_factor* is an average value for the distance we need to search for each nearest neighbor if the objects in the node are arranged in a straight line according to their distances from v, and the query is at the boundary. We can see that it is a rough guess but it is sufficient for our purpose. This *sigma_factor* is stored in the node. Each of these factors will be used to estimate σ when the query object is known.

Now we are ready to describe our first n-nearest neighbor search algorithm. The pseudocode for the algorithm is given in Algorithm 4.1. The algorithm consists of two different types of search: Sigma_factor_search and Range_search. Sigma_factor_search may be activated for one or more times. As a result, the entire search process involves two or more passes of search.

At each non-leaf level, Sigma_factor_search (Algorithm 4.1) [1] derives a different threshold (node↑.sig × n × enlarge) from the *sigma_factor* of each node to guide the exploration of the tree. After one or more activations of Sigma_factor_search, we get an initial set of at least n nearest neighbors (so far). Then we let σ be the distance between the query and the n-th neighbor in this set. This σ guarantees that the n requested nearest neighbors will at most be at a distance σ from the query object. Range_search (Algorithm 4.2) uses this σ to explore the tree again. When the range search is finished, we obtain a final set of results in which the first n closest objects from the query will be our answer set.

[1] In Algorithm 4.1, 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, node↑.right is the pointer to the right child node, and node↑.sig is the *sigma_factor* for the node. enlarge is a global variable which has an initial value of 1 when the nNN search begins, and is incremented after each invocation of Sigma_factor_search.

Algorithm 4.1 The *sigma_factor* Algorithm.

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

```
begin
  enlarge := 1;   W := φ;
  while |W| < n do
    Sigma_factor_search(q,n,root,enlarge,W);
    increment enlarge;
  endwhile
  sort w ∈ W in the order of increasing distance from q;
  σ := d(w_n, q);
  Range_search(q,n,root,σ,W);
  sort w ∈ W in the order of increasing distance from q;
  return W;
end
```

Procedure Sigma_factor_search(q,n,node,enlarge,W)
Input: q (query point), n (number of nearest neighbors requested), node (vp-tree node),
enlarge (enlargement factor), W (a set of nearest neighbors obtained so far).
Output: updated W.

```
begin
  if node is leaf then
    add node to W;
  else
    dist := d(node↑.v,q);
    if dist < node↑.mu then
      if {dist < node↑.mu + (node↑.sig × n) × enlarge} then
        Sigma_factor_search(q,n,node↑.left,enlarge,W);
      if {dist ≥ node↑.mu − (node↑.sig × n) × enlarge} then
        Sigma_factor_search(q,n,node↑.right,enlarge,W);
    else
      if {dist ≥ node↑.mu − (node↑.sig × n) × enlarge} then
        Sigma_factor_search(q,n,node↑.right,enlarge,W);
      if {dist < node↑.mu + (node↑.sig × n) × enlarge} then
        Sigma_factor_search(q,n,node↑.left,enlarge,W);
    endif
  endif
end
```

Algorithm 4.2 The Range_search Algorithm.

```
Procedure Range_search(q,n,node,σ,W)
Input: q (query point), n (number of nearest neighbors requested), node (vp-tree node),
       σ (search range threshold), W (a set of nearest neighbors obtained so far).
Output: updated W.

begin
  if node is leaf then
    if d(node,q) ≤ σ then add node to W;
  else
    dist := d(node↑.v,q);
    if dist < node↑.mu then
      if {dist < node↑.mu + σ} then
        Range_search(q,n,node↑.left,σ,W);
      if {dist ≥ node↑.mu − σ} then
        Range_search(q,n,node↑.right,σ,W);
    else
      if {dist ≥ node↑.mu − σ} then
        Range_search(q,n,node↑.right,σ,W);
      if {dist < node↑.mu + σ} then
        Range_search(q,n,node↑.left,σ,W);
    endif
  endif
end
```

In Sigma_factor_search, the value of $n \times sigma\_factor$ is used in the pruning of nodes during the search. Since *sigma_factor* represents a certain average distance between two data objects in a node, the value "$n \times sigma\_factor$" can be used to estimate σ if we are requesting n nearest neighbors. If in a pass, less than n objects are returned, we repeat Sigma_factor_search, but this time increasing the search range by adjusting an enlargement factor, the **enlarge** variable in the pseudocode. The main objective of Sigma_factor_search is to yield a set of candidate nearest neighbors from which we can achieve a good estimate on σ. Then Range_search should be able to locate the real nearest neighbors using this value of σ which is derived from a sample of nearest neighbors of the query object.

The traversal strategy of Range_search is very similar to Sigma_factor_search. The only difference is that we use σ as the search range and σ is also used to filter unqualified objects. The pseudocode description of Range_search is in Algorithm 4.2.

Algorithm 4.3 The Constant-α Algorithm.

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

```
begin
  W := φ;
  Constant_alpha_search(q,n,root,W);
  sort w ∈ W in the order of increasing distance from q;
  σ := d(w_n, q);
  Range_search(q,n,root,σ,W);
  sort w ∈ W in the order of increasing distance from q;
  return W;
end
```

Procedure Constant_alpha_search(q,n,node,W)
Input: q (query point), n (number of nearest neighbors requested), node (vp-tree node),
W (a set of nearest neighbors obtained so far).
Output: updated W.

```
begin
  dist := d(node↑.v,q);
  levels_to_traverse := a constant α;
  while levels_to_traverse > 0 do
    if dist < node↑.mu then
      node := node↑.left;
    else
      node := node↑.right;
    endif
    levels_to_traverse := levels_to_traverse − 1;
  endwhile
  W := all data objects stored in the subtree rooted at node;
end
```

4.2 The Constant-α Algorithm

In the *sigma_factor* algorithm, it is possible for the Sigma_factor_search procedure to explore multiple subtrees once the search range $d(v,q)\pm threshold$ touches multiple subsets. In that case, the search process has to make considerable access cost. Sigma_factor_search basically aims at providing information for σ estimation. It is desirable to minimize the cost incurred by it. This is the fundamental idea behind the constant-α algorithm.

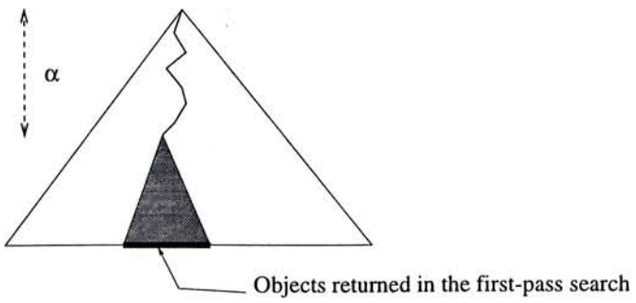


Figure 4.2: Illustration of the constant-α algorithm.

Algorithm 4.3 presents the constant-α algorithm. It has the same Range_search procedure as described in the *sigma_factor* algorithm, but a new method for estimating σ to replace the *sigma_factor* approach. The new method is implemented in Constant_alpha_search. Note that we no longer need any *sigma_factor*'s in the vp-tree. Thus, we are not required to perform the related calculations during index construction.

The traversal strategy of Constant_alpha_search mainly depends on a constant α. The algorithm traverses from the root of the tree down to the level which is equal to the constant α, provided that the level number for the root is zero. In other words, α represents the number of levels required to traverse. An important point is that the algorithm only explores the tree in a single path for α levels.

To begin with, the distance between the query object and the vantage point of the current node is first computed. The algorithm then explores the one subtree if the distance value is smaller than μ, or the other subtree if the distance value is greater than or equal to μ. Eventually, the algorithm examines a set of data objects in the subtree rooted at level α. By choosing a suitable α, Constant_alpha_search will deliver at least n objects. σ is then determined by the distance between the query object and the n-th nearest object to it. Figure 4.2 illustrates such a search. Observe that the performance can be further enhanced if the vp-tree is modified so that the leaf nodes are linked up by pointers in a sequential order, similar to the B^+-tree; and a non-leaf node x records the number of data objects in the subtree rooted at x.

The rationale behind the constant-α algorithm is based on the assumption that the nearest neighbors of an object and the object itself are likely to be organized in

Algorithm 4.4 The Single-Pass Algorithm.

```
Procedure Single_pass_search(q,n,node,σ,W)
Input: q (query point), n (number of nearest neighbors requested), node (vp-tree node),
       σ (search range threshold), W (a set of nearest neighbors obtained so far).
Output: W (a final set of n nearest neighbors to q).

begin
  if node is leaf then
    if d(node,q) ≤ σ then
      insert node to W; (W is a sorted list of objects in increasing distance from q)
      if |W| = n then σ := d(w_n, q);
  else
    dist := d(node↑.v,q);
    if dist < node↑.mu then
      if {dist < node↑.mu + σ} then
        Single_pass_search(q,n,node↑.left,σ,W);
      if {dist ≥ node↑.mu − σ} then
        Single_pass_search(q,n,node↑.right,σ,W);
    else
      if {dist ≥ node↑.mu − σ} then
        Single_pass_search(q,n,node↑.right,σ,W);
      if {dist < node↑.mu + σ} then
        Single_pass_search(q,n,node↑.left,σ,W);
    endif
  endif
end
```

the same subtree. Given that subtree, we should have a rough estimation of the n requested nearest neighbors. The Constant_alpha_search of the algorithm attempts to locate the subtree that holds enough objects for estimating a precise σ. The method achieves this by means of an optimal value of the constant α. As the value of α increases, Constant_alpha_search descends further down the tree, which leads to a smaller size of subtree to be traversed and a less precise σ. Using a less precise σ, Range_search requires more time. Therefore, an optimal α value represents a balance between the searching efforts of the two types of search.

4.3 The Single-Pass Algorithm

Both of the previous two algorithms require some initial set of candidate nearest neighbors to estimate the σ value. To guarantee a final set of results containing

n nearest neighbors, we take the distance between the query object and its n-th nearest candidate as the search range threshold (σ) for the subsequent range search. In practice, σ encloses *more* than n. During the range search, we notice that if the distance between the query object and the current n-th nearest neighbor candidate is smaller than σ, σ can be reduced to that distance value. This dynamical optimization of σ can avoid unnecessary probing in succeeding search steps.

In other words, we may set σ as infinitely large and dynamically optimize it whenever a nearer n-th neighbor is found. The single-pass algorithm uses such a simple strategy to perform the search. It does not involve any initial estimation of σ (initially, $\sigma = \infty$). Instead, we do a depth-first search; once the leaf level is reached, the algorithm keeps a record of the objects encountered, and lets σ be determined by the distance between the query object and its current n-th nearest neighbor. We believe that if we can promptly improve σ to an optimal value, many nodes will be pruned away, making the search more efficient. Algorithm 4.4 depicts such an algorithm. Note that the algorithm is triggered by the procedure call Single_pass_search(q,n,root,∞,ϕ).

