University of Alberta

Metric Techniques for High-Dimensional Indexing

by

Christian Digout ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-95737-3
Our file Notre référence
ISBN: 0-612-95737-3

The miracle isn't that I finished ... The miracle is that I had the courage to start.
- John Bingham

Dedication

I would like to dedicate this thesis to my grandfather, Alfred Joseph Digout.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Mario Nascimento. Without Mario's advice, patience and guidance, this thesis may have never been started. His help and teaching in the writing of this thesis was invaluable. Mario has spent considerable time finding errors that I don't think I would have noticed if I had read through this thesis a hundred more times. Mario's sense of humor was also a big plus when sometimes the research for this thesis was not going according to plan.

Thanks to Alex Coman for the advice and input he provided at various times along the way and for coding contributions during preliminary experiments.

I would like to also thank my family and friends for all the support they have provided which has enabled me to write this thesis. I would especially like to thank Pam for all the help and support she provided during the hard times while writing this thesis.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Alberta Informatics Circle of Research Excellence (iCORE) and the Canadian Cultural Content Management Research Network, a Network financed through Heritage Canada's New Media Research Networks Fund.

Contents

List of Tables

List of Figures

List of Acronyms

Acl	Active Cluster List
CC-Array	Cluster Cell-Array
CDA	Cluster Distances Array
CPU	Central Processing Unit
GCH	Global Color Histogram
I/O	Disk operation (Input/Output)
ISA	Incremental Selection Algorithm
K-NN	K-Nearest Neighbors
MAM	Metric Access Method
MBR	Minimum Bounding Rectangle
MBS	Minimum Bounding Sphere
OSEQ	OMNI-Sequential
OSEQ$^+$	OMNI-Sequential$^+$
OSEQ*	OMNI-Sequential*
PCA	Principal Component Analysis
RQ	Range Query
SeqScan	Sequential Scan

Chapter 1

Introduction

Managing and querying high-dimensional data is important in many domains such as time sequence data [2], protein sequence data [22], density-based clustering algorithms [15] and multimedia data [24]. For instance, images have features such as color and texture, which are typically mapped onto high-dimensional vectors for facilitating similarity searches. It is common for an image to be represented by a D-dimensional global color histogram (GCH), e.g., [31, 32], where D is typically large. GCHs represent images by their color distribution, images are considered more similar if they have similar GCHs. In Figure 1.1, images (a) and (b) will have more similar GCHs than images (b) and (c). In Figure 1.1, we can see (using the L_1 metric) the distance between images (a) and (b) is 0.125 while the distance between images (b) and (c) is 0.750. When processing high-dimensional similarity queries, an efficient access method is needed because a full linear scan on the *raw* data set is typically not practical. On the other hand, techniques based on a linear scan of *preprocessed* data have been quite successful.

Most access methods require the distance function (or similarity function) to be a metric. Given a set of objects $s_i \in S$, a distance function $d(\ )$ is a metric if it obeys the following properties:

1. Symmetry: $d(s_a, s_b) = d(s_b, s_a)$
2. Non-negativity: $0 \leq d(s_a, s_b) < \infty$
3. Reflexivity: $d(a, a) = 0$
4. Triangular inequality: $d(s_a, s_b) \leq d(s_a, s_c) + d(s_b, s_c)$

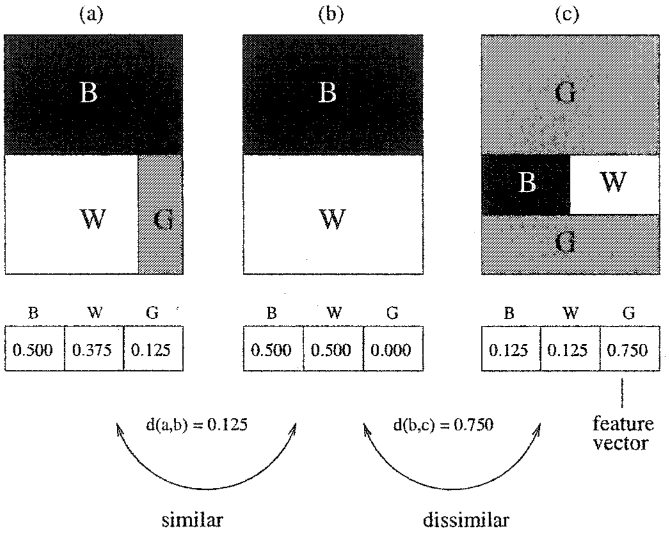


Figure 1.1: The feature vectors for the three images show the percentages of each color (B = black, W = white, and G = grey) the image is composed of, d(a,b) represents the distance between images a and b and is normalized between 0 and 1

A *Similarity Search* is defined as searching in a set of objects for the most similar objects to the query object based on some criteria or similarity function. The two most important types of queries for similarity search in high-dimensional data and metric data are range queries (RQ) and nearest neighbor queries (NN). Given a query object q and a database S, range queries are defined as $RQ(q,r) = \{s_i \in S | \forall s_i \in S : d(s_i,q) \leq r\}$ where r is the radius (range) of search and $d(i,j)$ is a metric distance function which measures the distance between database objects i and j. In Figure 1.2(a), the answer to range query q would be to return all objects which are less than distance r to q, i.e., all objects inside the ring around q. The nearest neighbor of q is defined as $NN(q) = \{s_{NN} \in S | \forall s_i \in S : d(s_{NN},q) \leq d(s_i,q)\}$. Nearest neighbor queries can also be generalized to retrieve the K-nearest objects to the query object. In Figure 1.2(b), $K = 3$, therefore the 3 nearest objects to q will be returned as the answer.

Indexing high-dimensional data is a hard problem due to the dimensionality curse. Often, as the dimensionality of the feature space increases, the ratio of

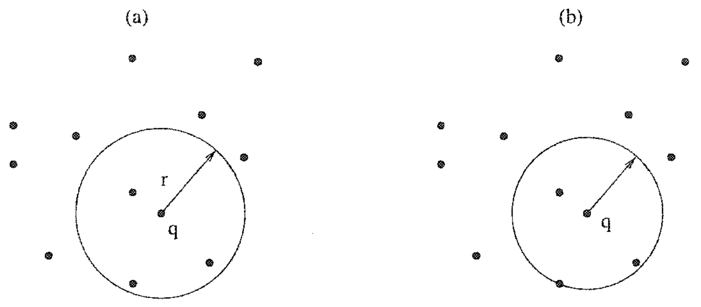


Figure 1.2: (a)Range Query (radius = r) (b)K-Nearest Neighbor Query (K=3)

the distance of the nearest neighbor and the distance of the farthest neighbor decreases [8]. This makes the problem difficult because many access methods attempt to discard objects based on their approximate distances to the query object while processing similarity queries. As the ratio of the distance of the nearest objects and the distance of the farthest objects decreases, access methods have a more difficult time differentiating between the objects close to the query object and those far away from the query object. In [34] the authors show that many access methods may degenerate to a sequential scan once the number of dimensions is sufficiently large. There are two main reasons why this occurs. One is that the cost of random disk I/Os is much greater than the cost of sequential disk I/Os. If an access method with a random access pattern does not access relatively few data pages, a sequential scan may be faster. It is also argued in [34] that a sequential reading of a set of pages on disk is faster than randomly reading as few as 20% of the same pages, which can likely be the case for high-dimensional data. In an interview with ACM Queue, Jim Gray says "Certainly we have to convert from random disk access to sequential access patterns. Disks will give you 200 accesses per second, so if you read a few kilobytes in each access, you're in the megabyte-per-second realm, and it will take a year to read a 20-terabyte disk. If you go to sequential access of larger chunks of the disk, you will get 500 times more bandwidth–you can read or write the disk in a day. So programmers have to start thinking of the disk as a sequential device rather than a random access device" [27]. This illustrates the importance of access methods using sequential access patterns as opposed to random access patterns. Yet another reason is that most access methods use a tree-like structure with hierarchical minimum bounding regions to index objects. Minimum bounding regions are defined as the smallest region which contains a set of objects. The overlap and volume of these bounding regions increase sharply with the increase in the dimensionality, causing many branches of the tree to be searched; as well, the fanout of the indexing tree decreases, yielding taller and thinner search trees.

The main contribution of this thesis are three new access methods aimed at providing a solution to the problem of efficient similarity search for high-dimensional data and metric data.

Motivation for our first two approaches comes from examining the OMNI-Sequential (OSEQ) algorithm which makes use of the OMNI Access method [16]. This method reduces the dimensionality of the feature space into a lower dimension, determined by a number of reference points (referred to as *pivots*). Pivot objects, which are selected from the data set, and OMNI-coordinates (precomputed distances from the data objects to the pivot objects) are used to increase the performance of similarity queries.

In this thesis we introduce two new access methods, the OMNI-Sequential$^+$ (OSEQ$^+$) and the OMNI-Sequential* (OSEQ*). Using the OSEQ$^+$, the time required to answer a similarity query can be further decreased by (1) identifying the best pivot object and by (2) changing the order the OMNI-coordinates are stored. The OSEQ* algorithm builds on the OSEQ$^+$. In this algorithm we propose to select more pivot objects than will be used at query time and at query time only use the best pivot objects to improve the performance of the OSEQ*. In Section 3.3 we will explain how to select these pivot objects. We show that for a vector space, Principal Component Analysis (PCA) selects better pivot objects than the pivot objects selected by the algorithms proposed in [12] and [16]. Our experimental evaluation using real and synthetic data sets shows that one can achieve up to three times faster query processing when compared to the original OSEQ at the cost of little storage overhead. We also show experimentally that the OSEQ$^+$ method is up to 24 times faster than the VA-File when processing range queries and up to 69 times faster than a sequential scan of the data set.

Motivation for our third access method comes from further examination of the VA-File [34]. The VA-File is arguably the most efficient access method for similarity search in high-dimensional data. It partitions the data space into a grid and creates approximations of the data objects based on the cells in the grid containing the objects. Its main advantage is it sequentially scans the file containing these approximations, which is much smaller than the size of the original data file.

The third contribution of this thesis is we show how to construct a grid[1] for any metric space (including of course Euclidean space) and perform efficient similarity searches. Such a grid is used to cluster the objects efficiently using any clustering algorithm, even if the data set is not vectorial. We will show how the clusters can be searched efficiently for similarity search queries and how the correctness of the answer set is guaranteed by the properties of the grid. Our experimental results show that we can perform query processing up to 42 times faster with our access method than a sequential scan of the original data set and up to 10 times faster than the VA-File. In addition, our method works in all metric spaces, can be easily implemented and scales well with increasing the number of nearest neighbors retrieved.

1.1 Thesis Outline

This thesis is structured as follows. Chapter 2 discusses the related work to indexing high-dimensional data and metric data. Chapter 3 details the OSEQ$^+$ and OSEQ* methods and the experimental results showing the efficiency of our proposed methods. The Metric Grid is described in Chapter 4, along with how the grid can be used to perform efficient similarity search and experiments are displayed comparing the M-Grid to the VA-File and a sequential scan of the data set. A conclusion and summary of this thesis will be given in Chapter 5.

[1] The grid is actually a pseudo-grid. It is not possible to construct a real grid in general metric space because there may be no actual data space with actual dimensions to be partitioned.

Chapter 2

Related Work

There are many proposed methods attempting to solve the problem of efficient similarity search for high-dimensional data sets and general metric data sets. Most methods suffer from the dimensionality curse, the one notable exception is the VA-File, or cannot index metric data. In this chapter we present an overview of access methods which can be used for high-dimensional data sets and/or metric data sets.

The proposed indexing structures described in this chapter either use a tree-based structure, a linear scan-based structure or a combination of the two. Tree-based structures try to group objects which reside closest to each other in the data space. These groups of objects are typically stored as minimum bounding regions which are defined as the smallest region which contains the set of objects. Minimum bounding regions are stored in a hierarchical manner for similarity search. The tree structure (Figure 2.1) is searched by visiting branches in the tree which intersect the query region. For range queries, the volume of the query region remains constant with radius r, while for nearest neighbor queries, the volume will decrease as nearer objects to the query object are found, reducing the radius of the query region. Trees exhibit random access patterns, therefore if a tree-based access structure cannot access relatively few data pages, a sequential scan of the same set of pages may be faster. This is often the case for high-dimensional data sets and metric data sets.

Linear Scan-based approaches preprocess and store the data in sequential files (Figure 2.2). At query time portions of the preprocessed data are read

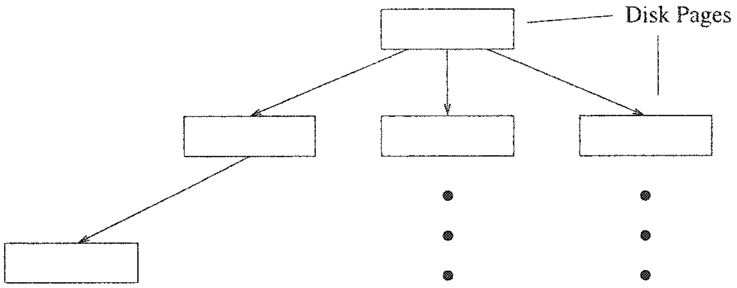


Figure 2.1: Organizing data in a tree structure for similarity search

sequentially. Although this can mean accessing more disk pages, sequential disk accesses are much less expensive than random disk accesses, therefore the efficiency of similarity search can be increased even if more disk pages are accessed. Two possible linear scan-based approaches to tackle the problem of high-dimensional and/or metric similarity searches are: to reduce the dimensionality of the feature space or to reduce the granularity of the feature space. Reducing the dimensionality of the feature space is achieved by mapping objects from the original feature space into a lower dimensional space. The coordinates of each object in the lower dimensional space can be read sequentially. The mapping into a lower dimensional space preserves the relative distances of the objects so similarity search can still be preformed while still guaranteeing the correctness of the answer. Reducing the granularity of the feature space requires approximating the position of objects in the data space. This approach sequentially reads the approximations of the objects (which requires less storage space than the original feature vectors) and performs a refinement step to guarantee the correctness of the answer.