4.4 Discussion

In the *sigma_factor* algorithm, *sigma_factor* refers to an average distance we need to search for each nearest neighbor of a query object. If we request n nearest neighbors for a query object, we expect the range "$n \times$ *sigma_factor*" would cover all the desired nearest neighbors in most cases. However, in the case when the nearest neighbors are distant from the query object, not every nearest neighbor falls within the said range. Then the search range has to be enlarged. As shown in the pseudocode of the algorithm, the **enlarge** variable will be incremented to increase the search range after each pass of search. Here the concern is how the range should be increased. If the increase is too much, the range will certainly be large enough to include sufficient nearest neighbor candidates, but the cost for this will be more nodes being visited. On the contrary, an inadequate enlargement of the range will cause the search to be repeated until n candidates are found. In short, the enlargement factor is an important

parameter for the *sigma_factor* algorithm.

The main drawback of the *sigma_factor* algorithm is that the process of the σ estimation may lead to more than one passes of search. The constant-α algorithm gives a guarantee of one by means of a single-path traversal of the tree. This definitely saves the search time and node accesses. The σ estimation process does not use any search range as in the *sigma_factor* method. Instead, it only examines, at each non-leaf level, which partition the query object falls into. The search then stops at the one partition that includes the query object and also contains at least n nearest neighbor candidates for deriving σ. As discussed before, how many of those objects is determined by the constant α, and we aim at finding an optimal value of it to balance the efforts between the estimation process and the subsequent range search. We see that the constant-α algorithm adopts a simpler approach to obtain the set of candidate answers.

The single-pass algorithm provides two advantages over the previous two methods. Firstly, no pre-set parameters, such as *sigma_factor*'s and α, are required. Secondly, it involves only one pass of search. It begins by descending the tree in a single-path fashion towards the leaves. As soon as n objects have been encountered, the value of σ is set as the distance between the query object and its n-th nearest object. From that point onwards, branches that exceed the σ range will get filtered out. As σ is refined to a smaller value due to much nearer objects being encountered, more and more branches will be pruned away. If σ can be quickly refined to a precise value, only a small part of the tree needs to be accessed.

4.5 Performance Evaluation

To study the performance of our three proposed n-nearest neighbor search algorithms for the vp-tree, we implemented the vp-tree and the search algorithms in C and UNIX on an UltraSPARC, and ran two sets of experiments. In the first, the three algorithms were compared with each other. In the second, we compared the vp-tree with the R^*-tree. The R^*-tree is found to be the fastest known variation of R-trees. We used the original implementation of the R^*-tree by Berchtold, Keim and Kriegel [5], which is

able to support n-nearest neighbor queries. Next we describe the setup, as well as our results and observations.

4.5.1 Experimental Setup

We used 2 synthetic datasets and 1 real dataset. The synthetic datasets are similar to the ones used in [34] and the details of them are:

- Clustered 10,20,30,40,50D: sets of 10000, 20000, 30000, 40000 and 50000 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 [-.05,+.05] relative to the cluster center in each dimension.

- Uniform 5,10,15,20,25D: sets of 10000, 20000, 30000, 40000 and 50000 uniformly distributed vectors in the interval [0,1] on each dimension.

The real dataset was provided by Berchtold, Keim and Kriegel [5] and contains about 70 Mbyte of Fourier points of variable dimensionality, representing shapes of polygons. We randomly extracted five groups (in sizes of 10000, 20000, 30000, 40000 and 50000) 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, m child pointers, and the *sigma_factor* value (for the *sigma_factor* algorithm). m denotes the branching factor. In a leaf node we keep the actual data objects (feature vectors). The branching factor of internal nodes and the maximum number of data objects contained in a leaf are determined by the page size. Tables 4.1 and 4.2 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 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.

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, $= \lfloor \frac{page-header}{inode} \rfloor$	

Table 4.1: Parameters for calculating the branching factor of internal nodes.

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

4.5.2 Results

Comparison among the three search algorithms Note that for the constant-α algorithm the value of α was set to 2 in all the following experiments because such a value results in generally good performance for all the datasets we used.

In Figure 4.3 we present the performance of the three algorithms on various sizes of synthetic clustered datasets. Each algorithm was required to answer 8-nearest neighbor queries. As expected, we see that the constant-α algorithm performs better than the *sigma_factor* algorithm and the single-pass algorithm provides the best results. Since the σ estimation involved in the *sigma_factor* algorithm may require multiple-path searching and may also lead to more than one passes of search, this algorithm has to make considerable effort in handling the queries. As opposed to the *sigma_factor*

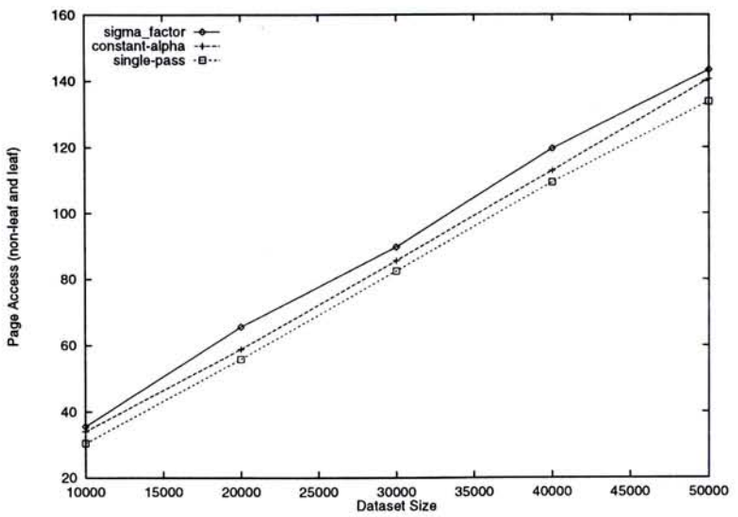


Figure 4.3: Page accesses vs. dataset size. Comparison among the three proposed algorithms on synthetic clustered data, dimension=30, #nearest neighbors=8.

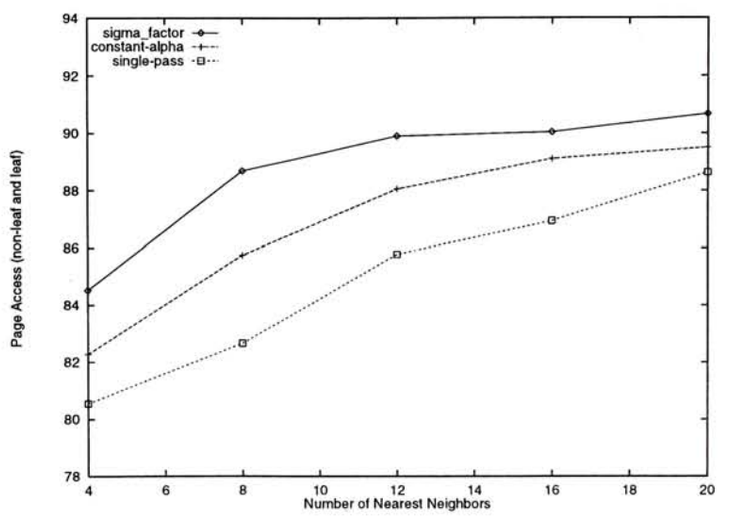


Figure 4.4: Page accesses vs. number of nearest neighbors requested. Comparison among the three proposed algorithms on synthetic clustered data, dataset size=30000, dimension=30.

algorithm, the constant-α method guarantees its σ estimation must be completed in one pass of a search that can eliminate multiple-path traversal. Consequently, the constant-α method requires on average 6% fewer page accesses than the *sigma_factor* algorithm. Because the single-pass algorithm does not need an extra pass (or passes) to do σ estimation, it has the best performance, making around 6% and 11% savings in page accesses over the constant-α algorithm and the *sigma_factor* method respectively.

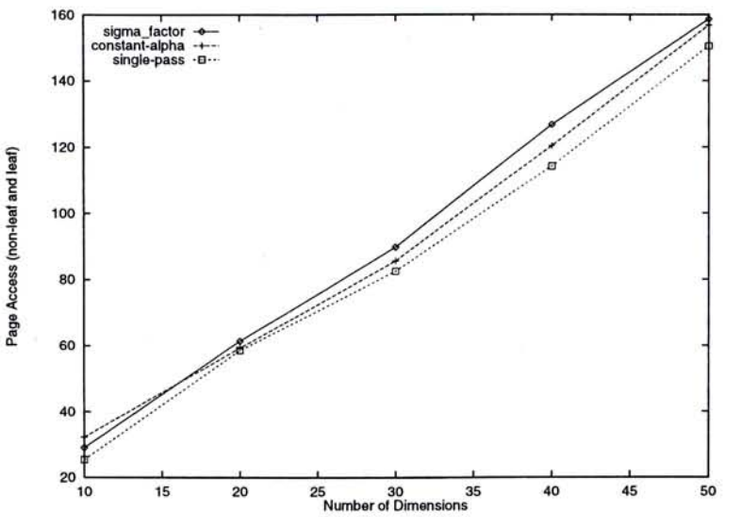


Figure 4.5: Page accesses vs. number of dimensions. Comparison among the three proposed algorithms on synthetic clustered data, dataset size=30000, #nearest neighbors=8.

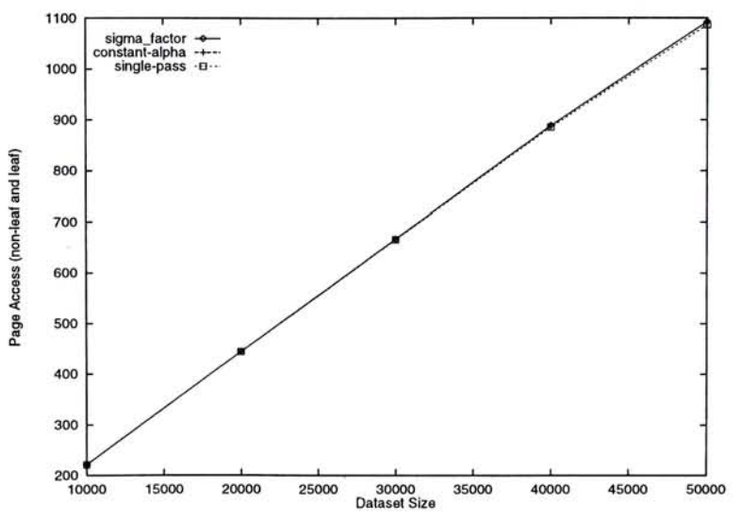


Figure 4.6: Page accesses vs. dataset size. Comparison among the three proposed algorithms on synthetic uniform data, dimension=20, #nearest neighbors=8.

To see the dependency on the desired number of nearest neighbors, we ran n-nearest neighbor queries, for n=4, 8, 12, 16 and 20, on the dataset of 30000 objects. Figure 4.4 gives the page accesses versus the number of nearest neighbors requested. The single-pass algorithm is still better than the other two methods although the actual differences between the three methods are not very significant.