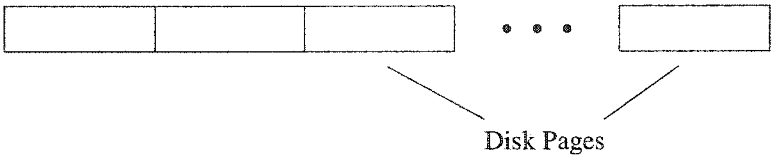


Figure 2.2: Storing preprocessed data for similarity search in a sequential file

2.1 Indexing Structures for Vector Data

Vector data has the property that all attributes of an object are numerical. This gives positional information about the data objects in a vector space that can be used for indexing. There are many indexing structures proposed which can only index vector data because they require information about the position of each object in the data space in order to create approximations and minimum bounding regions of groups of objects. Two partitioning strategies for vector data are data partitioning and space partitioning. The indexing structures partition the data space in order to obtain approximations of where objects lie in the data space. This allows sub-regions to be accessed in the data space and to discard objects or groups of objects from the answer set without computing the actual distance between the query object and every object in the data set.

2.1.1 Data Partitioning Structures

Indexing structures using the data partitioning strategy divide the objects in a hierarchical manner depending on the distribution of the data set. The objects are partitioned into cells or minimum bounding regions which are stored in nodes of equal size. Indexing structures which use the data partitioning strategy are the R-tree, the R*-tree, the TV-tree, the SS-tree, the X-tree and the SR-tree.

The R-tree [17] is a multi-dimensional indexing structure. It was originally designed for 2D spatial data (though it can be easily generalized to multi-dimensional space) and can handle rectangles and point data. The data in a R-tree is partitioned into minimum bounding rectangles (MBRs) in a hierarchical manner and attempts to minimize the area of the MBRs. As it will be clear shortly, it is important to minimize the area of the MBRs because the larger their area, the greater the probability an MBR will intersect the query region, leading to accessing more nodes during similarity search.

The R-tree is height-balanced tree, meaning each leaf node is the same distance from the root node. Each leaf node has the form $(ObjID, R_t)$ where

$ObjID$ is a pointer to the actual object in the database and R_t is the MBR that contains the object. Non-leaf nodes are composed of $(ChdP, R_t)$ where $ChdP$ is a pointer to a child node and R_t is the MBR that contains all the objects in its sub-tree. An example R-tree is shown in Figure 2.3(a) and refers to the points in Figure 2.3(b). Rectangle R1 completely covers rectangles A, B and C which are situated in R1's sub-tree.

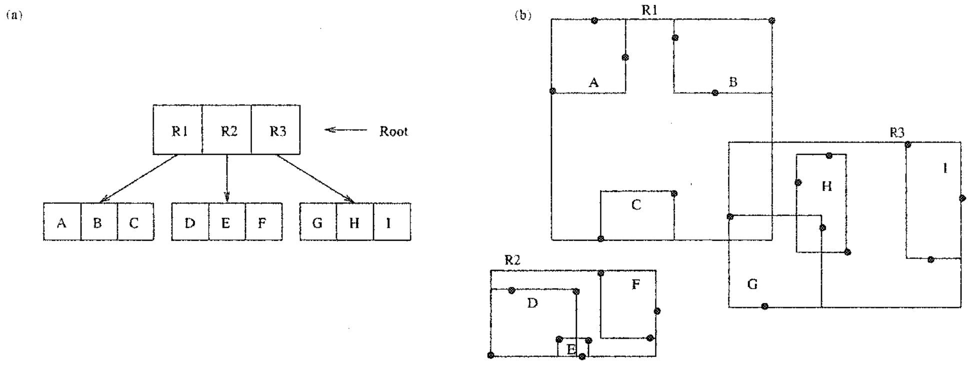


Figure 2.3: The R-tree Structure

The R-tree is a dynamic indexing structure, objects can be inserted and deleted from the tree without completely rebuilding the tree. Each node in the R-tree corresponds to one disk page and must contain between m and M objects except the root node. The root node must be able to have less than m children because when beginning to build the R-tree, initially there are no objects and the only node is the root node. The nodes in the R-tree are not usually completely full as it is unlikely for all nodes to have M objects because objects are inserted into the node in the tree for which the volume of the MBR increases the least, not the node which has the least objects. The reason for creating a lower bound m and an upper bound M on the number of objects in a node is prevent storage utilization from being too low but still maintain the tree structure. The value of M is dependent on how many objects can fit on one disk page because it is assumed the complete R-tree is too large to fit in main memory and must be stored in secondary memory. The value of m is less than or equal to $\frac{M}{2}$ because if a node splits during the insertion of an object and $m > \frac{M}{2}$, the objects would not be able to be split into two new nodes and have at least m objects in each of the two new nodes. Nodes can overflow or underflow as objects are inserted or deleted from the R-tree. If a node overflows, i.e., the number of objects in the node becomes greater then M, the node is split and the split propagates towards the root node maintaining the height of the tree. For deleting objects, if a node underflows, i.e., the number of objects in the node becomes less than m, all the objects in the node are reinserted into the tree.

The search algorithm for the R-tree begins at the root node and descends toward the leaf level. At each node the distance between the query object and the MBR is computed as the minimum distance to any point in the MBR (Figure 2.4). If the MBR of the node intersects the query region, its sub-tree is also searched. The query region can intersect more than one MBR so many sub-trees and branches may have to be searched. In Figure 2.3(a) and (b), the distance from MBRs R1, R2 and R3 will be computed to the query object. If the MBRs intersect the query region, their sub-trees will also have to searched. Sub-trees of MBRs not intersecting the query region do not have to be searched because no objects in the sub-tree can be in the answer set to the query. The reason for this is the MBR contains all objects in its sub-tree, therefore if the MBR does not intersect the query region, no objects inside the MBR can either. The search algorithm ends when all leaf nodes which intersect the query region have been accessed.

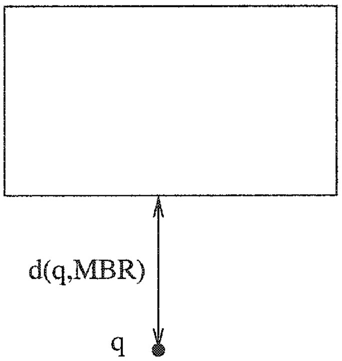


Figure 2.4: Computing the distance from the query object to MBRs in the R-tree

For NN queries, the search algorithm for the R-tree (Figure 2.5) begins by

```
R-tree NN Search Algorithm: Find the nearest neighbor to the query object
Input  : Query Object $q$, Active Node List $Anl$ = [root], Nodes in the
         R-tree $n_i \in N$, Objects in the data set $s_i \in S$
Output: Nearest Neighbor (NN) to $q$
NN = Null
NNdist = $\infty$
while Anl Not Empty do
    $n_i$ = first node in $Anl$
    if $n_i$ is leaf node then
        foreach $s_i \in n_i$ do
            if $d(q, s_i) < NNdist$ then
                NN = $s_i$
                NNdist = $d(q, s_i)$
                Prune $Anl$ with NNdist
    else
        /*$n_i$ is an internal node*/
        foreach Child Node $c_i$ in $n_i$ do
            if $d(q, c_i) < NNdist$ then
                Insert $c_i$ into $Anl$
```

Figure 2.5: R-tree Nearest Neighbor Query Search Algorithm

computing the distance from the query object to each MBR in the root node. The distances are stored in a sorted list called the Active Node List (*Anl*). The MBR closest to the query object is searched first and this process continues recursively down to the leaf level. If a leaf node is accessed, the actual distance from the query object to each object in the leaf node is computed and the nearest neighbor distance (NNdist) is updated if a nearer object to the query object is found. Nodes in the *Anl* are pruned and removed if their minimum distance to the query object is not less than the current NNdist. Each Node in the *Anl* must be searched to guarantee the answer is correct, the algorithm stops when the *Anl* is empty, indicating there are no more MBRs closer than the NNdist to the query object.

For range queries, the search algorithm (Figure 2.6) is slightly different

```
R-tree Range Query Search Algorithm: Find all the objects within
distance $r$ of the query object
Input  : Query Object $q$, Query Radius $r$, Active Node List $Anl$ = [root],
         Nodes in the R-tree $n_i \in N$, Objects in the data set $s_i \in S$
Output: List of objects whose distance to $q$ is less than $r$, $rList$

while $Anl$ Not Empty do
    $n_i$ = first node in $Anl$
    if $n_i$ is leaf node then
        foreach $s_i \in n_i$ do
            if $d(q, s_i) \leq r$ then
                Insert $s_i$ into $rList$
    else
        /*$n_i$ is an internal node*/
        foreach Child Node $c_i$ in $n_i$ do
            if $d(q, c_i) \leq r$ then
                Insert $c_i$ into $Anl$
```

Figure 2.6: R-tree Range Query Search Algorithm

because the radius r of the query is known in advance unlike the actual NN distance. The search algorithm computes the distance to each MBR in the root node from the query object. Each MBR whose distance to the query object is less than or equal to r is placed in the *Anl* and must be searched. This continues until all nodes in the R-tree have been searched or pruned.

The R-tree is efficient for a small number of dimensions but is shown to perform poorly for high-dimensional data (dimensionality greater than 10) [7]. This occurs because in high-dimensional space, the volume and overlap of the MBRs increase sharply causing many branches in the tree to be searched for each query, as well, the fanout of the tree decreases. The importance of the R-tree is it laid a foundation for others to build upon. As will be seen shortly, many proposed indexing structures are based on the R-tree.

An indexing structure very similar to the R-tree is the R*-tree [5]. The R*-tree improves on the R-tree by not only minimizing the area of the MBRs, but also minimizing the overlap of the MBRs and increasing the storage utilization of the MBRs. Minimizing the overlap is important because if two MBRs exhibit high overlap, there is a high probability they will have to be both accessed for the same queries. Increasing the storage utilization will mean the disk pages are more full and the height of the tree will be less. This will reduce the number of nodes needing to be accessed to reach the leaf level of the tree and to answer similarity queries. The R*-tree employs the notion of *forced reinsert* which reinserts all entries of a node that overflows during the insertion of an object. This has been shown to increase the efficiency of the R*-tree because it provides an opportunity for dissimilar objects originally placed in the same node to be placed in different nodes as more similar objects are inserted into the tree. The R*-tree is shown to outperform the R-tree, but like the R-tree still suffers from the dimensionality curse.

The TV-tree (Telescopic-Vector tree) [23] is one of the first indexing structures proposed specifically for high-dimensional data. The TV-tree proposes to use only a few features to distinguish between objects in the database at each level of the tree. By using a few features, the tree suffers less from the dimensionality curse.

The TV-tree uses a hierarchical structure (Figure 2.7(a)) similar to the R-tree. Feature vectors are stored in the leaf nodes and the parent (internal) nodes contain the TMBR (telescopic minimum bounding region) of its children, this continues recursively until the root node. The authors note that TMBRs can be stored as any shape, but in the paper spheres are used as they can be represented more simply by just storing the centroid and the radius of the sphere. The TMBR of a node contracts or extends depending on the objects inserted into or deleted from the region. The TV-tree uses less dimensions close to the root and more dimensions as we descend down the tree to distinguish between objects. Techniques such as Karhunen Loeve transform, Discrete Cosine transfrom and Discrete Fourier transform can be used to find the dimensions which exhibit the most discriminative power. These dimensions are used at higher levels of the TV-tree when searching because the distance between objects can be distinguished more accurately using these dimensions than the dimensions with less discriminative power. This can prevent searching to the leaf level for many branches of the tree and allows the

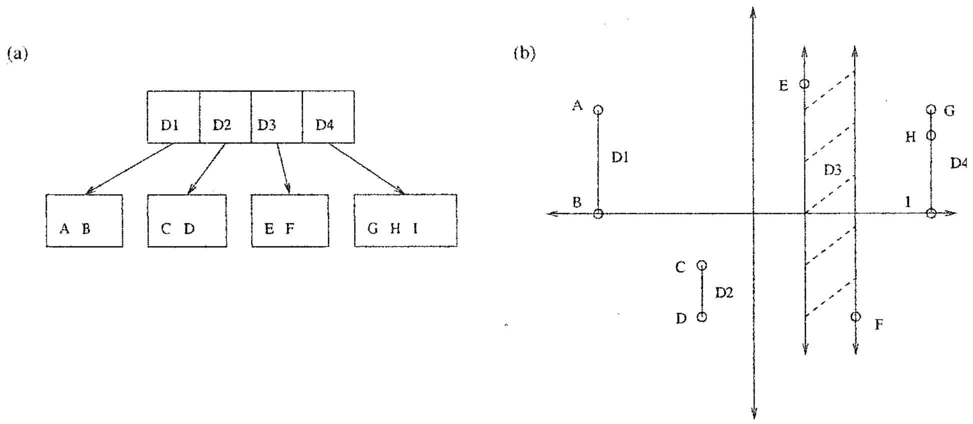


Figure 2.7: The TV-tree Structure

TV-tree to have a higher fanout in the higher levels of the tree because only a few of the most important dimensions are stored.

As the tree grows, the leaf levels may consist of some objects which agree on their first k dimensions, these dimensions are considered inactive. The dimensions the objects differ on are the active dimensions and are used to distinguish between objects in the same nodes. In Figure 2.7(b) only one active dimension is used. The active dimension for objects A and B is D1, while for objects E and F, the active dimension is D3. This means the TV-tree will only be able to distinguish between objects A and B using dimension D1. The TV-tree can however use more than one active dimension. TMBRs in the tree can overlap each other and like MBRs in the R-tree, it is more efficient to minimize the overlap of the TMBRs to minimize the number of branches to be accessed during similarity search.

Searching of the TV-tree is similar to the R-tree and always starts at the root node and follows the branches which intersect the query region. For NN queries, the most promising branches are searched first, i.e., the branches closest to the query object. This allows nearer objects to the query object to be found first and branches far away can be pruned. Objects are inserted into the most suitable branches of the TV-tree using the following criteria; (1) minimizing the overlap of the TMBRs, (2) the new object agrees on as many coordinates as possible in the sub-tree, (3) minimize the increase in the radius of the TMBR and (4) minimize the distance to the center of the TMBR. If the chosen node is full, leaf nodes are reinserted once and then split if the node overflows again, while internal nodes are always split. Deletion is straight forward, if the node underflows, the node is deleted and all the objects in the node are reinserted into the TV-tree.

The TV-tree has been shown to be more efficient than the R*-tree, but relies on two assumptions. There must be an "order of importance" of the dimensions of the data space. If all dimensions are equally important, the TV-tree will not be able to discriminate effectively between objects in the tree at high levels. The second assumption is that sets of feature vectors will tend to exactly match on some dimensions, especially on the important dimensions. This assumption often does not hold for real-valued data sets. The reason is that objects from real data sets, while they are often clustered, they will rarely have an exact match on many dimensions unless a discrete value distance function is used. If these two assumptions fail to hold for the TV-tree, its efficiency will sharply decline.