Figure 4.5 illustrates the results as the dimensionality of the data increases. Again, the single-pass algorithm achieves the best performance. The amounts of savings it

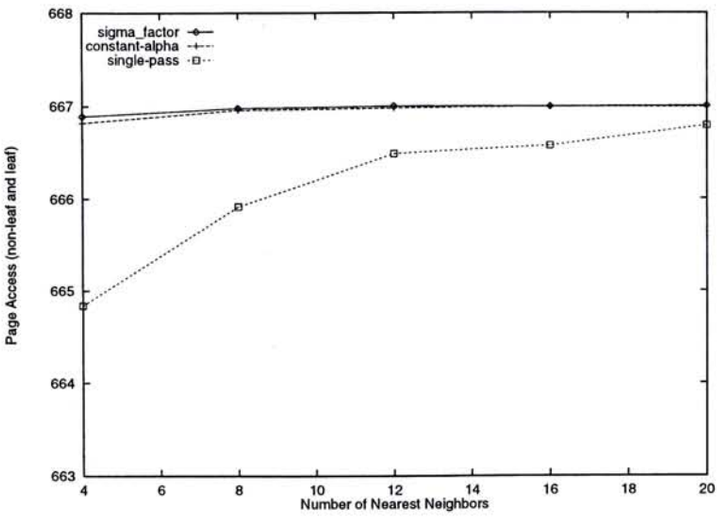


Figure 4.7: Page accesses vs. number of nearest neighbors requested. Comparison among the three proposed algorithms on synthetic uniform data, dataset size=30000, dimension=20.

makes in page accesses are similar to those for different sizes of datasets.

Then, we conducted similar experiments on the synthetic uniform data. As the data objects of a uniform dataset are far apart, finding n nearest neighbors to a query object will most likely require a search over a very large portion of the search space. This explains why the three algorithms access almost the same number of pages as the dataset size increases (Figure 4.6). In fact, we recognized that all three algorithms had searched over 95% of the total nodes of the index tree. This result is consistent with the performance reported in other indexing techniques [6, 32].

For the same reason, the three algorithms visit relatively the same number of pages for varying numbers of nearest neighbors (Figure 4.7). It seems that there is a gap between the single-pass algorithm and the other two, but the largest difference attained at 4-nearest-neighbors is indeed only about 0.3%.

From Figure 4.8 we observe that the single-pass algorithm gives good results for dimensions lower than 15. It needs up to 43% fewer page accesses compared to the *sigma_factor* algorithm. Recall that the superiority of the single-pass algorithm depends on how fast the σ value is dynamically improved. For uniform datasets where every nearest neighbor candidate is distant from the query object, the algorithm has to take a large amount of time to optimize σ. Thus, it cannot maintain its good

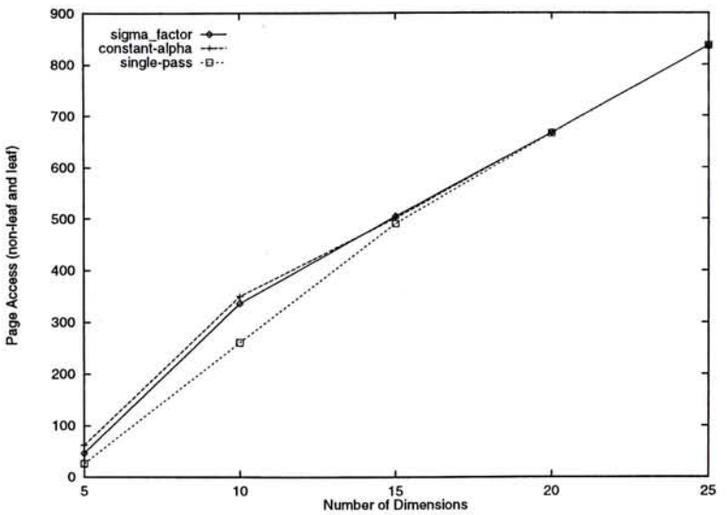


Figure 4.8: Page accesses vs. number of dimensions. Comparison among the three proposed algorithms on synthetic uniform data, dataset size=30000, #nearest neighbors=8.

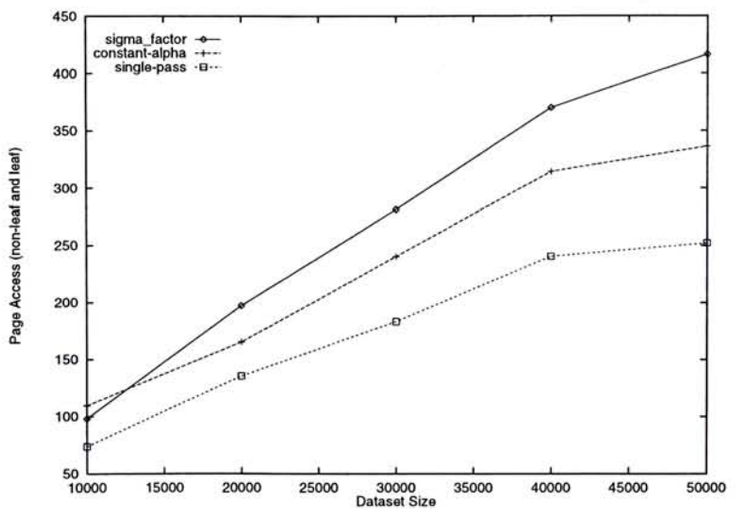


Figure 4.9: Page accesses vs. dataset size. Comparison among the three proposed algorithms on real data, dimension=16, #nearest neighbors=8.

performance as for clustered datasets. However, data in low dimensions are not as scattered as in high dimensions, leading to some of the nearest neighbors being close to the query object. In that case, the single-pass algorithm is able to perform better.

Figures 4.9-4.11 display the results for the real data. The three algorithms behave consistently as in the experiments for the synthetic clustered datasets, in that the single-pass algorithm performs the best. In Figure 4.9 the *sigma_factor* algorithm appears to perform better than the constant-α algorithm with small sizes of datasets.

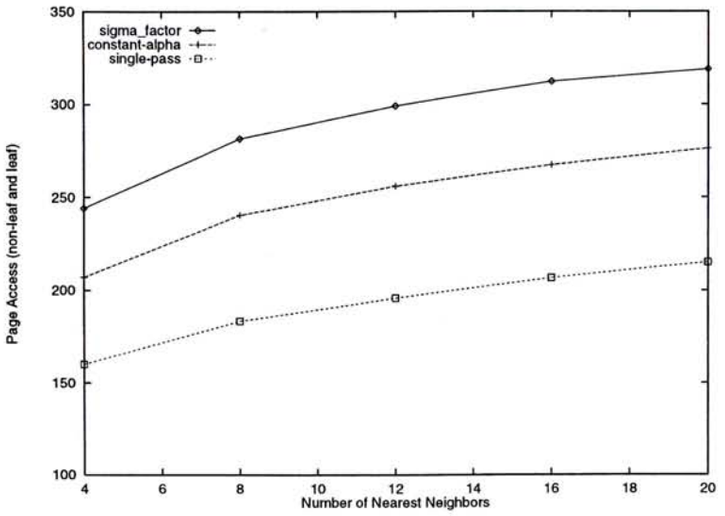


Figure 4.10: Page accesses vs. number of nearest neighbors requested. Comparison among the three proposed algorithms on real data, dataset size=30000, dimension=16.

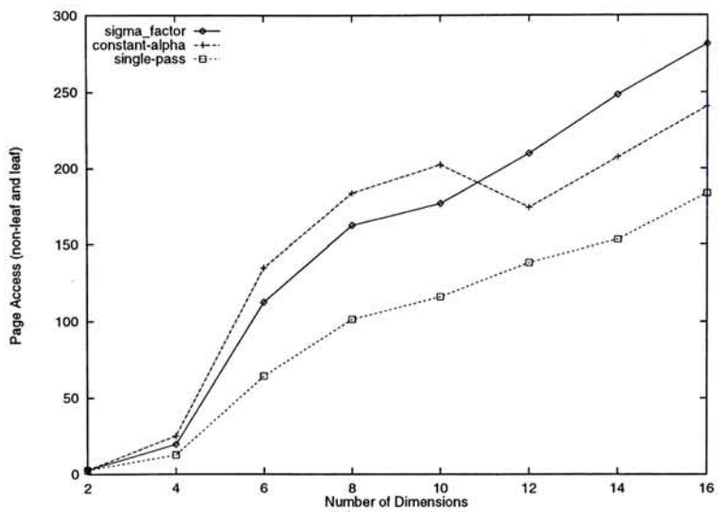


Figure 4.11: Page accesses vs. number of dimensions. Comparison among the three proposed algorithms on real data, dataset size=30000, #nearest neighbors=8.

Similarly, Figure 4.11 also show that the *sigma_factor* algorithm performs better than the constant-α algorithm for dimensions lower than 10. It is mainly because the vp-trees for low-dimensional datasets are so shallow that we cannot find an optimal value for α. As the datasets described in Figure 4.9 are all in dimensions of 16 which is lower than those in Figure 4.3, the above trend does not occur in Figure 4.3 even the dataset sizes in both figures are the same.

We can summarize our observations as follows:

- The single-pass algorithm performs better than both the *sigma_factor* algorithm and the constant-α algorithm.
- Our experiments show that the constant-α algorithm is not applicable to shallow vp-trees.
- The performance of all three algorithms deteriorates rapidly for uniform data.
- The search cost required by all three algorithms is insensitive to the amount of nearest neighbors requested.

Comparison with the R^*-tree The partitioning strategies adopted in vp-trees and R^*-trees are different, in that 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). So far as we know, none of the previous work compared the performance of the two structures. However, we believe such a comparison is significant because distance-based index methods can be applied to both the distance case and the vector space case. In fact, some recent work has proposed that distance-based indexing is one solution to the problem of indexing high-dimensional spaces [32].

Note that in all comparisons with the R^*-tree, the single-pass algorithm was used to perform the search, and 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 4.12-4.14 show the performance of the vp-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 R^*-tree.

Figure 4.12 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 vp-tree makes 55-65% savings in page accesses. The savings increase with the size of the datasets, indicating that the vp-tree method scales up well. For different numbers of nearest neighbors, we see from Figure 4.13 that the vp-tree makes around 54% fewer

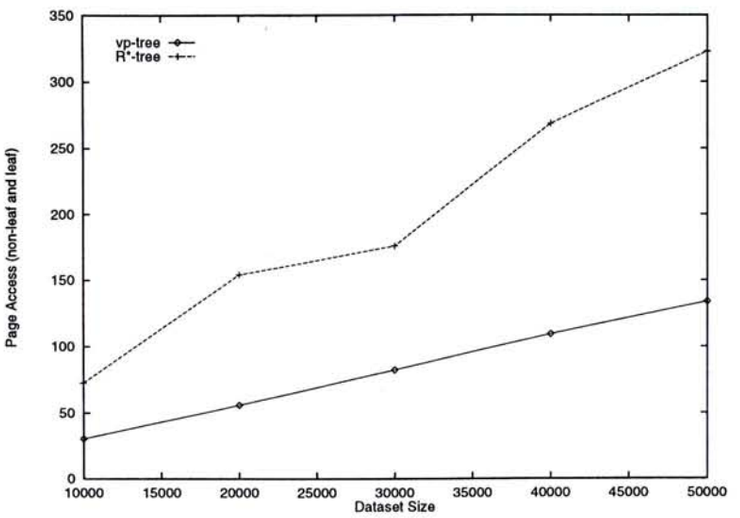


Figure 4.12: Page accesses vs. dataset size. Comparison with the R^*-tree on synthetic clustered data, dimension=30, #nearest neighbors=8.

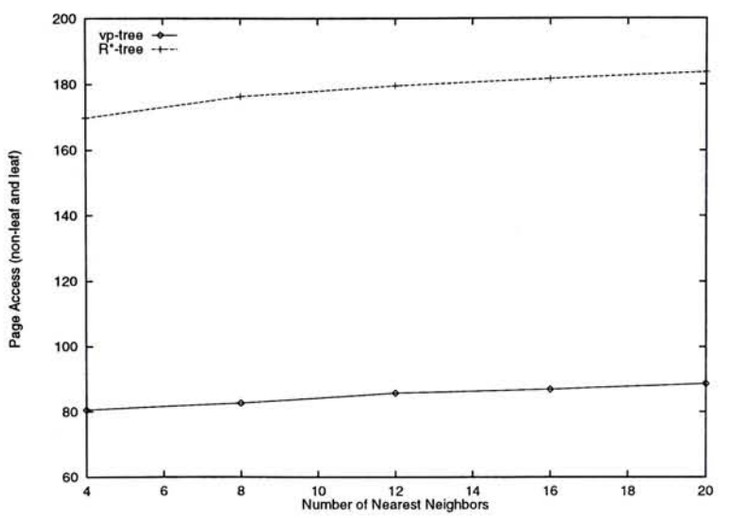


Figure 4.13: Page accesses vs. number of nearest neighbors requested. Comparison with the R^*-tree on synthetic clustered data, dataset size=30000, dimension=30.

page accesses than the R^*-tree. Notice that the increment in the number of nearest neighbors leads to only a small increase in the search effort for both vp-trees and R^*-trees. On the other hand, the impact of the dimensionality on the vp-tree and on the R^*-tree greatly differs from one another (see Figure 4.14). As the dimensionality increases (from 10 to 50), the vp-tree visits 39-62% fewer pages compared to the R^*-tree. The curves illustrate that the vp-tree scales up better with the dimensions than the R^*-tree does, and also provide further support for the findings made by previous work that R-trees stop being efficient for dimensionalities greater than 20 [10, 14, 32].

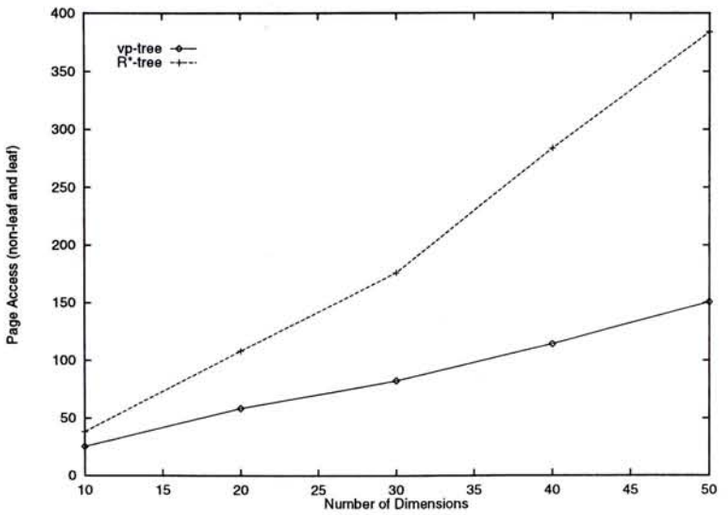


Figure 4.14: Page accesses vs. number of dimensions. Comparison with the R^*-tree on synthetic clustered data, dataset size=30000, #nearest neighbors=8.

The reasons why the vp-tree achieves better performance are discussed as below. (a) Since in an R^*-tree two limits for a closed bounded interval are stored on each dimension[2], 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. (b) 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. (c) The disk blocks used by our vp-tree were highly utilized. As the R^*-tree implementation focused on other issues, such as the prevention of overlap between bounding boxes, the utilization rate was not as high as the vp-tree's.

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 4.15-4.17 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 savings in page accesses increase with the size of the datasets, and both structures do not respond strongly to the increase of the number of nearest neighbors. For various dataset sizes, there are around 33% savings in page accesses (Figure 4.15). For the number of nearest neighbors from 4 to 20, the corresponding savings are

[2]See Section 2.1 for a more detailed description on the structure of the R-tree.

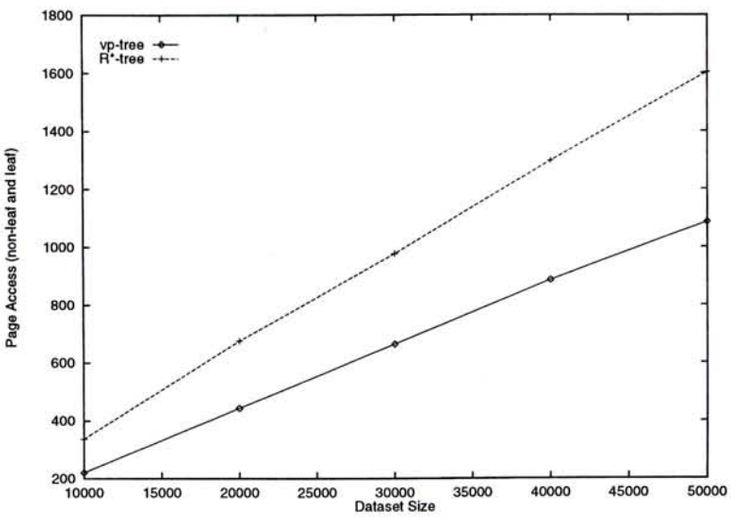


Figure 4.15: Page accesses vs. dataset size. Comparison with the R^*-tree on synthetic uniform data, dimension=20, #nearest neighbors=8.

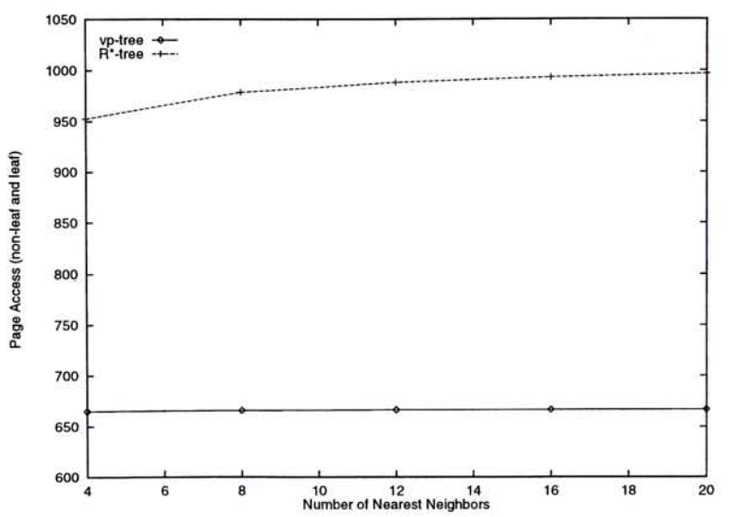


Figure 4.16: Page accesses vs. number of nearest neighbors requested. Comparison with the R^*-tree on synthetic uniform data, dataset size=30000, dimension=20.

around 32% (Figure 4.16). The savings achieved by the vp-tree can also be explained by the three reasons we have mentioned before. However, we can see that these savings are lower than those 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 in Figure 4.17 exhibit a slightly different trend from the above. In lower dimensions (5-12) the R^*-tree performs better than the vp-tree. But in dimensions higher than 12, the vp-tree gives better results. This indicates that R^*-trees can

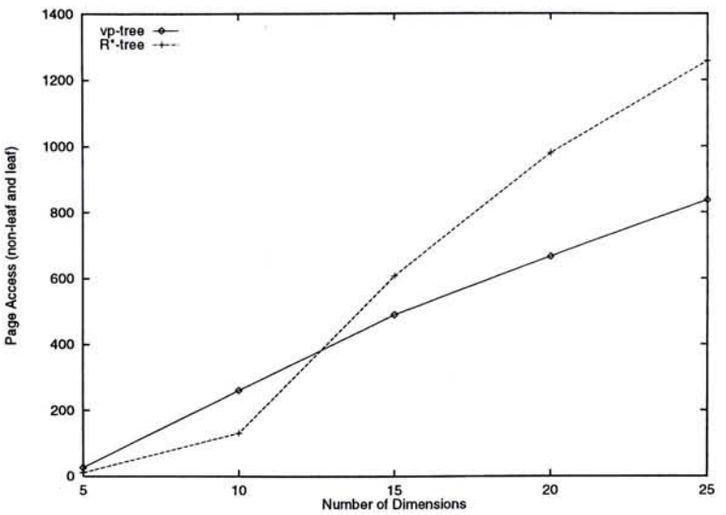


Figure 4.17: Page accesses vs. number of dimensions. Comparison with the R^*-tree on synthetic uniform data, dataset size=30000, #nearest neighbors=8.

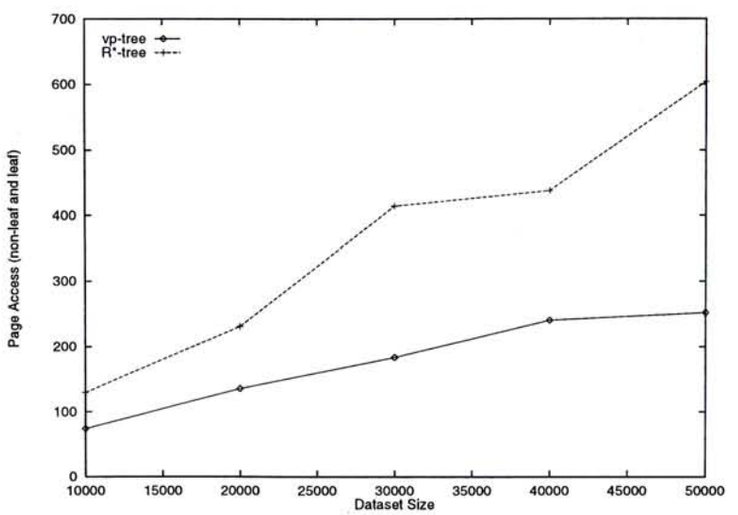


Figure 4.18: Page accesses vs. dataset size. Comparison with the R^*-tree on real data, dimension=16, #nearest neighbors=8.

provide fast query performance for low-dimensional data although the performance degrades with the dimensions.