The SS-tree (Similarity Search tree) [35] is a R-tree variant designed for similarity search in high-dimensional spaces. The main design of the SS-tree is very similar to the R*-tree except minimum bounding spheres (MBS) are used instead of minimum bounding rectangles to index the data space. Like the R-tree and the R*-tree, the SS-tree is a dynamic indexing structure for vector data.

The structure of the SS-tree is shown in Figure 2.8(a). The objects are grouped together by spheres in a hierarchical manner where the parent nodes sphere completely bounds all the spheres of the nodes beneath it in the tree (Figure 2.8(b)). Each internal node keeps track of the centroid and the radius of the sphere and the number of children in the subtree. This allows the SS-tree to have a larger fanout than the R*-tree because spheres can be stored in half the space of rectangles. Minimum bounding rectangles have to store two coordinates for each dimension while minimum bounding spheres only need to store the centroid which requires one coordinate for each dimension plus the radius of the sphere. The radius of the sphere is the distance to the furthest

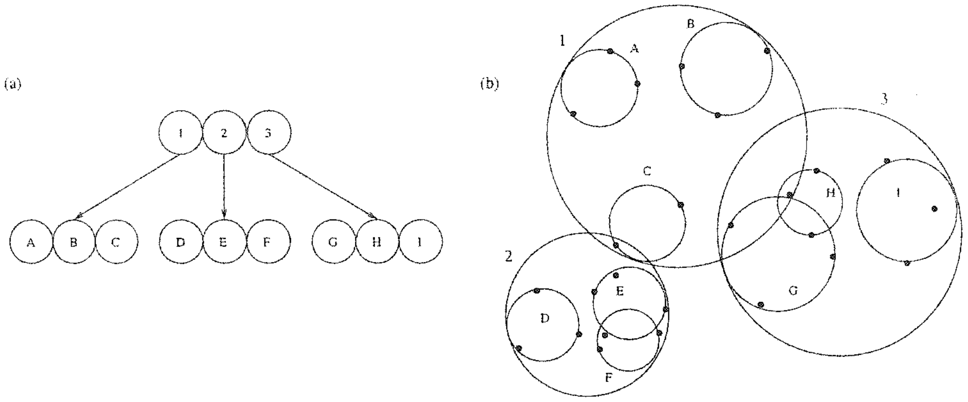


Figure 2.8: The SS-tree Structure

object from the centroid.

The algorithm to search the SS-tree is also similar to the R*-tree. The closest nodes to the query object are visited first for nearest neighbor queries. The distance between each MBS and the query object is computed as the minimum distance from the query object to the MBS. The algorithm stops when all nodes (MBSs) have been either visited or pruned. The SS-tree also uses the same insertion algorithm as the R*-tree. The concept of *forced insert* is used and each node has a minimum of m and maximum of M children. To insert an object into the SS-tree, the tree is traversed until the object's leaf node is found and all nodes along the way are updated by adjusting the radius of the node. The rule is to insert the objects into the subtree whose centroid is closest to the object. If the node overflows, the objects are reinserted into the tree. If the objects in the node have already been reinserted and the node overflows again, the parent node is split. The split algorithm searches for the dimension with the highest variance and then chooses a split location, this minimizes the variance in each of the new nodes.

The SS-tree is compared to the R*-tree and is shown to be more efficient. One reason for this is the SS-tree has a larger fanout than the R*-tree. This allows more MBSs to be stored in each node, decreasing the number of nodes which have to be accessed from disk during similarity search.

The X-tree (eXtended node tree) [7] is also an adaptation of the R*-tree and is intended for high-dimensional data. The X-tree attempts to deal with the shortfall of the R*-tree that the overlap of directories becomes large as the dimensionality of the data space increases. This causes many directories in the tree to be searched for the same queries because if two MBRs have a high degree of overlap, often both MBRs will intersect the query region for the same queries.

The solution the X-tree provides is to avoid splitting nodes in which the two nodes will exhibit a high degree of overlap. Instead of splitting a node and introducing two nodes with high overlap, the node is extended to create a supernode. In Figure 2.9 the shaded nodes indicate supernodes. Supernodes allow portions of the data to be organized sequentially on disk. If both nodes will have to be searched anyway, a sequential scan of the supernode is performed to avoid using a random disk access to access each of the nodes that would have resulted if the node had been split. The size of supernodes are multiples of the size of ordinary nodes. Results have shown the number of supernodes in the X-tree increases with the dimensionality. The storage utilization of the X-tree also increases because there are less nodes in the tree, reducing the space which is not used.

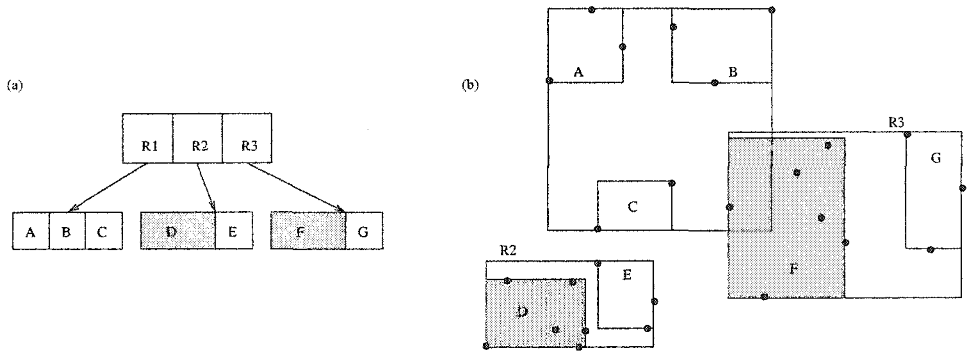


Figure 2.9: The X-tree Structure, the shaded regions are supernodes

The insertion algorithm for the X-tree is also similar to the R*-tree. The algorithm determines the node to insert the object in the same way as the R*-tree and inserts the object in this node. If the node splits, the split algorithm attempts to find a good split. If the split algorithm can not find a good split in which each of the new nodes will contain at least m MBRs, the split algorithm terminates without splitting the node and the current node is extended to a supernode. The deletion of objects is similar to the R*-tree.

The X-tree has two special cases. For data sets which exhibit no overlap, no supernodes will be created and the X-tree will be very similar to the R*-tree. For high-dimensional data with high overlap, the complete directory could become one large supernode and the X-tree would be equivalent to a sequential scan of the data set. Most data sets will create an X-tree similar to Figure 2.9 with a portion of nodes being extended to supernodes. The X-tree is shown to be up to two orders of magnitude more efficient than the R*-tree tree for high-dimensional data sets. The X-tree, like the R*-tree, still suffers from the dimensionality curse, but delays it to higher dimensions than the R*-tree.

The SR-tree (Square/Rectangle tree) [20] is an R-tree variant designed to overcome the deficiencies of the R*-tree and the SS-tree for high-dimensional data. The main difference in the structure of the SR-tree from the R*-tree and the SS-tree is the SR-tree uses minimum bounding spheres (MBSs) and minimum bounding rectangles (MBRs). An example SR-tree is shown in Figure 2.10(a) and refers to the points in Figure 2.10(b). Minimum bounding regions in the SR-tree are defined by the intersection of MBRs and MBSs. This is to decrease the volume of the minimum bounding regions and improve the disjointness between the regions. Figure 2.11(a) and (b) illustrate this concept as the shaded regions show the intersection of the MBRs and MBSs. By using both MBSs and MBRs, nodes E and F do not exhibit any overlap (Figure 2.11(a)) and the minimum bounding regions of all nodes are reduced.

The R*-tree and SS-tree have been shown to perform poorly in high-dimensional spaces because the directories in the trees tend to exhibit high overlap causing many directories to be searched for each query. The intuition behind the SR-tree is the diameter of hyperspheres tend to be much shorter than the diagonal of hyperrectangles in high-dimensional space, but the volumes of hyperspheres are much larger. By using both spheres and rectangles, the SR-tree inherits the benefits of both the minimum bounding spheres of the SS-tree and the minimum bounding rectangles of the R*-tree. The SR-tree can prune sub-trees more effectively using MBSs and MBRs and prevent having

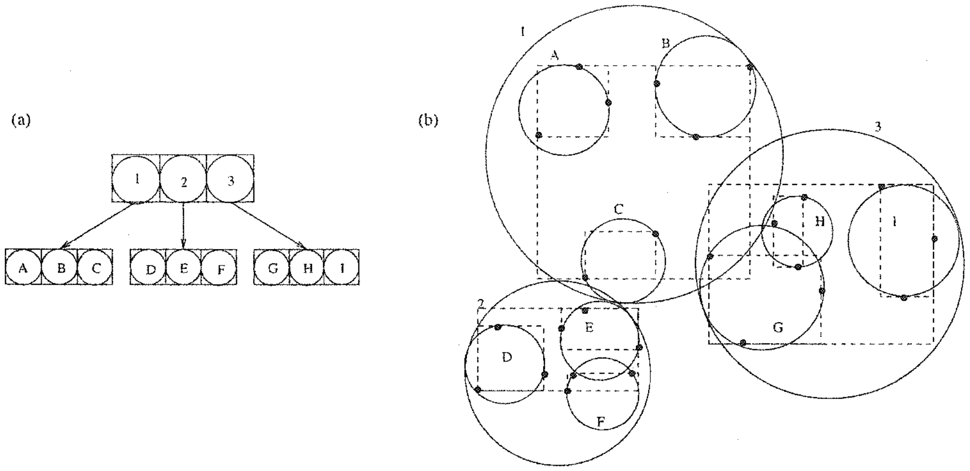


Figure 2.10: The SR-tree Structure

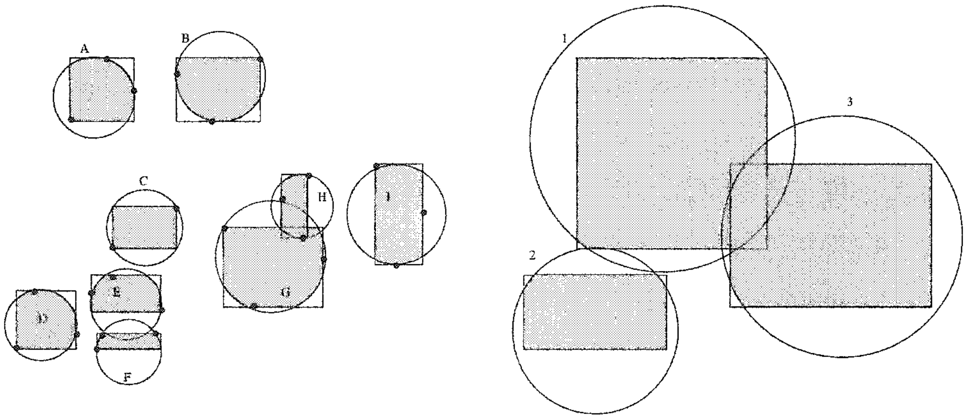


Figure 2.11: The intersection of the MBSs and MBRs in the SR-tree (shaded regions)

to access a large portion of the leaf level of the tree.

The insertion algorithm for the SR-tree is based on the SS-tree and chooses the subtree whose centroid is closest to the object being inserted. The authors choose to insert objects into the SR-tree based on the centroids of the MBSs rather than use the MBRs because experimental results show this method to be more effective. When a leaf node overflows, a portion of the entries are reinserted into the tree. The node is split if the entries are reinserted into the same nodes. The deletion of objects is performed the same way as the R-tree, if a node underflows, the entries in the node are reinserted into the tree. For the deletion and insertion of objects in the SR-tree, both the minimum bounding spheres and minimum bounding rectangles have to be updated.

The nearest neighbor search algorithm used for the SR-tree is very similar to the R*-tree and the SS-tree. One difference is the R*-tree computes the distance between the query object q and the MBR as the minimum distance to the MBR (Figure 2.12(a)) and the SS-tree computes the distance between q and the MBS as the minimum distance to the MBS (Figure 2.12(b)). Since the SR-tree stores MBRs and MBSs in each node, the distance from the query object to the bounding region R is computed as the longer of the minimum distance to the MBR and the MBS. This allows the distances to regions to be estimated more accurately. In the example in Figure 2.12(c), the distance to the minimum bounding region for the SR-tree ($d(q, R)$) will be equal to $d(q, MBR)$ and not $d(q, MBS)$ because is this example $d(q, MBR) > d(q, MBS)$.

Nodes in the SR-tree contain a smaller number of entries than the SS-tree and R*-tree because it stores a MBR and a MBS for each minimum bounding region in each node. This causes the SR-tree to have a smaller fanout. The SR-tree on the other hand prunes branches more efficiently than the R*-tree and the SS-tree because the minimum bounding regions of the SR-tree have a smaller volume. This occurs because the minimum bounding regions of the SR-tree are formed by the intersection of the MBRs and the MBSs (shaded regions in Figure 2.11). The SR-tree requires far fewer disk accesses at the leaf level because more branches of the tree are pruned at higher levels of the SR-tree because the minimum bounding regions have a smaller hyper-volume.

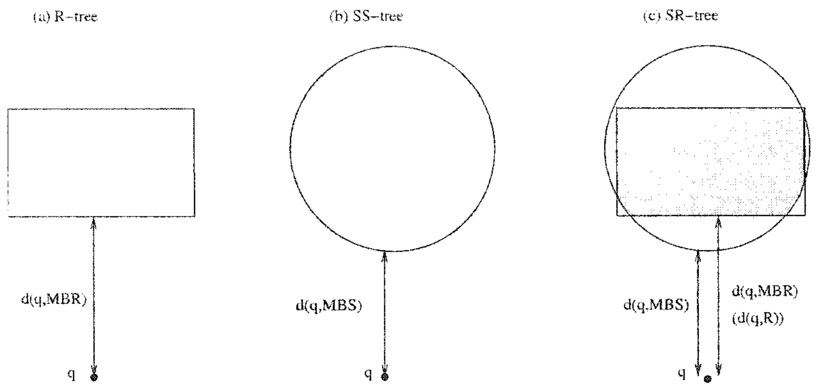


Figure 2.12: How the distance is computed for the minimum bounding regions in (a) the R-tree, (b) the SS-tree and (c) the SR-tree

The SR-tree has been shown to outperform the R*-tree and the SS-tree but still suffers from the dimensionality curse.

2.1.2 Space Partitioning Structures

Unlike the data partitioning strategy, space partitioning strategies partition the data space into cells or a grid. The objects are indexed depending on the cell in the grid that contains the object. Here we will present two access methods which use the space partitioning strategy, the VA-File and Clindex.

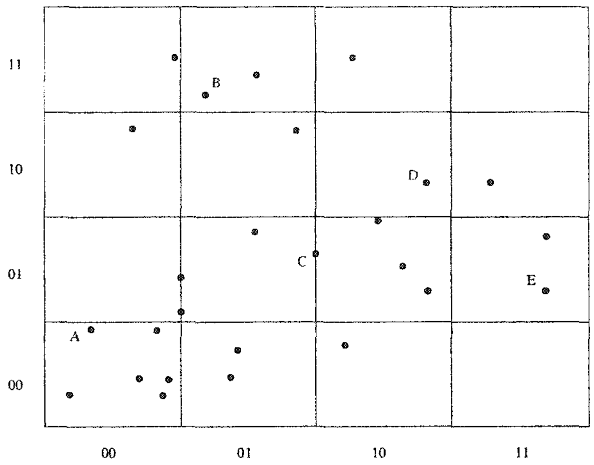
The VA-File (Vector Approximation File) [34] is proposed for similarity search in high-dimensional spaces. The VA-File method does not use a tree structure but instead stores an approximation of the feature vector of each object in a sequential file and performs a sequential scan of the vector approximations. The vector approximations are compressed representations of the original feature vectors and are typically 25% of the size of the original feature vectors.

To obtain the vector approximations, the data space is divided 2^l times along each dimension where l is the number of bits used per dimension. This divides the data space into $2^{l \times D}$ hyperrectangular cells where D is the dimensionality of the data space and each object is approximated by the bit string of the cell where it lies. In the example in Figure 2.13, two bits are used per dimension resulting in 16 ($2^{2\times2}$) hyperrectangular cells ($l = 2, D = 2$). In the example, object B can be approximated by $(01, 11)$ (4 bits of storage) instead of the original values of $(0.299, 0.783)$, which will require two floats to store the values. The bit string is used to compute the approximation of each object because the partitions in each dimension are stored.

For each query the VA-File is read sequentially and a lower bound and upper bound on the actual distance between the approximate feature vector of each object and the query object is computed. This is illustrated in Figure 2.14. The lower bound distance is computed from the query object to the nearest point in the object's cell and the upper bound is computed as the distance from the query object to the farthest point in the object's cell. Objects with lower bounds greater than the query radius (for range queries) or the current kth smallest upper bound (for K-nearest neighbor queries) are pruned from the candidate set because their actual distance to the query object must be too large for the object to be in the answer set. This allows many objects to be pruned from the candidate answer set without having to retrieve the object's full feature vector. To guarantee the correctness of the answer, a refinement step is performed where the actual distance between objects not pruned and the query object is computed. Typically, if l is chosen to be large enough, this method only requires retrieving a small number of actual feature vectors in the refinement step.

One problem with the VA-File is l has to be chosen before the VA-File is constructed. If l is chosen poorly, it can severely limit the performance of the VA-File. If l is too small, a large number of feature vectors will have to be retrieved from the original data file. This is expensive because each feature vector must be accessed randomly because there is no order in the original data file. If l is chosen to be too large, there will be very little savings over a sequential scan of the original data set because the entire VA-File is read for every query. The VA-File must also compute a large number of high-dimensional distance calculations. The number of distance calculations is linear with the cardinality of the data set.

The VA-File has been shown to exhibit linear performance with respect



Vector Data

A	0.126 0.243
B	0.299 0.783
C	0.500 0.412
D	0.684 0.601
E	0.878 0.328

Approximation Data

A	00 00
B	01 11
C	01 01
D	10 10
E	11 01

Figure 2.13: Partitioning the data space with the VA-File to obtain approximations for each object

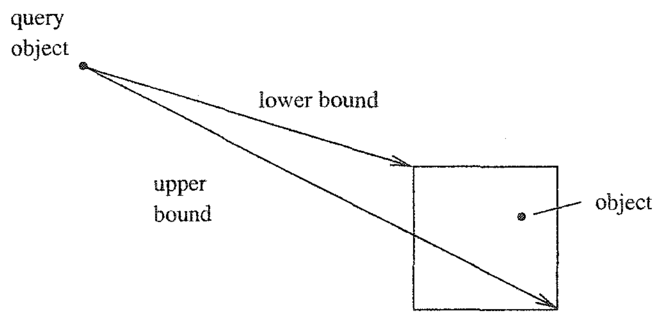
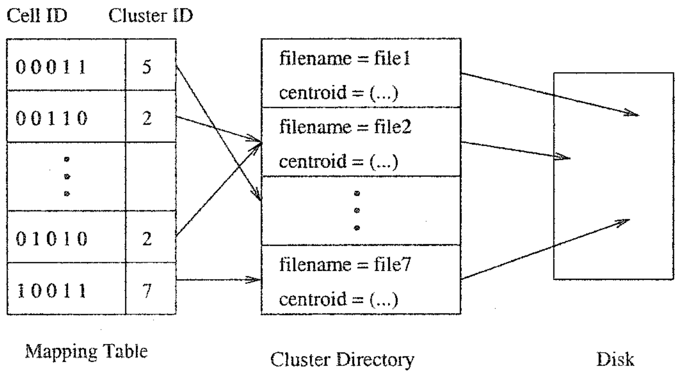


Figure 2.14: Calculating lower and upper bounds for the VA-File

to the cardinality and dimensionality of the data set. It should be noted the VA-File can only index vector spaces because a grid of the data space must be constructed. In high-dimensional spaces, the VA-File has been shown to outperform the R*-tree and the X-tree.

An Access method which partitions the data space into a grid and clusters the data set is Clindex [21]. Clindex answers approximate similarity search queries, i.e., it does not guarantee the correct answer. By answering approximate similarity queries, Clindex trades accuracy for efficiency. Each dimension is divided into strips which form cells and the cells are grouped into clusters using a simplified version of the CLIQUE clustering algorithm [3]. The clusters are stored sequentially on disk and an index is built to provide fast access to the clusters. This enables the algorithm to perform approximate similarity search by visiting a few clusters and reading all the objects sequentially within the visited clusters.

For similarity search, the cell the query object belongs to is first identified by mapping the query object's feature vector to the ID of the cell that contains it. The cell containing the query object is looked up in the mapping table (Figure 2.15). Only cells containing objects are stored in the mapping table, if the cell contains any objects, then the cluster it belongs to is accessed. If the cell is empty, then the distance from the query object to the centroid of each cluster is computed to determine the closest clusters.



Cell ID	Cluster ID
00011	5
00110	2
⋮	
01010	2
10011	7

filename = file1
centroid = (...)

filename = file2
centroid = (...)

⋮

filename = file7
centroid = (...)

Figure 2.15: Clindex

Clindex provides approximate similarity search so the answer set is not exact. The authors say approximately 90% of the exact answer set is returned on average after retrieving only a few of the closest clusters to the query object. The effectiveness of this method will also depend on how good the clustering algorithm is and the size of the clusters. The authors only test Clindex on vector data, and it is shown to be more efficient than other methods, but because it does not return the exact answer, its effectiveness is not guaranteed.

2.1.3 Hybrid Partitioning Structures

In an attempt to increase efficiency, some access methods have combined the data and space partitioning strategies. We will describe two indexing structures which use both partitioning strategies, the A-tree and the IQ-tree.

The A-tree (Approximation Tree) [29] is designed from studying the SR-tree and the VA-File. The authors of the A-tree attempt to use the best attributes of both structures to create an index structure efficient for similarity search in high-dimensional spaces. The A-tree partitions the data into minimum bounding regions, but also partitions the data space inside MBRs to create approximations of the child MBRs. Like the VA-File and SR-tree, the A-tree is only suitable for indexing vector space.

The A-tree uses a tree structure similar to the R-tree family of data structures. The main distinction of the A-tree is it uses Virtual Bounding Regions (VBR) to approximate MBRs which can be stored more compactly to increase the fanout of the tree. In Figure 2.16, Rectangle B (R_B) is approximated by V_B which can be stored using less space, i.e., using less bytes than the absolute coordinates of R_B because the coordinates of V_B are stored as binary sub-codes and the actual coordinates of R_B are stored as floats. The VBRs in the A-tree are calculated using the sub-codes and the coordinates of the parent MBR. This is possible because each node in the A-tree contains the MBR of the node and the VBRs of each childnode. The data space of the parent MBR is partitioned allowing the coordinates of the VBRs to be computed from the sub-codes. In this example, each sub-code of V_B can be stored using 4 bits, while each coordinate of R_B will be stored using 1 float (32 bits). Unlike the VA-File which uses absolute vector approximations and uses the

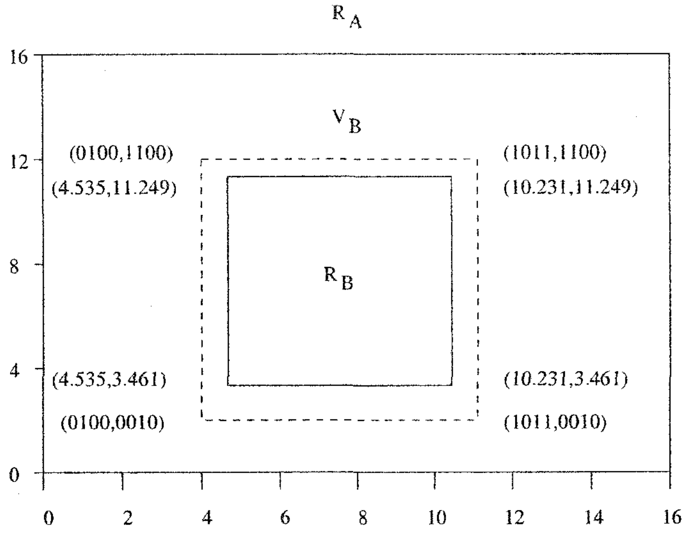


Figure 2.16: An example of how VBRs (V_B) approximate MBRs (R_B) in the A-tree

same level of accuracy throughout the data space, the approximations in the A-tree adapt to the data distribution. This allows VBRs in more dense regions to be approximated by more bits and increases the accuracy of the VBRs.

Like the SR-tree, the A-tree uses minimum bounding spheres (MBS) to insert objects. The A-tree does not use the MBSs for similarity search because experiments have shown they are not as effective as MBRs for pruning branches in high-dimensional spaces. In high-dimensional spaces the hyper-volume of MBSs increase more rapidly than MBRs which increases the probability of visiting the node. Updating the A-tree is similar to the SR-tree, the main difference is the VBRs have to be updated and the MBSs are stored separately as they are only used for the insertion of objects.

Nodes in the A-tree consist of one MBR and its children VBRs. This allows each node to contain information about two levels of the tree as each VBR in a node covers its entire subtree. VBRs approximate MBRs using less bits, but to ensure an exact answer, each VBR completely contains the MBR it approximates, e.g., in Figure 2.16, V_B completely contains R_B.

Searching the A-tree is similar to the R-tree. For similarity search nodes are visited in order of the minimum distance to its MBRs so the nearest sub-trees are searched first. Nodes are only pruned if the query region does not intersect the VBR of a node. If the VBR of a node intersects the query region, then the node containing the actual MBR that the VBR approximates is accessed. This process continues down to the leaf level of the tree.

In most tree index structures, the number of entries in a node is less than the maximum number of entries, so there is space not being used in nodes. The A-tree employs full utilization to make full use of all storage in a node. The empty space is used to increase the accuracy of the approximations (VBRs) and reduce the approximation error. This makes it less likely to access a subtree because the query region intersects the VBR, but does not intersect the MBR. Performance tests have shown the A-tree to access less disk pages than the VA-File and SR-tree for high-dimensional data.

It has been shown that hierarchical, directory based indexing structures are more efficient in low-dimensional spaces. These indexing structures tend to exhibit the dimensionality curse and are outperformed by a simple sequential scan of the original data set in higher dimensions. Linear scan-based compression techniques are more effective in high-dimensional data spaces as they tend not to suffer from the dimensionality curse, i.e, the VA-File. The IQ-tree [6] is designed from examining the R*-tree and the VA-File. The IQ-tree's intention is to perform well in both low-dimensional and high-dimensional spaces by using the data partitioning strategy of the R-tree family of indexing structures and the space partitioning strategy used by the VA-File.

The IQ-tree is a combination of the R*-tree and the VA-File and tries to combine the benefits of both. It consists of 3 levels. The first level is a flat directory of MBRs (Figure 2.17(a)). As in the R*-tree, the MBRs describe the region containing all the objects in its subtree. The main difference is the IQ-tree does not use a tree structure to store the MBRs as the R*-tree does, but instead uses a flat directory file which can be read sequentially to prevent expensive random disk accesses. The MBRs are also not constructed down to the leaf level, i.e., the data is only partitioned as far as it is considered beneficial. The stopping point is chosen based on a cost model to maximize the efficiency of the IQ-tree. The second level of the IQ-tree is similar to the VA-File and consists of quantized data pages (Figure 2.17(b)). At this level the IQ-tree partitions the data space containing the objects (the MBRs). Unlike the VA-File, the IQ-tree uses different compression rates, i.e., variable bit encoding is used. This allows higher density regions of the data space to use more bits and increase the accuracy of the approximations when required. The data pages are a fixed length so the number of approximations stored on each page depends on the accuracy of the approximations. The third level of the IQ-tree (Figure 2.17(c)) consists of the original data pages of the feature vectors of the objects. The actual feature vectors of objects not pruned must be accessed in the refinement step. In Figure 2.17(b), the quantized data page with 32 bits has a null pointer because when it is determined the approximation should be quantized to 32 bits, the original feature vector is stored instead, therefore the original feature vector will be accessed in level 2 and is omitted from level 3. A cost model is derived for the IQ-tree and is used to determine the compression rate and accuracy of the approximations which provides the most efficient similarity search for the data set. The goal is to find the lowest compression rate which provides sufficient accuracy to prune most objects not in the answer set to prevent having to access many objects in the third level of the tree.

For similarity search, the IQ-tree begins by sequentially reading the entire flat directory file of MBRs. The distance to the MBRs is computed in the same way as the R*-tree. MBRs not intersecting the query region are pruned and the objects in these MBRs are not considered further. The approximations in the second level of the tree are in the same order as the MBRs in the first level and the approximations of objects in MBRs not pruned are read sequentially. Each time a pruned object is reached, the IQ-tree calculates where the next object to be read is located. The IQ-tree either reads the pruned objects or skips over them depending on which is more efficient. The reason is it can be faster to read a few pruned objects sequentially rather than use a more expensive random disk access to skip to the next object not pruned. The third level of the IQ-tree is used for the refinement step and all points which have not been pruned in the first two levels are accessed.