For the real data, we first studied the dependency on the dataset size. We used datasets in five sizes (10000, 20000, 30000, 40000 and 50000) and fixed the dimensionality at 16. Figure 4.18 presents the number of page accesses versus the dataset size. The vp-tree performs better than the R^*-tree, making 43-61% fewer page accesses. The gap seems to open up as the dataset size increases. Figure 4.19 gives the performance results for varying numbers of nearest neighbors. The test dataset contained

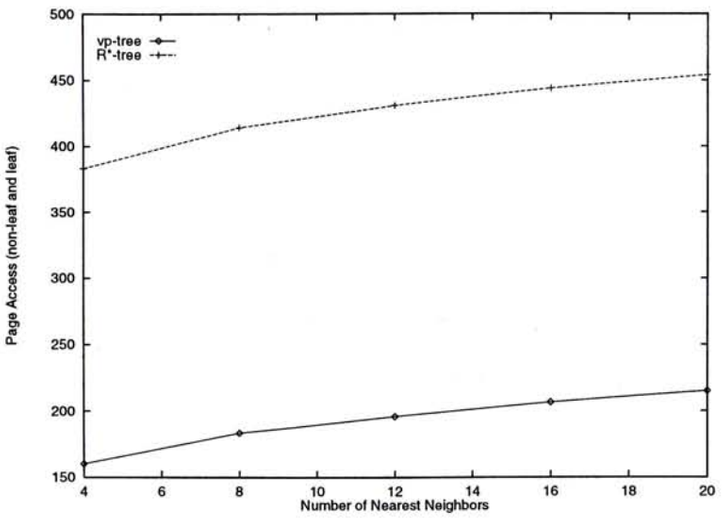


Figure 4.19: Page accesses vs. number of nearest neighbors requested. Comparison with the R^*-tree on real data, dataset size=30000, dimension=16.

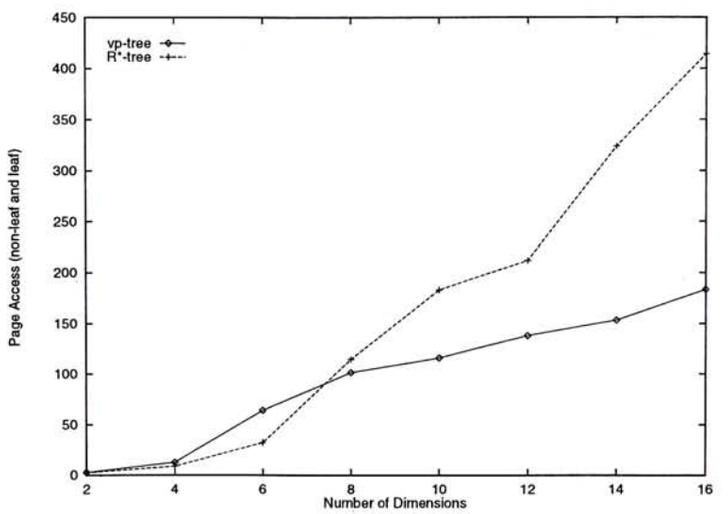


Figure 4.20: Page accesses vs. number of dimensions. Comparison with the R^*-tree on real data, dataset size=30000, #nearest neighbors=8.

30000 16-dimensional objects. Again, the vp-tree outperforms the R^*-tree, with an average of 58% fewer page accesses. The same dataset size (30000 objects) was chosen to experiment the impact of the dimensionality of data. As shown in Figure 4.20, both structures behave in a similar way as in the experiments on the uniform data (Figure 4.17). The R^*-tree gives better results in dimensions from 2 to 7, while the vp-tree performs much better in higher dimensions.

Since the overall dimensions of the real data are considerably lower than those of the clustered ones, even though the vp-tree achieves similar percentages of savings

in the query cost, both vp-trees and R^*-trees have in fact put more effort into doing the search on the real data. By observing the actual number of page accesses being reported in Figures 4.18-4.20, it is clearly seen that the query cost required for the real data is larger than that for the clustered, but smaller than that for the uniform. As mentioned before, objects in uniform datasets are distant from each other, 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 [5], like most of the real datasets, is correlated or clustered. But 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.

Chapter 5

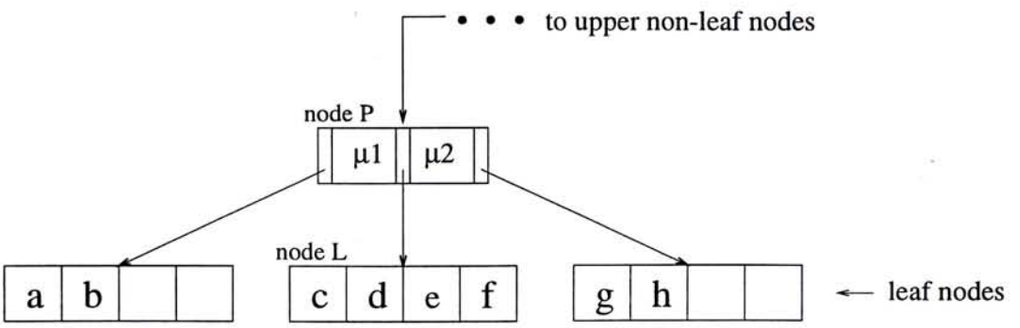
Update Operations on VP-trees

Because of the top-down partitioning strategy, updates of the vp-tree are complex to manage and a global reorganization of the structure may often result. Unlike the B-tree and its variants, we cannot conveniently split a node and propagate the splitting up the vp-tree, this is because the partitioning at the parent node affects the partitioning at the child nodes. As such, handling update operations that can maintain a balanced tree with minimal restructuring has been left as an open problem.

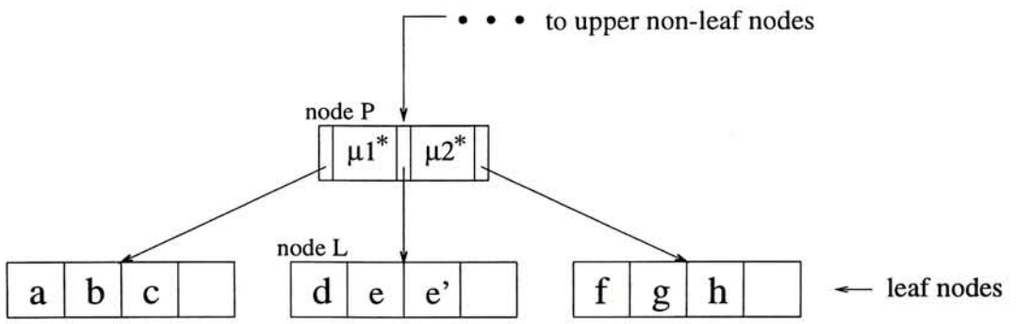
We detail the procedures for doing inserts and deletes on the vp-tree. We assume an m-ary vp-tree. The branching factor of non-leaf nodes and the maximum number of objects contained in a leaf are determined by the page size. For a non-leaf and non-root node, the number of child nodes varies between a minimum value and a maximum value m, as in a B-tree. The 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, 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, let P be the parent node of L, and we employ the following strategy.



(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.

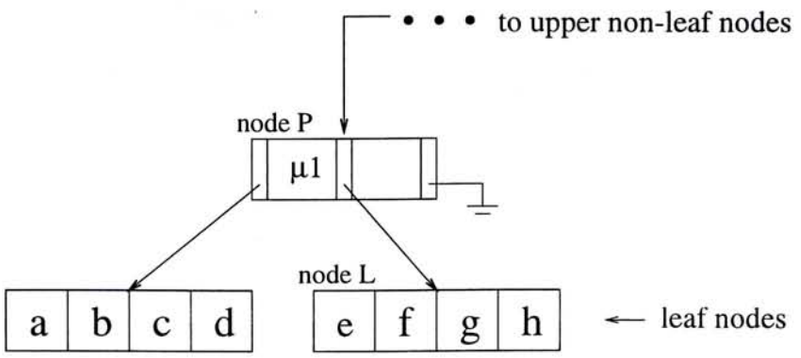
Figure 5.1: Redistribution among leaf nodes.

Examples are given in Figures 5.1 to 5.4.

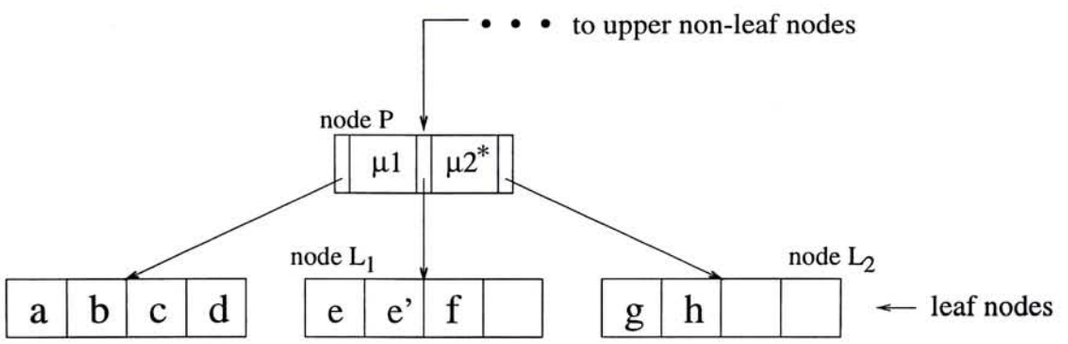
1. If any sibling leaf node of L is not full, redistribute all objects under P among the leaf nodes (Figure 5.1).

   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 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 i = 1 *to* F−1



(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 L_1 and L_2, and e' gets inserted.

Figure 5.2: Splitting of leaf node.

$P\uparrow.\mathsf{mu_i} := (\max\{\mathsf{d(v, S_j)} \mid \forall \mathsf{S_j} \in \mathsf{SS_i}\} + \min\{\mathsf{d(v, S_j)} \mid \forall \mathsf{S_j} \in \mathsf{SS_{i+1}}\}) \div 2;$

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

$P\uparrow.\mathsf{child_i} :=$ the leaf node containing $\mathsf{SS_i}$.

2. Else if the parent P has room for one more child, split the leaf node L (Figure 5.2).

   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.