The IQ-tree has been shown to outperform the X-tree and the VA-File in

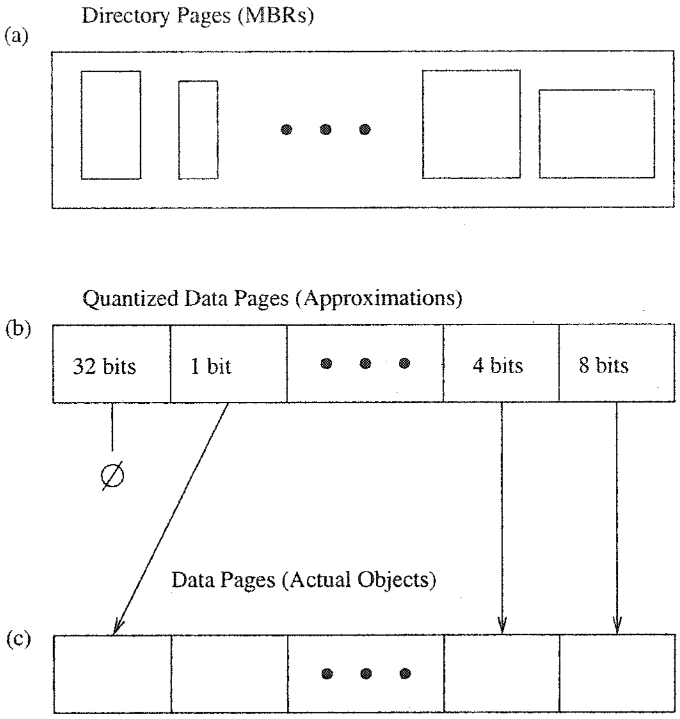

Directory Pages (MBRs)
(a)
Quantized Data Pages (Approximations)
(b)
32 bits
1 bit
4 bits
8 bits
∅
Data Pages (Actual Objects)
(c)

Figure 2.17: The IQ-tree Structure

uniform and clustered data sets. The IQ-tree combines both the tree structure and the minimum bounding regions of the R*-tree and the approximations of the VA-File to make an indexing structure useful for both low and high dimensional spaces.

2.2 Indexing Structures for General Metric Data

General Metric data, unlike vector data, does not have to consist of all attributes being numerical, nor does the number of attributes have to be constant for each object, i.e., video clips, audio clips, finger prints, etc. The only information available to index metric data is the distance between all data objects can be computed. Metric Access Methods (MAM) require the distance function is a metric so objects can be pruned using the triangular inequality property.

In metric spaces objects can be pruned during similarity search using the triangular inequality property. By using precomputed distances between ob-

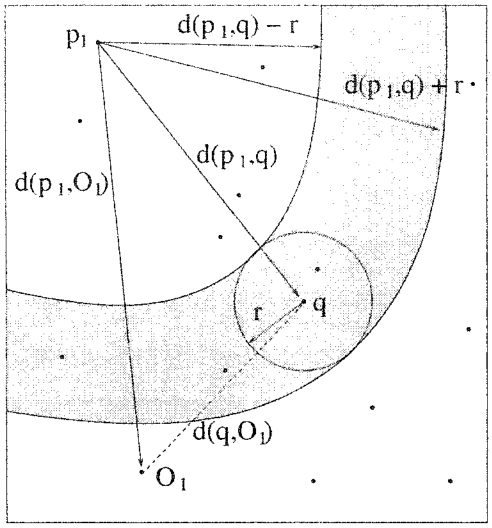


Figure 2.18: Pruning objects in Metric Space using the triangular inequality

jects and pivots, objects which are far away from the query object can be discarded without computing the object's actual distance to the query object. For a query object q and a query radius r, the pruning ring for a pivot p_i is defined by two radii, $d(p_i, q) - r$ and $d(p_i, q) + r$. In Figure 2.18, O_1 lies outside p_1's pruning ring. As such, O_1 can be pruned because the triangular inequality says that $d(q, O_1) + d(p_1, q) \geq d(p_1, O_1)$, hence $d(q, O_1) > r$ and therefore O_1 cannot be a possible answer to query q. This way, by using $d(p_1, q)$ and $d(p_1, O_1)$ (which is precomputed), the computation of $d(q, O_1)$ is avoided.

Metric Access Methods (MAMs) can prune objects or groups of objects from the answer set without computing the object's actual distance to the query object using the triangular inequality property. To make use of the triangular inequality property, MAMs use pivots and precompute distances from the data objects to the pivot objects. The methods for selecting pivots for different MAMs differ considerably and many pivot selection techniques have been proposed. For general metric spaces it may be impossible to artificially create pivots, i.e. finger prints, therefore pivots must be chosen from the data set. For vector spaces, pivots can be created artificially inside or outside the data space and techniques specific to vector spaces can be applied.

The most simple method to select pivots is using the random pivot selection technique. This method arbitrarily chooses objects in the data set and uses them as pivots. Selecting pivots in this manner can produce pivots with high pruning power and pivots with low pruning power. Generally this method does not select pivots as good as other pivot selection techniques.

The authors in [16] propose to select pivots from the data set using the HF algorithm. The HF algorithm begins by randomly choosing an object s_r in the data set and the object farthest from s_r is selected to be the first pivot p_1. The second pivot object is chosen to be the object farthest from p_1. Additional pivot objects are found by calculating the following error:

$$error_i = \sum_{j=1}^{k} |d(p_1, p_2) - d(p_j, s_i)|$$

for each object not yet chosen as a pivot object, where k is the number of pivots already chosen. The object that minimizes the $error_i$ is selected as the next pivot. In Figure 2.19, if s_r is the randomly selected object in the data set, the HF algorithm will choose p_1 and p_2 as pivots because p_1 is the farthest object from s_r and p_2 is the farthest object from p_1. A similar method is proposed in [25] except the first pivot object is chosen randomly.

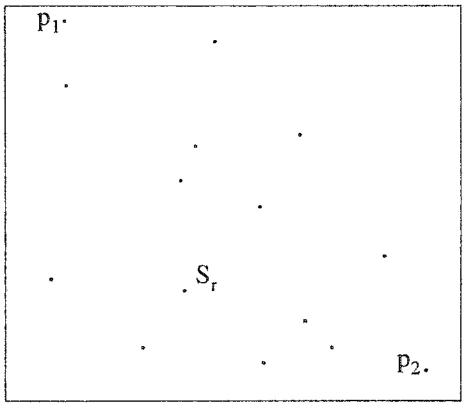


Figure 2.19: The pivots chosen with the HF algorithm

The authors state that pivots should be chosen to be "orthogonal, far apart, and with the origin coinciding with the query center". However, the HF algorithm cannot guarantee the pivot objects will be orthogonal to each other.

The HF algorithm selects pivots with higher pruning power than randomly selecting pivots but does not guarantee all the pivots will be good.

The Incremental Selection algorithm (ISA) is proposed in [12] to select pivots for MAMs. ISA selects pivots that best distinguish the distances between pairs of objects chosen from the data set. In Figure 2.20 ISA will select p_2 rather than p_1 because $|d21 - d22| > |d11 - d12|$. The pivot p_1 would be considered a poor pivot because $d11 \approx d12$, i.e., p_1 cannot distinguish any difference in the position of O1 and O2 because in metric space only the distance between objects in known.

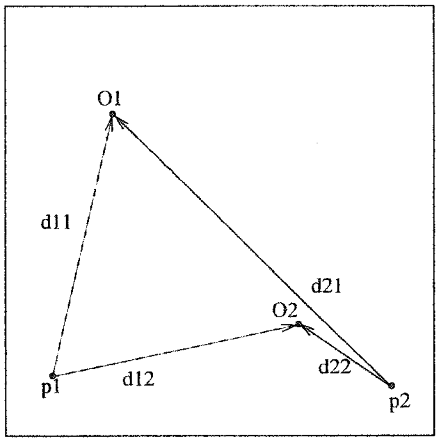


Figure 2.20: Selecting pivots using the Incremental Selection technique

The Incremental Selection Algorithm selects the pivots which maximize the following equation: $D_{\{p_i\}}([A_r],[A'_r]), 1 \leq r \leq A$ where A is the number of pairs of objects selected from the data set and $D_{\{p_i\}}([A_r],[A'_r])$ is $\sum |d(A_r,p_i) - d(A'_r,p_i)|, 1 \leq r \leq A$. ISA selects the first pivot as the one which maximizes this criterion. Additional pivots are chosen which maximize the distance between the same pairs of objects considering the first $i-1$ pivots are fixed. This is given by the following equation: $D_{\{p_1,\ldots,p_i\}}([A_r],[A'_r] = max(D_{\{p_1,\ldots,p_{i-1}\}}([A_r],[A'_r],D_{\{p_i\}}([A_r],A'_r]))$.

Based on experiments in [12] and from our own experiments, we have found this method to select pivots with higher pruning power than the other pivot selection techniques for general metric spaces. By selecting the objects in the data set as pivots which best distinguish the distances between pairs of objects in the data set, these pivots have a higher probability of distinguishing the distance between objects in the data set and the query objects. This allows a greater number of objects to be pruned at query time, decreasing the time required for MAMs to answer similarity queries.

For general metric spaces, the Incremental Selection algorithm is effective for selecting pivots with high pruning power. A pivot selection technique proposed for data sets in the vector space is Principal Component Analysis (PCA) [10]. PCA is shown to select pivots which exhibit higher pruning power and takes advantage of the properties of vector space.

PCA identifies the axes in a D-dimensional data set which exhibit the greatest variance. The principal axes of the data set are orthogonal to each other and by choosing pivot objects on the principal axis, the pivots will be orthogonal to each other. This is one of the desired properties stated in [16], as the volume of the intersection of the pruning rings is reduced in this situation.

To perform PCA on a data set, the dispersion matrix is determined by $t_{ij} = (1/N) \sum_{i=0}^{N} [(x_i - \bar{x_i})\ (x_j - \bar{x_j})]$ where $\bar{x_j}$ is the average value of the x_j's. The eigenvectors (ϕ_i) and associated eigenvalues (λ_i) of the dispersion matrix can be determined by singular value decomposition. The first principal axis is given by the eigenvector associated with the largest eigenvalue, the second principal axis is given by the eigenvector associated with the second largest eigenvalue, so on and so forth. After the principal axes are found, the pivots are created as artificial points *outside* the data space on the principal axes [10]. The pivot objects should be outside the data space so that the section of the pruning ring that intersects the data space will resemble a stripe which is perpendicular to the principal axes (see Figure 2.21). If the pivot objects are not chosen outside the data space, the pivots may not be orthogonal for every query object. In Figure 2.22 the pivot objects are chosen to occur on the principal axes of the data set but inside the data space. In this example we can see that even though the pivots are selected on the principal axes of the data set, the pivots may not be orthogonal with respect to the query object. This also may cause the intersection of the pruning rings to be larger, the intersection of the pruning rings in Figure 2.22 is much larger than the intersection of the pruning rings in Figure 2.21. By selecting the pivots outside the data space, the pivots will be orthogonal for all query objects.

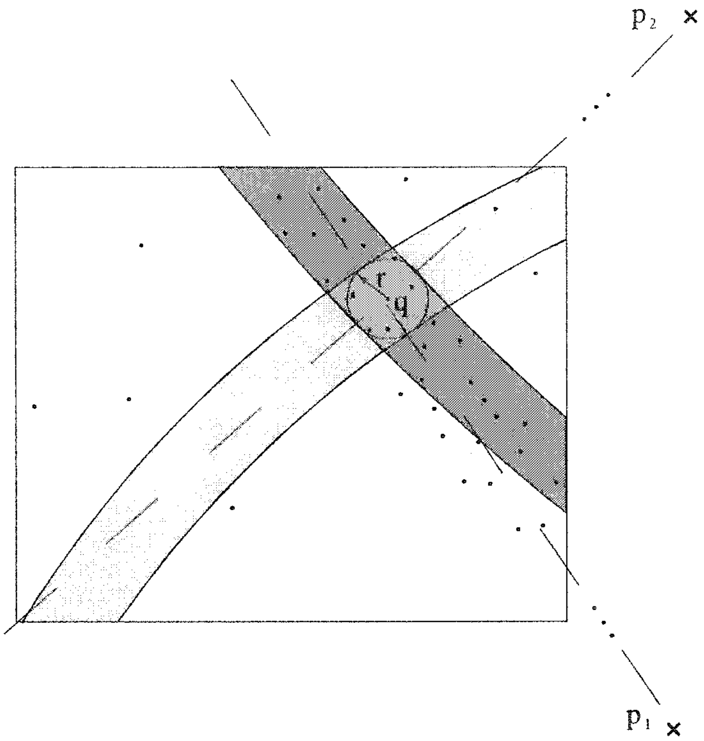


Figure 2.21: Comparing the pruning power of pivots on each of the principal axes selected outside the data space, the pruning ring of p_1 is perpendicular to the first principal axis of the data set

Choosing the set of pivots based on PCA results in the first pivot object having on average greater pruning ability than the other pivot objects. This is because the first pivot object occurs on the first principal axis and therefore its pruning ring will intersect the major axis of the data set. Figure 2.21 illustrates this concept. The pruning ring of p_1 prunes many more objects than the pruning ring of p_2 because the portion of p_1's pruning ring, which intersects the data set, cuts perpendicularly through the principal axis of the data set.

The indexing structures proposed for vector data cannot index all general metric data sets as they rely on the spatial information of the objects. On the other hand, MAMs can index and perform similarity search on vector data

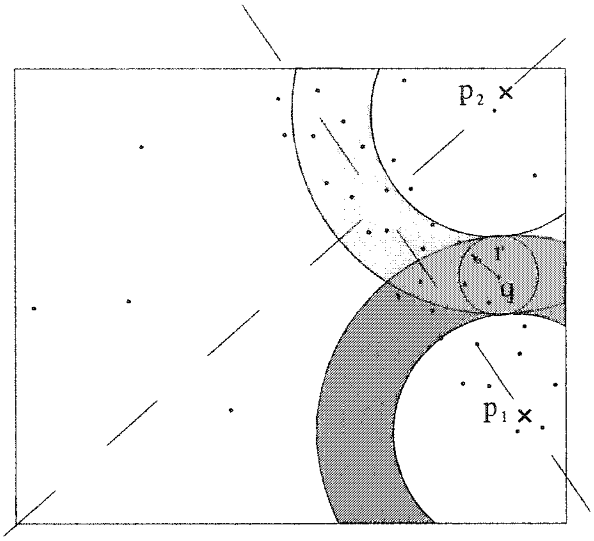


Figure 2.22: Choosing pivot objects inside the data space cannot guarantee the pivots will be orthogonal for every query object

because often a L_p-metric is used as the distance function. This allows MAM to treat vector data as if it is general metric data and index the data set using only the distance between objects and not information about the spatial distribution of the objects in the data set. Next we describe a few popular Metric Access Methods, the BK-tree, the FQ-tree, the Fixed Height FQ-tree, the FQ-array, the MVP-tree, the M-tree, the Slim-tree, Multiple Similarity Queries and the OMNI access method.

The first metric tree based on the triangular inequality property is the BK-tree (Burkhard-Keller tree) [11]. The authors developed an indexing structure to minimize the number of distance calculations required to answer a similarity query for general metric data. Only discrete distance functions are considered for the BK-tree and the number of disk accesses is not considered as an efficiency measure.

The BK-tree arbitrarily chooses a pivot from the data set and divides all

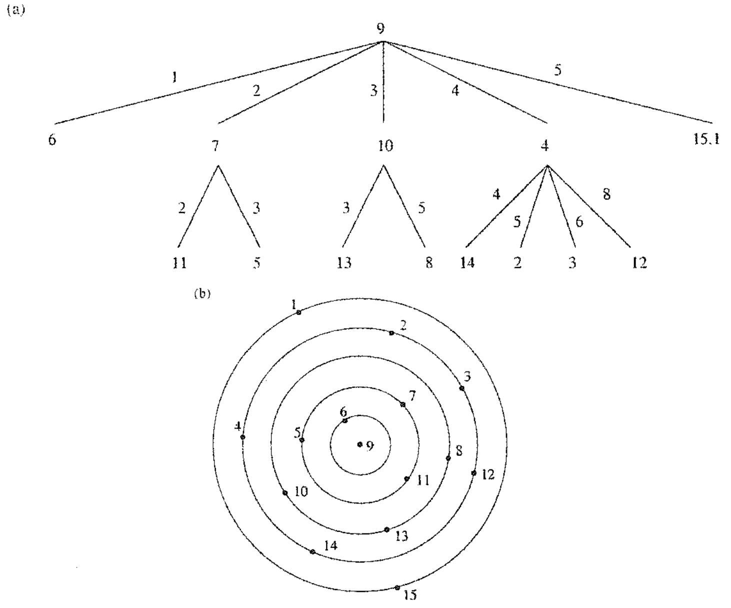


Figure 2.23: The BK-tree Structure

objects into subsets by their distance to the pivot. Since a discrete distance function is used, all objects in the same sub-tree are exactly the same distance from the pivot object at the root of the BK-tree. Additional pivots can be chosen from each subset and the tree can be built recursively with the subsets of objects being divided into smaller and smaller subsets. An example BK-tree is shown in Figure 2.23(a) and refers to the points in Figure 2.23(b). The stopping condition for building the BK-tree is when each branch has less than m objects in its sub-tree. In the example in Figure 2.23(a), $m = 2$, so the branches are divided into smaller groups until there are no more than two objects in each sub-tree. This leads to leaf nodes being at different heights in the tree.

For similarity search, the search algorithm for the BK-tree (Figure 2.24) begins by computing the distance between the query object q and the pivot at the root (object 9 in Figure 2.23(a)) and initially setting the NN to the root pivot p_r and the nearest neighbor distance NNdist to $d(q, p_r)$. The first branch in the BK-tree to be searched is the branch containing objects the same distance from p_r as q. This increases the probability the NN will be found quickly and the NN distance will be decreased so more branches in the tree can be pruned. Branches of the tree are pruned if they are not in the

```
BK-tree NN Search Algorithm: Find the nearest neighbor to the query object
Input  : Query Object $q$, Nodes in the BK-tree $p_i \in P$, Objects in the
         data set $s_i \in S$
Output: Nearest Neighbor to $q$ NN

NN = root pivot $p_r$
NN_dist = $d(q, p_r)$
j = NN_dist
foreach $k$ = j,j+1,j-1,j+2,j-2... do
    if $|k - j| < NN\_dist$ then
        Search_Sub(sub-tree at distance k from $p_r$, NN, NN_dist, $q$) // see
        Figure 2.25
return NN
```

Figure 2.24: BK-tree Nearest Neighbor Query Search Algorithm

```
Search_Sub Algorithm: Search the sub-tree
Input  : Nodes in the sub-tree to be searched $p_i$, Nearest Neighbor NN,
         Nearest Neighbor distance NN_dist, Query object $q$

if At leaf level of sub-tree then
    foreach $s_i \in$ leaf node do
        if $d(q, s_i) < NN\_dist$ then
            NN = $s_i$
            NN_dist = $d(q, s_i)$
else
    /* Not at leaf level of sub-tree */
    j = $d(q, p_i)$
    foreach k=j,j+1,j-1,j+2,j-2,... do
        if $|k - j| < NN\_dist$ then
            Search_Sub(sub-tree at distance k from $p_i$, NN, NN_dist, $q$)
```

Figure 2.25: The Search sub-tree Function for the BK-tree

range $d(q, p_i) \pm NNdist$ where q is the query object and p_i is the pivot. For each sub-tree not pruned, the distance between the query object and the pivot at the root of the sub-tree is computed. Branches in this sub-tree are then pruned in the same way as branches are pruned by the pivot at the root of the BK-tree. This process continues recursively until the leaf level of the sub-tree

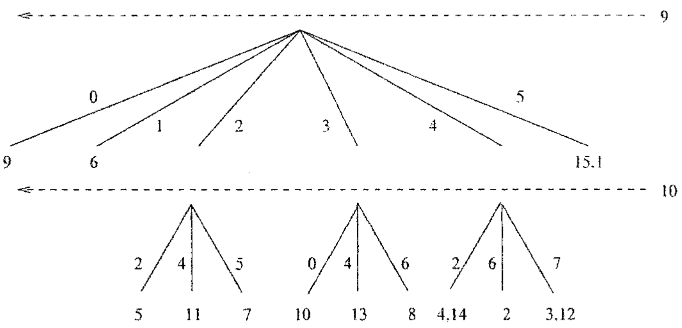


Figure 2.26: The FQ-tree Structure

is reached. The distance between all objects in branches not pruned and the query object is computed. This refinement step is necessary because although the method will not prune any objects in the answer set, objects not in the answer set may not be pruned. As nearer objects to the query object are found, NNdist is updated and more branches can be pruned using the new NNdist. For range queries, the radius is constant so the order the sub-trees are visited will not matter.

The Fixed-Queries tree (FQ-tree) [4] is similar to the BK-tree and is also based on the use of the triangular inequality property. The FQ-tree also tries to minimize the number of distance calculations for answering similarity queries and ignores the cost of disk accesses. The paper focuses on proximity (range) queries and only discrete distance functions are considered.

The FQ-tree, like the BK-tree arranges objects in a tree by their distance to an arbitrarily selected pivot from the data set. The major distinction of the FQ-tree is only one pivot is used at each level of the tree, each sub-tree in the BK-tree at the same level uses separate pivots. In Figure 2.26 we can see object 9 is used as the pivot at the root of the FQ-tree and object 10 is used as the pivot for all sub-trees at level 2. For the BK-tree (Figure 2.23) objects 7, 10 and 4 are all used as pivots for different sub-trees at level 2. The authors explain that when the distance function is expensive, traversing trees can be expensive because computing the distance between the query object and each pivot is costly. Only having one pivot per level makes it inexpensive to follow many paths of the tree.

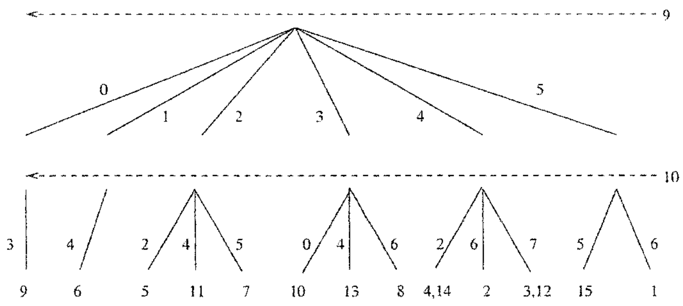


Figure 2.27: The Fixed-Height FQ-tree Structure

Searching the FQ-tree is similar to searching the BK-tree. The FQ-tree is searched by inspecting each branch of the tree. Any branches which do not fall in the range $d(q, p_i) \pm r$ can be pruned by the triangular inequality property where q is the query object, r is the radius range of search and p_i is the pivot at the i^{th} level of the tree. The distance between all objects in branches not pruned and the query object have to be computed in the refinement step to determine the final answer to the query. As mentioned for the BK-tree, the order branches are searched for range queries does not matter.

Like the BK-tree, sub-trees are only divided while there is greater than m elements in the sub-tree. In Figure 2.26, $m = 2$, therefore all the leaves of the tree contain no more than two objects. The authors mention the tree can be modified so all branches traverse to the same leaf level. This creates the Fixed-Height FQ-tree (Figure 2.27) and allows all objects to be pruned by all pivots. In the FQ-tree, if there are less than m objects in a sub-tree, the sub-tree is not divided further using more pivots, even if other sub-trees need to be divided further. If the distance function is expensive, the Fixed-Height FQ-tree should be more efficient than the FQ-tree because more objects can be pruned using pivots in the branches of the FQ-tree that were previously not sub-divided further. In Figure 2.26, object 6 can only be pruned using pivot 9, while in Figure 2.27, object 6 can also be pruned using pivot 10. Pivots in the FQ-tree and the Fixed-Height FQ-tree can be chosen other than randomly. Choosing pivots with higher pruning power will result in more branches in the tree being pruned and less distance calculations needing to be computed.

The FQ-array (Fixed Queries Array) was proposed in [13] and is very

Object	9	6	5	11	7	10	13	8	4	14	2	3	12	15	1
$d(P1,O_i)$	0	1	2	2	2	3	3	3	4	4	4	4	4	5	5
$d(P2,O_i)$	3	4	2	4	5	0	4	6	2	2	6	7	7	5	6

Figure 2.28: Fixed Queries Array

similar to a Fixed-Height FQ-tree. It consists of a two dimensional array of k distances from each object in the database to each of the pivots. In the example in Figure 2.28, $k = 2$, i.e., there are two pivots. The first pivot P1 is object 9 and the second pivot P2 is object 10 in Figure 2.23(b), this example uses the same pivots as the Fixed-Height FQ-tree does (Figure 2.27). A discrete distance function is used for the FQ-array as well and the objects are sorted by their distance to each of the pivots. In Figure 2.28, the distances from P1 to each object ($d(P1, Oi)$) are sorted. Likewise, the distances form P2 to each object ($d(P2, Oi)$) are sorted recursively with the distances to P1, i.e., if two objects have the same distance to P1, they are then sorted by their distance to P2. We can see in Figure 2.28, objects 5 and 11 are both distance 2 from P1, but object 5 occurs first in the FQ-array because it is closer to P2. The ordering of the objects in the array is the same as if we traversed a fixed-height FQ-tree from left to right (Figure 2.27).

Searching the FQ-array is also similar to the Fixed-Height FQ-tree as $d(q, p_i)$ is computed for each pivot p_i. The authors only consider range queries, so searching the FQ-array consists of finding the ranges $d(q, p_i) \pm r$ for each p_i in the linear array. A binary search can be used to find the intervals for each pivot because the distances in the array are sorted. Like the BK-tree and the FQ-tree, the actual distance between all objects not pruned and the query object must be computed to determine the final answer.

In that paper the measure of efficiency is also the number of distance calculations so adding more pivots should increase the efficiency as long as adding one extra pivot results in more than one additional object being pruned. The reason for this is for each pivot added, one more distance calculation needs to be computed (between the pivot and the query object). The authors say the performance is dependent on the number of pivots and the precision used to store the distance as they suggest if the distance function is discrete, a smaller precision may be used to store the distances. The number of disk accesses is not considered, therefore using a smaller level of precision is only useful if the amount of main memory is fixed. The FQ-array can also be adapted to use a continuous distance function.

The authors in [9] propose a static, height-balanced tree called the multi-vantage point tree (MVP-tree). The MVP-tree is an extension of the VP-tree [33] and attempts to provide a solution to some of the problems of the VP-tree. Like all the distance-based (metric) indexing structures, the MVP-tree requires a metric distance function, i.e., one that obeys the triangular inequality.

The MVP-tree arbitrarily selects a vantage point (pivot) from the data set and partitions the objects into two groups of equal cardinality, those closest to the vantage point and those farthest away. This is indicated by curve B in Figure 2.29(a). A second vantage point is then selected and the two groups are each divided into two more groups of equal cardinality (curves A and C), obtaining four branches in the tree (Figure 2.29(b)), one to represent each of the four partitions in Figure 2.29(a). This continues recursively until there are less than m objects in each sub-tree. The distances to the first p vantage points are stored for each object in the leaf level of the MVP-tree. These p distances are stored so they can be used as an additional filter to prune objects individually in branches which can not pruned. The MVP-tree can also be modified to hold more than 2 vantage points in each node.

The algorithm to answer range queries using the MVP-tree begins by computing the distance to the vantage points in the first node. Each branch in the tree which intersects the query region, i.e., branches in the range $d(q, p_i) \pm r$, where q is the query object, r is the radius of the range query and p_i is the vantage point, must be traversed to guarantee the correctness of the answer. The goal of the MVP-tree is to hopefully prune a significant number of branches quickly. When the search reaches the leaf level, the algorithm tries to prune objects individually using the stored distances to each of the first p vantage points using the triangular inequality property. The actual distance is com-

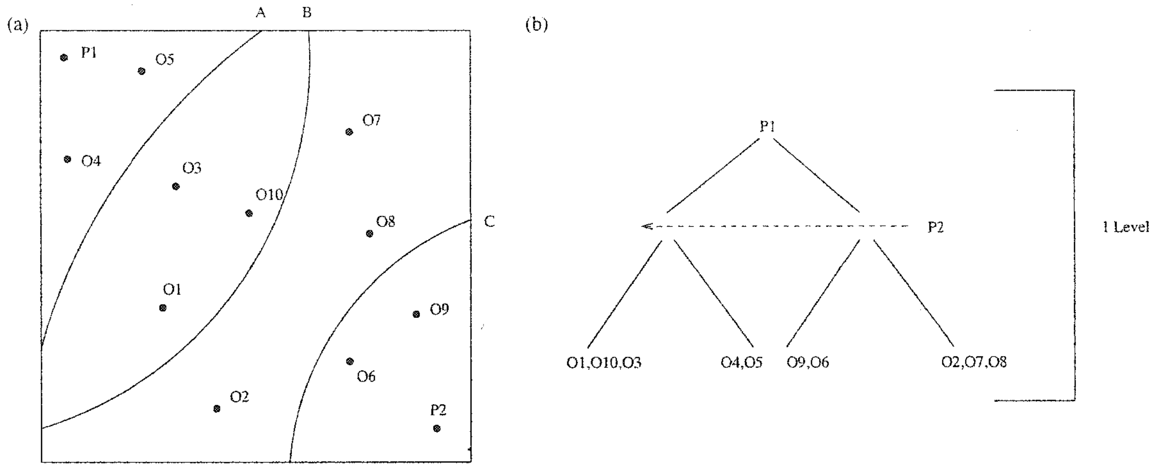


Figure 2.29: The MVP-tree Structure

puted to q for each object which cannot be pruned.