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

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

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

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

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

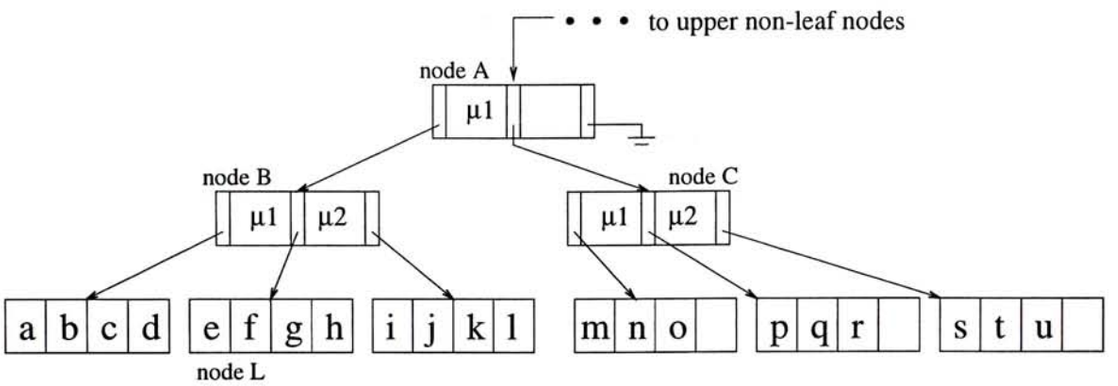
$P\uparrow.\mathsf{child}_{\mathsf{k}}$:= the leaf node containing SS_1;

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

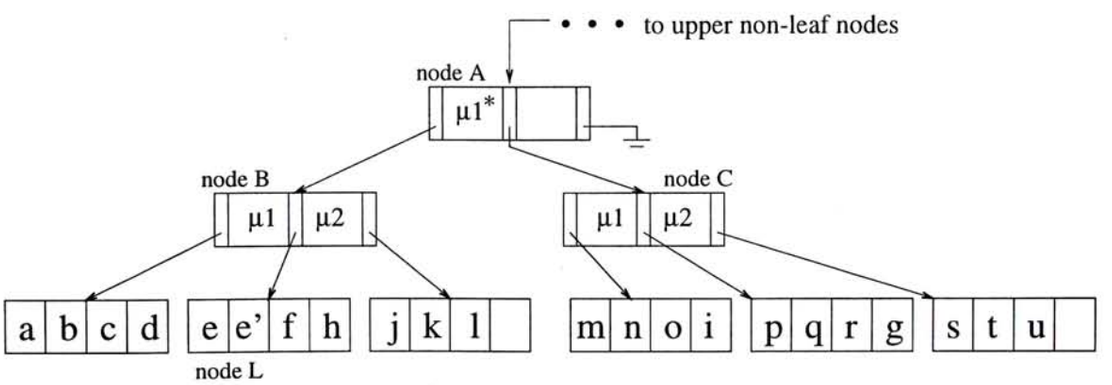
3. Else, find a nearest ancestor A of L that is not full. Let B be the immediate child node of A, and B is also the ancestor of L.

   (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 5.3).

   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)-th subtrees. B can be either the k-th or the (k+1)-th subtree. Let Num(k) and Num(k+1) be the number of objects stored in the k-th and the (k+1)-th 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) of the farthest distances from A's vantage point will be moved to the (k+1)-th subtree, so that both subtrees will eventually hold the same number of objects. Certainly, the boundary distances and pointers involved in the subtrees will be updated accordingly. On the other hand, if we find that the (k+1)-th subtree holds more objects, its objects



(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 5.3: Redistribution among subtrees.

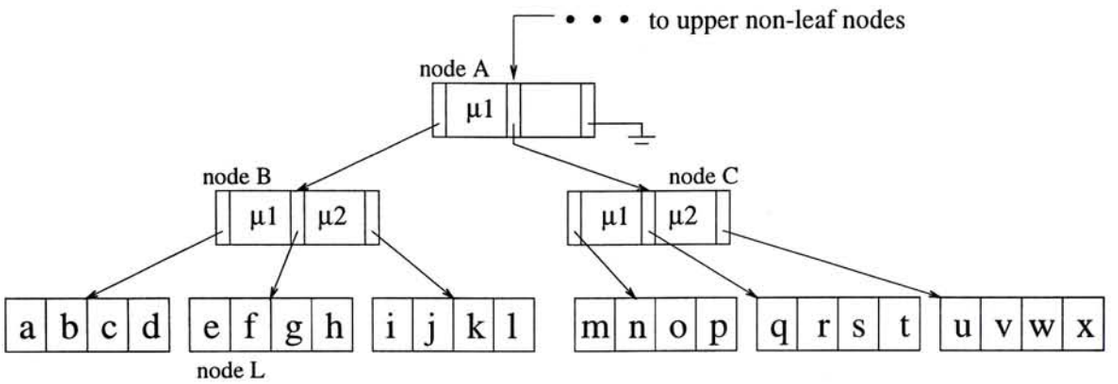
Algorithm 5.1 Algorithm for redistributing objects between two adjacent subtrees.

```
begin
  average := floor(Num(k) + Num(k + 1)) ÷ 2;

  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, SS1 and SS2 in order, where
        SS1 = {S1, S2, ..., S_Num(k)−w} and
        SS2 = {S_Num(k)−w+1, S_Num(k)−w+2, ..., S_Num(k)};

      for all Si ∈ SS2
          delete Si from the k-th subtree;
      A↑.muk := (max{d(v, Sj) | ∀Sj ∈ SS1} + min{d(v, Sj) | ∀Sj ∈ SS2}) ÷ 2;
      for all Si ∈ SS2
          reinsert Si to the (k+1)-th subtree;
  else
      Let S be the set of objects stored in the (k+1)-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+1)-th subtree to
      k-th subtree;
      w := Num(k+1) − average;

      Divide S into 2 subsets, SS1 and SS2 in order, where
        SS1 = {S1, S2, ..., Sw} and SS2 = {Sw+1, Sw+2, ..., S_Num(k+1)};

      for all Si ∈ SS1
          delete Si from the (k+1)-th subtree;
      A↑.muk := (max{d(v, Sj) | ∀Sj ∈ SS1} + min{d(v, Sj) | ∀Sj ∈ SS2}) ÷ 2;
      for all Si ∈ SS1
          reinsert Si to the k-th subtree;
  endif
end
```

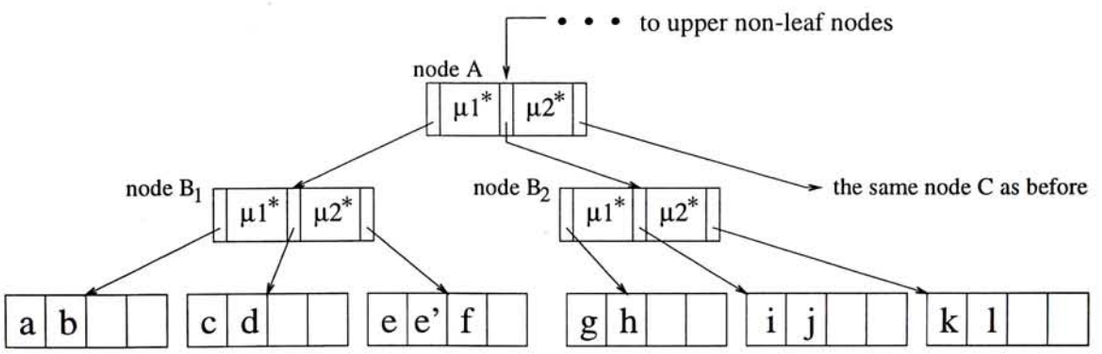
that are the closest to A's vantage point will be moved to the k-th subtree. Algorithm 5.1 gives the pseudocode description of the redistribution of objects between two adjacent subtrees.

(b) Else if A has room for one more child, split the non-leaf node B (Figure 5.4).

Assume B is the k-th child of A. Retrieve all objects stored in the subtree



(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 to B_1 and B_2, and e' gets inserted. Note that C remains unchanged.

Figure 5.4: Splitting of non-leaf node.

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 SS_1 and 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 SS_1 and SS_2 respectively.

for $i = k$ *to* $F-1$

$A\uparrow.mu_{i+1} := A\uparrow.mu_i$;

$A\uparrow.mu_k := (\max\{d(v, S_j) \mid \forall S_j \in SS_1\} + \min\{d(v, S_j) \mid \forall S_j \in SS_2\}) \div 2$;

for $i = k+1$ *to* F

$A\uparrow.child_{i+1} := A\uparrow.child_i$;

$A\uparrow.child_k :=$ make_vp_tree(SS_1);

$A\uparrow.child_{k+1} :=$ make_vp_tree(SS_2).

The insert 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.

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 $MIN_{data}(E)$ denotes the minimum number of objects that should be stored in the subtree rooted at node E, and is defined as:

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

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

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

1. If the parent node P of L 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, if the parent P underflows, 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. Algorithm 5.2 describes such a merge.

   Case 1: 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.

   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.

   Case 3: if the total spare room of the (k+1)-th 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 Algorithm 5.2). We then move mid objects from B to the (k-1)-th subtree and the rest to the (k+1)-th subtree, and delete B.

Algorithm 5.2 Algorithm for merging adjacent subtrees.

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 i = k *to* F−2
$A\uparrow.\mathsf{mu}_i := A\uparrow.\mathsf{mu}_{i+1}$;
for all $\mathsf{S}_i \in \mathsf{S}$
insert S_i to the (k+1)-th subtree;

elseif Case 2 *then*
for i = k−1 *to* F−2
$A\uparrow.\mathsf{mu}_i := A\uparrow.\mathsf{mu}_{i+1}$;
for all $\mathsf{S}_i \in \mathsf{S}$
insert 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)} ÷ 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, SS_1 and SS_2 in order, where
$\mathsf{SS}_1 = \{\mathsf{S}_1, \mathsf{S}_2, ..., \mathsf{S}_{\mathsf{mid}}\}$ and $\mathsf{SS}_2 = \{\mathsf{S}_{\mathsf{mid}+1}, \mathsf{S}_{\mathsf{mid}+2}, ..., \mathsf{S}_{\mathsf{Num(k)}}\}$;

$A\uparrow.\mathsf{mu}_{k-1} := (\max\{\mathsf{d}(\mathsf{v}, \mathsf{S}_j) \mid \forall \mathsf{S}_j \in \mathsf{SS}_1\} + \min\{\mathsf{d}(\mathsf{v}, \mathsf{S}_j) \mid \forall \mathsf{S}_j \in \mathsf{SS}_2\}) \div 2$;
for i = k *to* F−2
$A\uparrow.\mathsf{mu}_i := A\uparrow.\mathsf{mu}_{i+1}$;
for all $\mathsf{S}_i \in \mathsf{SS}_1$
insert S_i to the (k-1)-th subtree;
for all $\mathsf{S}_i \in \mathsf{SS}_2$
insert S_i to the (k+1)-th subtree;
endif

for i = k *to* F−1
$A\uparrow.\mathsf{child}_i := A\uparrow.\mathsf{child}_{i+1}$;
end

(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.

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

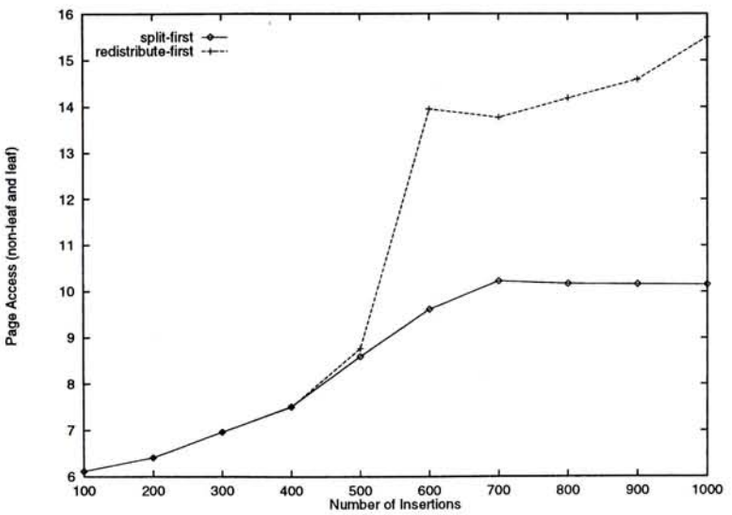


Figure 5.5: Page accesses vs. number of insertions on a synthetic clustered dataset of 10000 objects.

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, MIN_{leaf} for leaf nodes or MIN_{data} for non-leaf nodes).

5.3 Performance Evaluation

We conducted a number of experiments to show the correctness and the performance of our insert and delete 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 10000 objects. A separate vp-tree was constructed to organize the objects of each of the three samples[1]. Then we inserted 1000 new objects into each tree with the redistribute-first strategy as well as the split-first strategy. For every 100 insertions, we measured the average page accesses required for all the objects inserted so far. Figure 5.5 plots the results for the clustered sample. We also counted the times

[1]Details of the data samples have been given in Section 4.5.1

Intervals of insertions	Node splitting		Redistribution	
	occurrence	avg. cost (pages)	occurrence	avg. cost (pages)
1 - 100	0	0	0	0
101 - 200	0	0	0	0
201 - 300	0	0	0	0
301 - 400	0	0	0	0
401 - 500	0	0	6	34.67
501 - 600	1	38.00	38	84.05
601 - 700	0	0	2	50.00
701 - 800	0	0	10	53.40
801 - 900	0	0	6	95.50
901 - 1000	0	0	17	67.47

Table 5.1: 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 - 100	0	0	0	0
101 - 200	0	0	0	0
201 - 300	0	0	0	0
301 - 400	0	0	0	0
401 - 500	3	38.00	0	0
501 - 600	5	38.00	9	44.44
601 - 700	0	0	5	88.60
701 - 800	0	0	1	55.00
801 - 900	0	0	1	57.00
901 - 1000	0	0	1	32.00

Table 5.2: Access cost of splits and redistributions at non-leaf nodes with the split-first strategy - synthetic clustered sample.

that node splitting and redistribution had occurred at each interval of 100 insertions. Tables 5.1 and 5.2 show the count and the associated cost in page accesses for the redistribute-first and split-first strategies respectively. 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 inserting the first 400 objects does not involve any splitting or redistribution at non-leaf nodes, which suggests that the cost of inserting these objects is entirely due to splits and redistributions among leaves. As such operations are not costly, the curves in Figure 5.5 for both strategies increase slowly

with the number of insertions.

The two strategies start to behave differently after 400 objects have been inserted. From this point onwards, nodes tend to be fuller. There is an increasing need for splitting and redistribution among subtrees.

During the insertion of the next 100 objects, the redistribute-first strategy has chosen to do 6 redistributions with an average cost of 34.67 page accesses. On the other hand, the split-first strategy has chosen to split 3 nodes, the average page accesses required for each split are 38. As seen from Figure 5.5, the difference in terms of the total access cost made by the two strategies at that point is indeed small. However, when the number of insertions rises from 500 to 600, there the curve for the redistribute-first strategy shows a definite jump. 38 redistributions and 1 split have occurred. We recognize that prior insertions have already moved the nodes close to saturation. Hence, more and more redistributions are required, and each of these redistributions is very costly because a large number of subtrees are involved. When no more redistribution can be done, the strategy has to do the one node splitting. Even though one split has occurred, subsequent insertions still result in costly redistributions because there exist many full nodes. All these explain the high average cost for those 38 redistributions and the big jump shown in the figure.

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 for 700 insertions. Soon after that, nodes become fuller and fuller due to the insertion of the last 300 objects. The trend there resembles the one at the beginning. We can expect that the pattern between 500 and 1000 insertions of the curve will repeat continuously until the point when the whole tree is full.

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, the overall access cost made by the split-first strategy is much lower than the redistribute-first. In Figure 5.5 the curve for the split-first strategy stops increasing from 700 to 1000 insertions, for the reason that the utilization of nodes is generally low after certain number of splits and redistributions.

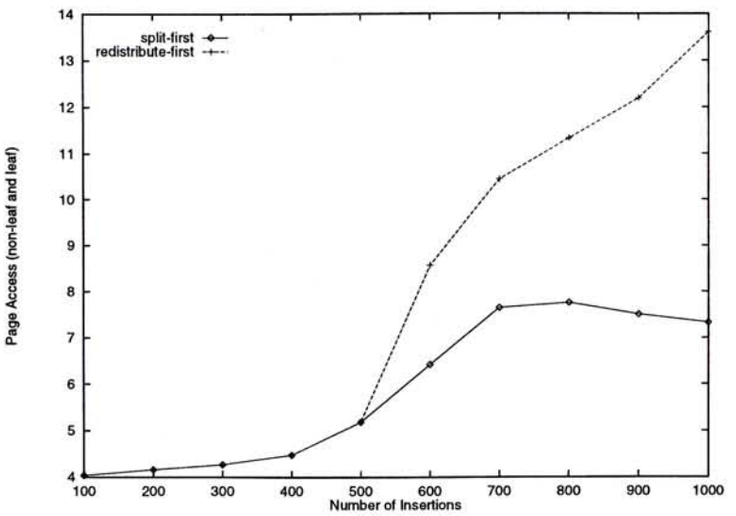


Figure 5.6: Page accesses vs. number of insertions on a synthetic uniform dataset of 10000 objects.

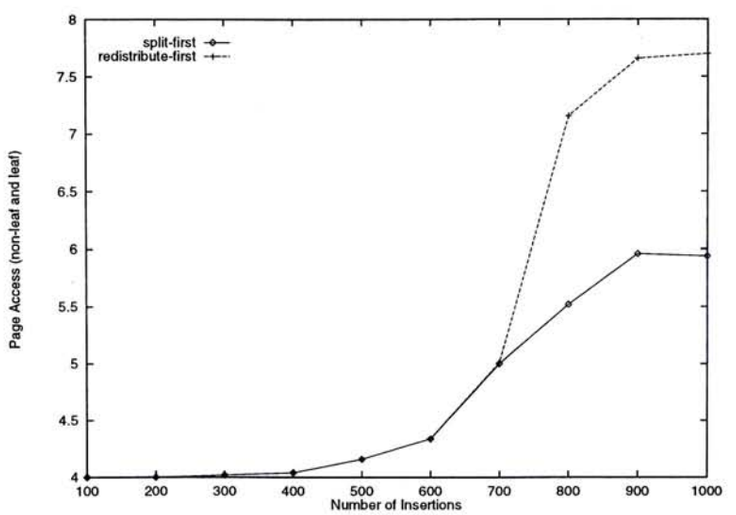


Figure 5.7: Page accesses vs. number of insertions on a real dataset of 10000 objects.

The results for the uniform and real data are shown in Figures 5.6 and 5.7 respectively. We can see from the figures that the two strategies exhibit a similar trend for both uniform and real samples as for the clustered one, and hence, the detailed counts of corresponding splits and redistributions are omitted for brevity. Since the tree for the clustered data is one level deeper than the trees for the other two samples due to a higher dimensionality, the insertions on the clustered sample require relatively more page accesses.

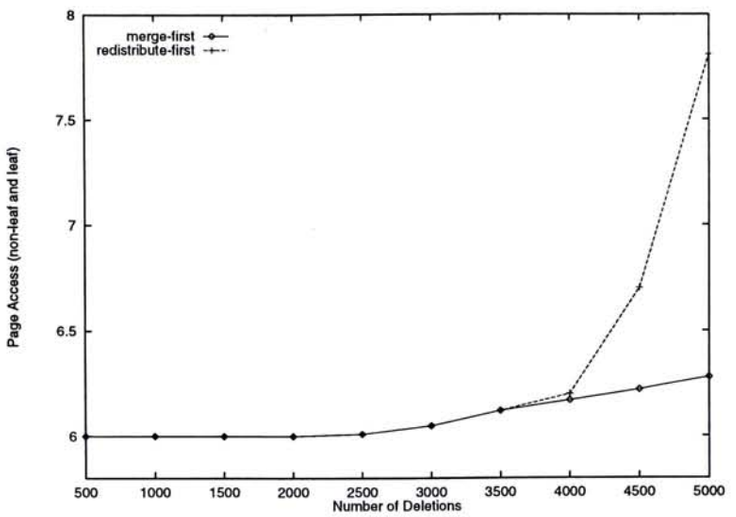


Figure 5.8: Page accesses vs. number of deletions on a synthetic clustered dataset of 10000 objects.

For measuring delete performance, we removed 5000 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 500 deletions, we measured the average page accesses required for all the objects deleted so far. Figures 5.8-5.10 give the results for the clustered, uniform and real samples. All the curves show a similar trend. Given that the heights of the trees for clustered and uniform samples are 3 and 2 respectively, we observe that the first 2000 deletions for both samples require only the corresponding minimum cost of deletes (6 and 4 page accesses respectively) with both merge-first and redistribute-first strategies. This is also true for the first 3000 deletions for the real data sample as the tree height for this sample is 2. In other words, deleting such amounts of objects has not caused any underflows.

However, as more and more objects are removed, redistributions or merges occur more often. Consequently, the average page accesses made increase 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 do not occur as frequently as in the case of

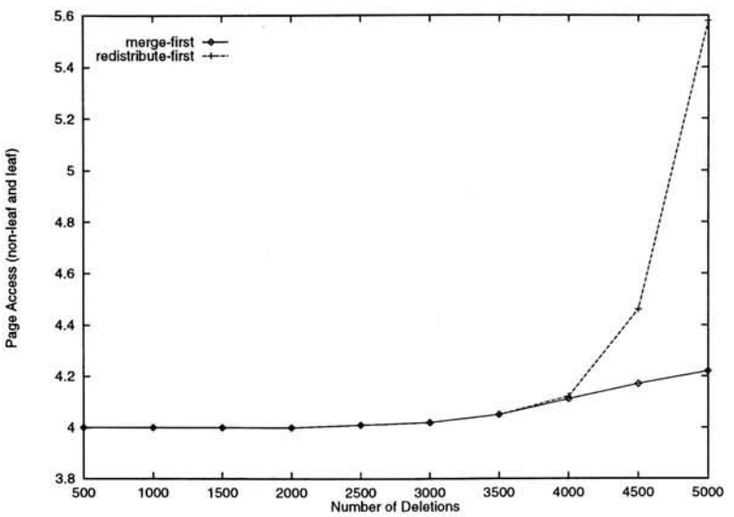


Figure 5.9: Page accesses vs. number of deletions on a synthetic uniform dataset of 10000 objects.