The MVP-tree differs in a number of ways from the VP-tree. In the VP-tree, each branch uses a different vantage point, this requires one distance calculation each time we traverse a node in the tree. The MVP-tree uses two vantage points in each node of the tree and each level of the MVP-tree corresponds to two levels of the VP-tree. This allows the second pivot in the node to be used by all branches, whereas the VP-tree would use a separate pivot for each sub-tree. An advantage of this characteristic is the MVP-tree can have larger fanouts while still utilizing the same number of vantage points. Constructing the VP-tree also requires one distance calculation between every data object and each vantage point in its path in the tree but does not store these distances to provide additional pruning. The MVP-tree stores the first p of these distances to provide additional pruning at the leaf level and save additional distance calculations.

The first vantage point at each level in the MVP-tree is chosen arbitrarily from the set of data objects in the sub-tree. The second pivot is chosen to be the farthest object from the first pivot in the subtree. The authors mention that any optomization technique to find pivots can be applied to the MVP-tree. The MVP-tree also does not consider the number of disk accesses as an efficiency measure, just the number of distance calculations.

The metric access methods previously discussed fail to consider the cost of disk I/Os for similarity searches in general metric spaces. The M-tree [14] is designed to minimize the number of distance calculations as well as the number of disk accesses. It is a paged, dynamic, height-balanced tree which uses only the distance between objects for indexing.

The M-tree consists of leaf nodes and internal nodes. Leaf nodes contain the actual data objects while internal nodes contain routing objects (similar to pivots). Each routing object is associated with a covering radius which is equal to the distance from the routing object to the farthest object in the sub-tree. Every object in the sub-tree of each node has to be within the covering radius ($r(O_i)$) of the routing object. This allows the M-tree to be able to apply the triangular inequality property to prune entire nodes using the distance to the routing object and the covering radius. The distance from the routing object to its parent is also stored in each node and can be useful for pruning branches without computing the actual distance between the query object and the routing object of the node when searching the M-tree.

In the example in Figure 2.30(a), the entire sub-tree of O_p can be pruned for q_1 by only computing $d(O_p, q_1)$ because the sphere defined by O_p and its covering radius ($r(O_p)$) does not intersect the query region of q_1, i.e., $|d(O_p, q_1) - r(O_p)| > r_{q_1}$. For q_2, the entire sub-tree of O_p can not be pruned by computing $d(O_p, q_2)$ alone because $d(O_p, q_2) - r(O_p) < r_{q_2}$, i.e., there are objects far enough away from O_p which could be in the answer set to the query. This means each child node of O_p (O_{ri}) must be compared to the query object individually. Since each sub-tree stores the distance to O_p, O_{r3} can be pruned without computing $d(q_2, O_{r3})$ because the distance from a routing object to its parent is stored and can be used to prune nodes. In the example $|(d(O_p, O_{r3}) + r(O_{r3})) - d(O_p, q_2)| > r_{q_2}$, i.e., it can be determined O_{r3} does not intersect the query region of q_2 based on the distance between O_{r3} and O_p. The distance to the remaining O_{ri} and the query object must be computed. After computing $d(O_{ri}, q_2)$ for each ri, O_{r1} and O_{r2} can be pruned while O_{r4} can not. The node containing O_{r4} must be accessed to determine if objects in its sub-tree are in the answer set to q_2 because $|d(q_2, O_{r4}) - r(O_{r4})| < r_{q_2}$.

The M-tree is suitable for range queries and K-nearest neighbor (K-NN)

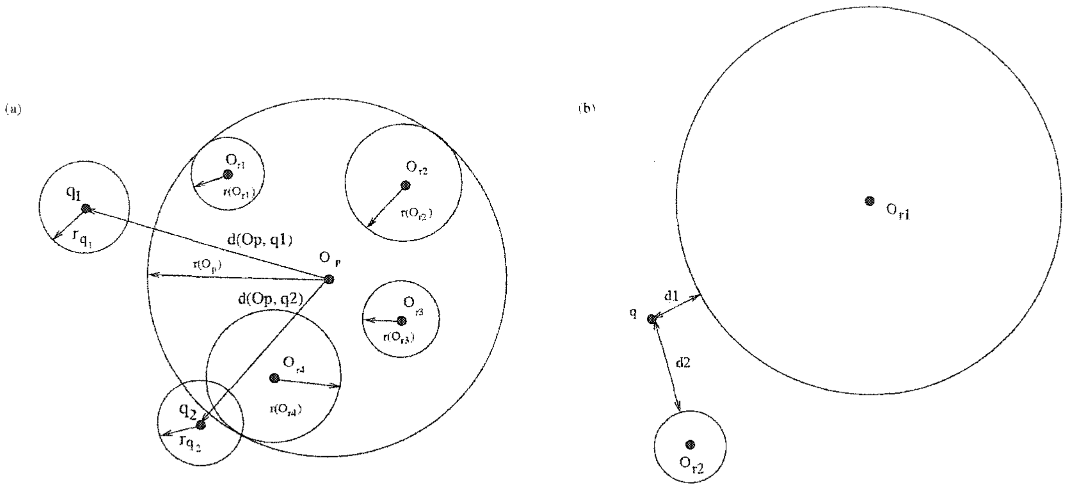


Figure 2.30: (a)Pruning nodes in the M-tree and (b)The order nodes are visited

queries. For range queries, all subtrees which cannot be pruned based on the covering radius and the distance between its routing object and the query object must be traversed to determine the final answer. For K-NN queries, like the R*-tree, subtrees are visited in order of the minimum lower bound distance to the query object. In Figure 2.30(b), O_{r1} would be searched first even though $d(q, O_{r1}) > d(q, O_{r2})$ because d1 < d2, i.e., the lower bound distance of O_{r1} is less than the lower bound distance of O_{r2}. The current K-NN distance is used as the radius to prune subtrees. The search algorithm stops when all subtrees have either been visited or pruned.

The M-tree is built by inserting objects into the most suitable sub-tree at each level of the tree. The criteria for the insertion algorithm is to follow the sub-tree for which the covering radius does not need to be increased and the second criterion is to choose the subtree which has the closest routing object. It is better to insert objects into sub-trees which do not have to be enlarged because the smaller the covering radius of sub-trees, the smaller the probability of visiting the node during similarity search. If the covering radius must be enlarged, the subtree which minimizes the enlargement is selected. When inserting objects leads to nodes being split, the split algorithm minimizes the covering radius and overlap of the two new split nodes. Although designed for general metric spaces, the M-tree has been shown to outperform the R*-tree in vector space.

The Slim-tree [19] is a dynamic indexing structure for metric data sets and is very similar to the M-tree. The triangular inequality property is used to prune sub-trees which do not intersect the query region. The main intent of the Slim-tree is to minimize the amount of overlap between nodes at the same level of the tree. This is to reduce the probability of having to access two sub-trees for the same queries and to reduce the time required to process similarity queries. The authors use the number of distance calculations and disk accesses as an efficiency measure.

The main structure of the Slim-tree is similar to the M-tree. The objects are stored in the leaves and the indexing nodes contain representative objects (similar to pivots and routing objects) which are the center of a minimum bounding regions which cover the objects in the sub-tree. Objects are inserted into the Slim-tree by locating the node at each level of the tree that covers it or the one whose representative object is closest. This is applied recursively at each level of the tree until the leaf level is reached. Minimum Spanning trees are used for splitting nodes which overflow during insertion to reduce the cost of splitting nodes. Results have shown Minimum Spanning trees produce trees as good as considering all pairs of objects for splitting nodes at only a fraction of the cost.

Regions in the tree can overlap each other, but the Slim-tree tries to minimize the overlap of nodes. In vector space, the overlap of two nodes is computed as the volume of common space shared by two nodes. There may be no concept of actual space with dimensions for metric data, so the volume of common space shared can not be computed. The authors define the amount of overlap for a metric tree as the number of objects which are covered by more than one sub-tree at the same level of the tree. The authors note good trees should have little or no overlap among nodes at the same level as this may lead to searching additional branches of the tree during similarity search.

A Slim-down algorithm is presented which decreases the amount of overlap in the tree producing tighter trees to reduce the total search time. The Slim-down algorithm selects a node i in the tree and finds the furthest object c

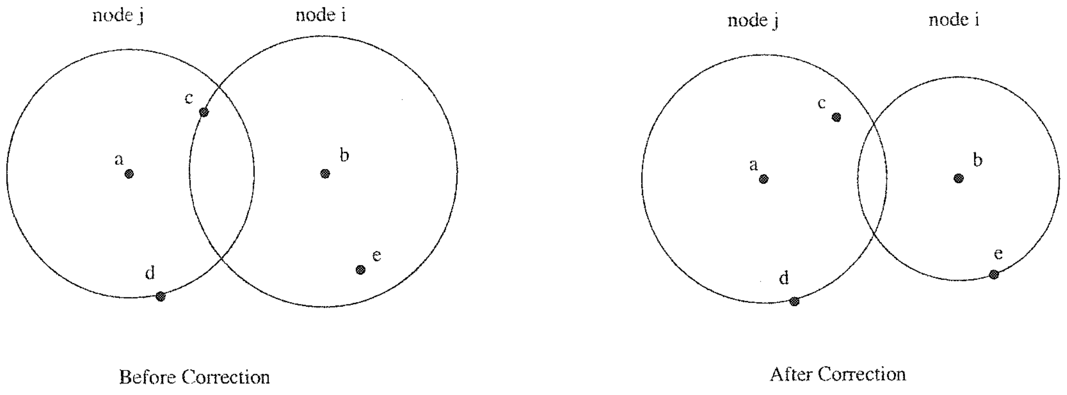


Figure 2.31: Before and after running the Slim-down algorithm

from the representative object of the node. The algorithm searches all other nodes at the same level of the tree for a node that also covers object c. If another node j does cover object c and is not full, c is removed from node i and inserted into node j. If node i is not empty, its covering radius is corrected. If node i is empty, then it is deleted. This can lead to fewer nodes and higher storage utilization because fewer nodes are used to index the same data set. The Slim-down algorithm is applied to all nodes of the tree and if any objects move during a full round of the algorithm, then the algorithm is applied again to all nodes at this level of the tree. Figure 2.31 illustrates the algorithm. Object c is the farthest object from the representative in node i and node j also covers object c. The algorithm moves object c to node j and node i's radius is corrected. After the correction, nodes i and j do not have any objects which are covered by both nodes.

Experiments have shown the Slim-tree requires fewer disk accesses and distance calculations compared to the M-tree. One reason this occurs is the Slim-tree has higher storage utilization of nodes and therefore requires less nodes to index the same data set. The main reason for this is the Slim-down algorithm is applied once the tree is built to rearrange parts of the tree making similarity search more efficient.

The authors in [10] propose to speedup similarity search processing in metric databases by using information computed from previous queries. By storing distances computed to the previous queries, the triangular inequality property can be used to prune objects which are far away from the query object.

This method is very similar to the techniques proposed in [4, 11, 13]. The method computes the distance from the current query object to the previous query objects and uses the previous query objects as pivots p_i. The objects in the data set are processed sequentially. For each object, if $|d(p_j, s_i) - d(q, p_j)| > r$, where q is the query object, s_i is an object in the data set and r is the query radius, then object s_i is pruned from the answer set. The actual distance between the query object and all objects not pruned must be computed to determine the final answer.

The authors also suggest that reference objects (pivots) can be selected prior to any queries and used to prune objects instead of using previous queries. Reference objects should be selected to be orthogonal to each other in order to prune the most objects. A method is given to select the best reference objects for vector space using Principal Component Analysis (PCA). The reference objects are selected to occur on the major axis of the data set which are orthogonal to each other. This method is also shown to achieve linear speedup when used in a parallel shared-nothing environment. The paper also proposes that queries can be processed in sets so data pages only have to be read once for a set of queries, this reduces the time required for similarity search per query.

Another MAM very similar to [10] is the OMNI Access method [16]. The main idea is to reduce the cost of similarity search by quickly filtering a large portion of objects which are not part of the answer set.

The OMNI Access method [16] selects pivot objects using the HF algorithm. It should be noted that any pivot selection technique can be used with the OMNI Access method. The distances between each object in the database and each pivot object is computed and stored on disk. These distances are used to prune objects during similarity search. Three methods are proposed which use the OMNI-Access method.

The first method (OMNI-sequential) begins by computing the distances between the query object and each pivot object. All the precomputed distances are read sequentially and each object not in the range $|d(q, p_j) - d(s_i, p_j)| > r$

for each pivot object p_j, query object q, data object s_i and radius r, is pruned using the triangular inequality property. This produces a candidate set for which a refinement step is needed to determine the final answer. During the refinement step, each object not pruned is accessed from the original data file and the actual distance to the query object is computed. For K-nearest neighbor queries, the current K-NN distance is used to prune objects. The current K-NN distance is updated as nearer objects to the query object are found. The OMNI-sequential method for K-nearest neighbor queries is not as efficient because there is no initial radius to use for pruning and the nearest neighbor radius can only be refined by computing the distance between the query object and the actual data objects in the database. In the beginning the algorithm will have to read all the actual data objects until the K-NN distance becomes small enough for objects to be pruned.

The second and third methods use existing index structures. The second method (OMNI B-tree) uses k B-trees, one to store the distances to the data objects for each of the pivot objects. The B-trees are then used to find all the objects in the range $|d(q, p_j) - d(s_i, p_j)| > r$ for each pivot object p_j. A list of candidate objects (objects not pruned) are produced from each B-tree and are merged together to produce one candidate set. A refinement step is also necessary to determine the final answer. For K-nearest neighbor queries an initial radius is approximated, if the resulting answer set does not produce at least K answers, the radius is enlarged until at least K-nearest neighbors are retrieved.