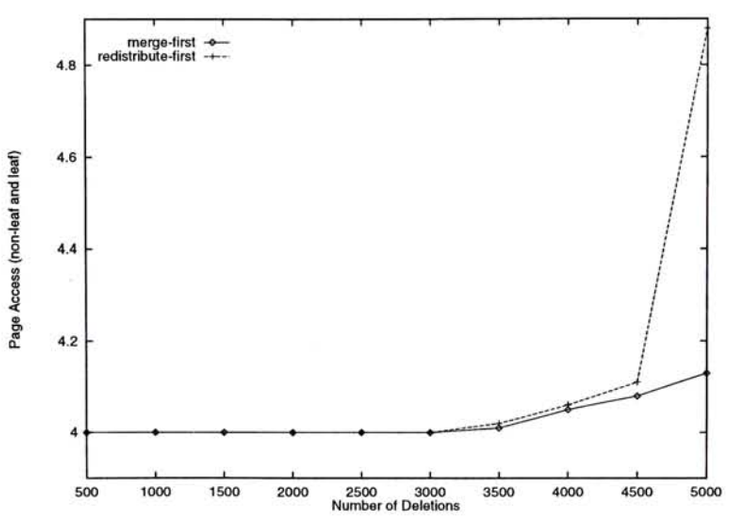


Figure 5.10: Page accesses vs. number of deletions on a real dataset of 10000 objects.

redistribute-first.

Note that we found neither redistributions nor merges at non-leaf nodes for the deletions we made. This indicates that it is rare for a subtree to underflow.

Chapter 6

Minimizing Distance Computations

In a high-dimensional space, the distance calculations between data objects are expected to be computationally expensive. As such, the major concern in most previous work on distance-based indexing is to minimize the number of distance computations in order to aim at efficient query processing. The mvp-tree [32] 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 6.1, each node of the mvp-tree can be viewed as two levels of a vp-tree, but involving fewer vantage points[1]. 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 in a number of experiments.

[1] In Figure 6.1, v1,v2,v3 denote the different vantage points used in the nodes.

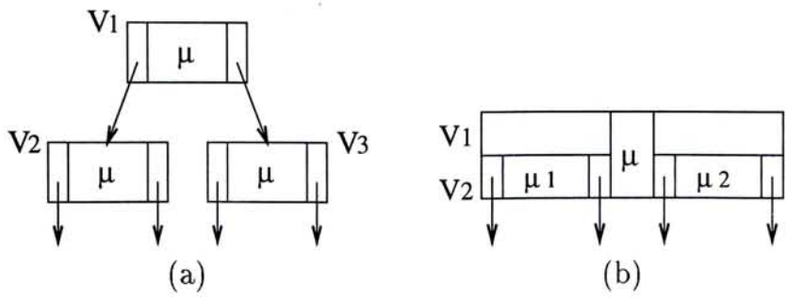


Figure 6.1: Node structures for (a) a binary vp-tree and (b) a binary mvp-tree.

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 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 Algorithm 2.1 (also the method used in the original vp-tree [36, 8]). 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 makes a higher fanout at the non-leaf nodes and a smaller tree size, and consequently enhances 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 Algorithm 2.1) 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 favourable partitioning of the dataset and a reduction of distance computations.

Unlike the previous approach, each node will have its own vantage point no matter if the nodes are at the same level or not. However, not every such vantage point is different. Some of them are in fact 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 2.1, 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 2.1. 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 significantly reduce the storage size of the tree and increase the fanout of the non-leaf nodes, in particular if the vantage points are 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 the value of p can be optimized for

certain datasets.

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 to it one of our n-nearest neighbor search algorithms, the single-pass method. 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 [32] 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 6.1 and 6.2. In these tables, the column labelled '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

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 6.1: Number of distance computations per search.

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 6.2: Page accesses per search.

the original vp-tree structure. Note that the results made by the original vp-tree are provided only for reference.

Table 6.1 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.

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 6.2 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.

Chapter 7

Conclusions and Future Work

We have tackled the problem of n-nearest neighbor search for multimedia data objects given only pair-wise distances between themselves. One method tries to first infer a feature vector for every data object from the distances provided, while preserving the distances between the objects. Then it employs some existing feature-based indexing method as the access mechanism. Since this method assumes that objects are points in some unknown high-dimensional space, and transforms them into a lower space, inaccuracies may occur in preserving the distances during transformation. This makes the index fail to locate all the required nearest neighbors in an n-nearest neighbor search. We have proposed to apply the vp-tree method to the problem. Being a distance-based index structure, the vp-tree solves the problem by partitioning the search space directly based on the distances between data objects, which are in fact the only input we have. Such an approach provides three main advantages. First, the pre-processing steps involved in inferring feature vectors can be eliminated. Second, the difficulty in preserving distances is avoided. More importantly, the correctness of query results can be guaranteed. Lastly, the method can be applied to domains where data are represented by multidimensional vectors as well. In other words, for indexing multidimensional data, distance-based methods make another choice in addition to feature-based indexing.

We have proposed three n-nearest neighbor search algorithms for the vp-tree. The value of a threshold σ which bounds the distance of the n-th nearest neighbor

from the query is critical to the problem of n-nearest neighbor search. Our first two algorithms, the *sigma_factor* algorithm and the constant-α algorithm, use their own method for estimating a σ value that must guarantee the presence of n nearest neighbors, then perform a range search with the estimated σ value. The single-pass algorithm, our third algorithm, does not need an estimation of σ. It starts with a σ which is infinitely large and dynamically optimizes the value whenever the algorithm encounters a candidate answer during the search.

We have shown by experiments that our algorithms scale up well with dataset size for synthetic clustered data and our real data. From our comparison, the single-pass algorithm performs the best and is a good choice since it requires neither a preset value for any parameter nor an extra pass of search to estimate σ. When compared to one popular feature-based index structure, the R^*-tree, the vp-tree with this algorithm consistently performs better for high-dimensional data in n-nearest neighbor search, with up to 73% savings in the access cost. The main reason is that the vp-tree scales up well with the dimensionality whereas the R^*-tree does not. This illustrates the competitiveness of distance-based methods in the indexing problem of multidimensional data.

In order to make the functionalities of an index structure complete, we believe that update algorithms are of the same importance as the search algorithms. We have proposed a solution to the update problem for the vp-tree, which was left open in previous work. We have observed that the split-first strategy of doing inserts is better than redistribute-first insertion because the former method delays saturation of nodes so that redistributions among multiple subtrees can be avoided. For delete operations, merge-first deletion requires less access cost compared to redistribute-first deletion, for the reason that the merge-first strategy helps reduce the possibility of underflows in subsequent deletions.

When the distance function is complex or feature vectors are in high dimensions, distance computations between data objects correspond to a critical factor to the performance of the vp-tree. We have investigated two methods for reducing the number

of distance computations. One is by associating only one vantage point with each non-leaf level, the other is by reusing a fixed number of vantage points. We have compared these methods with the approach proposed for the mvp-tree. For the comparisons, we have extended our n-nearest neighbor search algorithms to the mvp-tree. Experimental results show that reusing vantage points performs better than both the 'single vantage point per level' approach and the mvp-tree approach. Besides making the least distance computations between query point and vantage points, the approach of reusing vantage points also reduces the number of page accesses in n-nearest neighbor search.

7.1 Future Work

We have observed that a bottom-up construction of an index structure would lead to relatively easier procedures for doing updates. This is mainly because we can conveniently split or merge the index nodes. As for future work, we shall focus on developing a distance-based index structure that will grow in a bottom-up fashion but also have the advantage of using a distance function to organize data objects. The mvp-tree is so far as we know the only variant of the vp-tree, we shall consider other variations of the vp-tree. It is clear that the choice of vantage points is important to the performance of the vp-tree. The best vantage point should make the number of data points in the boundaries of partitions as low as possible so that the chance of exploring multiple branches can be minimized. It has been shown in our work on minimizing distance computations that various vantage point selection methods contribute differently to the access cost of searching. While the randomized selection algorithm proposed in previous work operates well in practice, we are interested in looking into other alternatives in detail. Regarding the desire to reduce multiple-branch traversal during search operations, creating redundancy in the vp-tree seems to be a promising approach. The idea is to duplicate the objects located on the boundary into both partitions of it. We shall further our work in this direction. On top of our current insert and delete algorithms for the vp-tree, we shall see if

there are heuristics of doing the updates, in particular if the tree need not be kept balanced all the time. We shall also extend our n-nearest neighbor algorithms and update algorithms to other distance-based index structures such as the Geometric Near-neighbor Access Tree (GNAT).

Bibliography

[1] W. G. Aref and Hanan Samet. Optimization strategies for spatial query processing. In *Proceedings of the 17th International Conference on VLDB*, pages 81–90, September 1991.

[2] Manish Arya, William Cody, Christos Faloutsos, Joel Richardson, and Arthur Toga. Qbism: a prototype 3-d medical image database system. *IEEE Data Engineering Bulletin*, 16(1):38–42, March 1993.

[3] N. Beckmann, Hans-Peter 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.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

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

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

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

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

[9] P. Ciaccia, F. Rabitti, and P. Zezula. Similarity search in multimedia database systems. In *Proceedings of the First International Conference on Visual Information Systems*, pages 107–115, February 1996.

[10] 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.

[11] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, September 1995.

[12] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[13] 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.

[14] 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.

[15] K. Hinrichs and J. Nievergelt. The grid file: a data structure to support proximity queries on spatial objects. In *Proceedings of International Workshop on Graph Theoretic Concepts in Computer Science*, pages 100–113, 1983.

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

[17] H. V. Jagadish. A retrieval technique for similar shapes. In *Proceedings of ACM SIGMOD*, pages 208–217, May 1991.

[18] R. Jain, S. N. Jayaram Murthy, and Peter L-J Chen. Similarity measures for image databases. In *FUZZ-IEEE '95*, 1995.

[19] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *Proceedings of the 20th International Conference on VLDB*, pages 500–509, September 1994.

[20] 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.

[21] Joseph B. Kruskal and Myron Wish. *Multidimensional scaling.* SAGE publication, Beverly Hills, 1978.

[22] King-ip Lin and C. Faloutsos. Fastmap: a fast algorithm for indexing, datamining and visualization of traditional and multimedia datasets. In *Proceedings of ACM SIGMOD*, 1995.

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

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

[25] 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.

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

[27] E. G. M. Petrakis and C. Faloutsos. Similarity searching in large image databases. Technical Report CS-TR-3388, University of Maryland, December 1994.

[28] John T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of ACM SIGMOD*, pages 10–18, 1981.

[29] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of ACM SIGMOD*, May 1985.

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

[31] 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.

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

[33] T. Wallace and P. Wintz. An efficient three-dimensional aircraft recognition algorithm using normalized fourier descriptors. *Computer Graphics and Image Processing*, 13:99–126, 1980.

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

[35] 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.

[36] 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.