The third method (OMNI R-tree) uses an R-tree, the coordinates are the distances to each of the pivot objects. For range queries, all branches in the OMNI R-tree intersecting the query region are searched. When the leaf level of the tree is reached, the distance to all objects in the leaf node must be computed to the query object to determine the final answer. For K-NN queries, an initial radius can be estimated and enlarged if less than K-nearest neighbors are retrieved like the OMNI B-tree does. A second method is to perform K-NN search in the OMNI R-tree by searching the nearest sub-trees to the query object first. The K-NN distance is updated as nearer objects to the query object are found. Each time the K-NN distance is updated, branches in the OMNI R-tree are pruned.

The OMNI Access methods can be used with any metric distance function. It has been shown to be quite efficient, e.g., it outperforms the Slim-tree. In the following chapter the OSEQ access method will be described in greater detail because it forms the basis for two of our access methods, the OSEQ$^+$ and the OSEQ*. We choose to base two of our access methods on the OSEQ because it is simple to implement, linear scan-based and efficient.

Chapter 3

The OSEQ+ and the OSEQ* Access Methods for High-Dimensional Data and Metric Data

3.1 OMNI Access Method

The Omni Access method has been recently proposed in [16] and resulted in the family of Omni structures. Its noteworthy mentioning that variations of the same idea have also been proposed and used in [4, 10, 11]. Table 3.1 summarizes the notation that will be used in this section.

Table 3.1: Summary of notation

P	set of pivots $\{p_1, p_2, ..., p_k\}$
S	database of objects (s_i)
$d(x, y)$	distance between objects x and y
N	number of objects in the data set
D	number of dimensions in the data set
q	query object
r	query radius for a range query
$d_{min}(p_j)$	$d(p_j, q) - r$
$d_{max}(p_j)$	$d(p_j, q) + r$
$\mathcal{P}$	preferential pivot
k	number of pivots

The main idea behind the Omni Access method is to reduce the cost of similarity search by quickly filtering a large portion of objects which are not part of the answer set. The OMNI Access method uses a set of pivots, $P = \{p_1, p_2, ..., p_k\}$, for which the distance between the pivot objects and every object in the database is precomputed. Each distance $d(p_j, s_i)$ is referred to as an OMNI-coordinate and the set of distances $\{d(p_1, s_i), d(p_2, s_i), \dots, d(p_k, s_i)\}$ for each object s_i in the database S, is referred to as the OMNI-vector of the object. These sets of distances are used to prune candidate objects for a range query as follows. For a query object q, query radius r and $p_j \in P$, p_j's pruning ring is defined by two radii, $d_{min}(p_j)$ and $d_{max}(p_j)$, where $d_{min}(p_j) = d(p_j, q) - r$ and $d_{max}(p_j) = d(p_j, q) + r$. An object $s_i \in S$ is pruned if $|d(p_j, s_i) - d(p_j, q)| > r$, where $d(i, j)$ is a metric distance function that measures the distance between objects i and j. For instance, in Figure 3.1(a), for the query object q and query radius r, O_4 lies outside p_1's pruning ring (shaded area) and can be pruned because by the triangular inequality $(d(p_1, q) + d(q, O_4) \geq d(p_1, O_4))$, $d(q, O_4) > r$ and therefore O_4 cannot be a possible answer to query q. This way, by using $d(p_1, q)$ and $d(p_1, O_4)$ (which is precomputed), the computation of the actual distance $d(q, O_4)$ is avoided. One should note that reducing the number of distance calculations can be very important since the distance function can be expensive, (e.g., [28, 30]), where the distance is obtained via the solution of a network transportation problem. In Figure 3.1(b) we can see that the candidate space (shaded area) can be reduced further by using additional pivot objects; for instance by adding pivot p_2, O_1 is now also pruned.

The OMNI Access method does not prune any objects which are part of the answer set, but it may yield false positives, requiring a refinement step to determine the exact answer. In Figure 3.1(b), only O_2 is in the answer set to the query object q and query radius r, but O_3 cannot be pruned because $|d(p_1, O_3) - d(p_1, q)| < r$ and $|d(p_2, O_3) - d(p_2, q)| < r$, i.e., O_3 lies in the shaded region in Figure 3.1(b). The actual distance between the query object and each object not pruned must be computed to determine which objects are part of the final answer. Only after this refinement step will object O_3 be eliminated as a possible answer to the query. Hence, it is important to reduce the size of the candidate set as much as possible because retrieving objects from disk and computing distance calculations in the refinement step can be

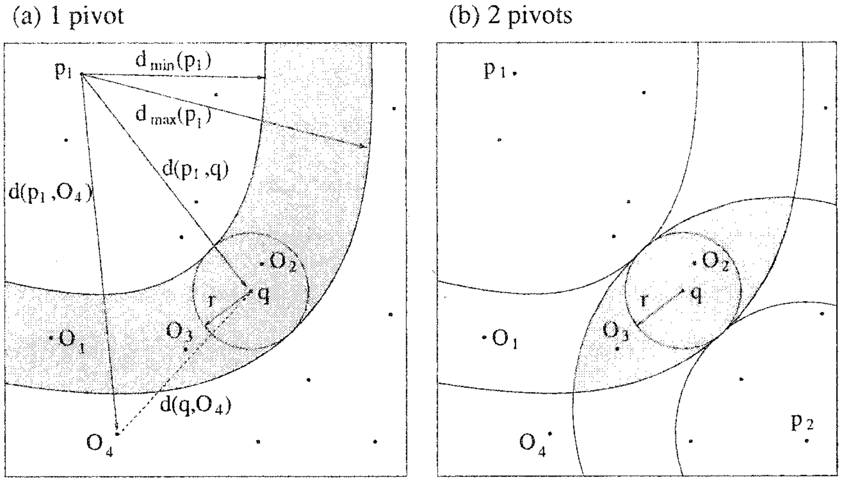


Figure 3.1: Pruning candidate objects using pivots

expensive.

In order to prune objects using pivots, a radius is required. For nearest neighbor search in the R-tree family of index structures, the radius can be refined as the tree is traversed up and down. The OSEQ$^+$ and OSEQ* methods are based on a single scan of the OMNI-file and the data file. To determine the actual nearest neighbor radius by refining the radius during the sequential scan, will require computing the actual distance of many objects to the query object while scanning the OMNI-file, which we want to avoid because it is expensive. For this reason we focus on range queries, which have a fixed radius, for the OSEQ$^+$ and OSEQ* methods.

The OMNI Access method selects pivots from the data set using the HF algorithm [16]. Like many Metric Access methods (MAM), the OMNI Access method can use pivots selected with any of the pivot selection techniques described in Section 2.2. The authors state that pivots should be "orthogonal, far apart, and with the origin coinciding with the query center." In Figure 3.1(b), the HF algorithm will likely choose p_1 and p_2 as pivots because these are the objects in the data set which are the farthest from each other[1]. Furthermore, the authors suggest that the number of pivots should be chosen based on the

[1] This will also depend on which object is arbitrarily selected from the data set to start the algorithm

intrinsic (fractal) dimensionality (δ) of the data set [26] and should be between $\lceil\delta\rceil + 1$ and $2 \times \lceil\delta\rceil + 1$. Based on our experiments, we observed that pivots should indeed be orthogonal, origin coinciding with the query center, but not necessarily far apart, rather the best pivots are closest to the query object. We explore this observation in Sections 3.2 and 3.3.

The algorithm to build the Omni-file (the file containing the Omni-vector for each object) for the Omni-sequential method (OSeq) is shown in Figure 3.2. The algorithm begins by selecting pivots using the HF algorithm. The distances between each pivot and each data object (the Omni-coordinates) are computed and stored in the Omni-file.

```
OSeq Build Algorithm: Building the Omni-file
Input  : The data set $s_i \in S$, the number of pivots $k$
Output: Pivot set $p_j \in P$, the Omni-file

Select the set of pivots $P$ with the HF algorithm
foreach $s_i \in S$ do
    foreach $p_j \in P$ do
        compute $d(p_j, s_i)$ and store in the Omni-file
```

Figure 3.2: OSeq Build Algorithm

The Omni-Sequential Search algorithm for range queries [16] is shown in Figure 3.3. The algorithm attempts to prune each object using their Omni-coordinates ($d(p_j, s_i)$) and processes the objects sequentially in the database. The Omni-vector for each object is typically much smaller than the actual size of the feature vector of the object and the entire Omni-file is read sequentially from disk. This ultimately allows many objects to be pruned by reading a small number of pages from disk and consequently yielding a fast sequential scan. As discussed above, the algorithm initially finds a candidate set which needs to be further verified. The refinement step for the algorithm is expensive, hence the goal is to prune as many objects as possible. In the refinement step, the distance between each object not pruned and the query object is computed. There is no order in the data file so the objects are accessed from disk using random disk I/Os.

A few observations that can be explored about the OSeq method are:

```
OSeq Search Algorithm: Find all answers to a given range query
Input  : The data set $s_i \in S$, Pivot set $p_j \in P$, query object $q$, query
         radius $r$, the OMNI-coordinates $(d(p_j, s_i))$
Output: answer set $A$ to the range query

Initialize candidate set $C$ to contain all $s_i \in S$
foreach $p_j \in P$ do
    calculate $d(p_j, q)$
    $d_{min}(p_j) = d(p_j, q) - r$
    $d_{max}(p_j) = d(p_j, q) + r$
foreach $s_i \in C$ do
    foreach $p_j \in P$ do
        if $d(p_j, s_i) < d_{min}(p_j)$ or $d(p_j, s_i) > d_{max}(p_j)$ then
            remove $s_i$ from $C$ and break inner loop
foreach $s_i \in C$ do
    calculate $d(s_i, q)$
    if $d(s_i, q) \leq r$ then
        place $s_i$ in answer set $A$
return A
```

Figure 3.3: OSEQ Search Algorithm

1. The pivot objects selected with the HF algorithm are not equally good, some have higher pruning ability than others.

2. The OSEQ Search algorithm reads the entire OMNI-file from disk because the OMNI-vector for each object is stored together on the same disk page.

3. The OSEQ Search algorithm requires a large number random disk accesses in the refinement step to access the actual feature vectors of the candidate objects. This occurs because the actual distances of objects not pruned by the pivots have to be computed requiring the feature vectors of objects, which occur randomly in the data file, be read from disk.

In the following sections we describe techniques that address these problems.

3.2 Restructuring the OMNI-File – OSEQ$^+$

The OMNI Sequential$^+$ (OSEQ$^+$) method is designed to resolve some of the problems of the OSEQ method pointed out above and further decrease the time required to answer range queries for high-dimensional data and metric data. It improves on the original OSEQ by:

1. Finding a preferential pivot, i.e., a pivot object which on average prunes more objects than other pivots.
2. Restructuring the OMNI-file so that the OMNI-coordinates of each object are ordered and do not reside on the same disk page (and thus can be accessed independently).
3. Presorting the OMNI-file and the data file such that the objects not pruned are stored closer together on disk.

3.2.1 Preferential Pivot

If we can find a preferential pivot (one that prunes more effectively), denoted as $\mathcal{P}$, which can prune a significant number of objects, the remainder of the OMNI-coordinates of these objects will not be needed. These OMNI-coordinates will not have to be read from disk, which will speed up similarity query processing.

We conducted experiments using two data sets to test the pruning power of pivot objects selected by different pivot selection algorithms. The first data set is real and consists of 59,652 global color histograms (GCH), each having 64 dimensions, obtained from images in a set a of Corel CDROMS and will be referred to as the COREL data set. The second data set is synthetic and contains 1,000,000 uniformly distributed, 64-dimensional GCHs, hereafter this data set will be denoted as 1MGCH. For pivot selection algorithms such as the HF algorithm, where the order in which the pivots are selected does not indicate any importance, i.e., the main criteria is pivots are chosen to be far apart from each other, $\mathcal{P}$ is chosen to be the pivot which is closest on average to all objects in the database. Figure 3.4 shows the percentage of objects pruned by pivots selected with the HF algorithm. The pivot objects on the x-axis are sorted by their average distance to all objects in the data set. We can see the pivots which are closer on average to other objects in the data set prune more objects than the pivots farther away. One reason this occurs is pivot objects close to other objects better approximate the actual distance between these objects and the query object. Therefore, for the HF algorithm we select $\mathcal{P}$ as the pivot object whose average distance to all objects in the database is smallest.

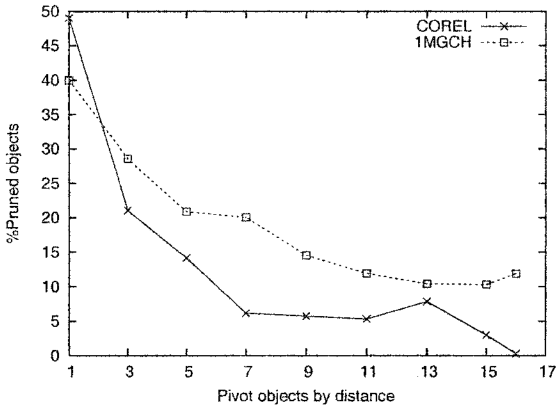


Figure 3.4: Pruning power of pivots selected by the HF algorithm and sorted by the pivots' average distance to all objects in the database

For pivot selection techniques such as Principal Component Analysis and the Incremental Selection algorithm, the order the pivots are selected in is important. For these algorithms, we select $\mathcal{P}$ as the most important pivot determined by the algorithm. For PCA, the most important pivot is p_1. Pivot p_1 is created on the first principal axis of the data set, i.e., the axis with the greatest variance, this causes the pruning ring of p_1 to be perpendicular to the first principal axis of the data set which is a desirable characteristic. In Figure 3.5, the pivots objects are in the order they are created using the PCA pivot selection technique. The curve for 1MGCH remains fairly stable and all the pivots have similar pruning ability because the data set is composed of uniformly distributed GCH. This means all dimensions in the data set are of approximately equal importance, therefore all the pivots have similar pruning power. For real data sets, i.e., COREL, often the data set is correlated so some dimensions are more important than others causing pivots to have different pruning ability. For the Incremental Selection algorithm, p_1 is also the most important pivot because it is chosen as the object which best distinguishes the distances between pairs of objects in the data set. Since p_1 is chosen as the pivot which overall best distinguishes the distances between objects in the data set, it should have the highest pruning ability. In Figure 3.6 we can see the first pivot selected with the Incremental Selection algorithm prunes more objects than the other pivots. The reason is each pivot selected with the Incremental Selection algorithm takes into consideration the pivots previously chosen. This means additional pivots are selected which distinguish well between objects in the data set that previous pivots cannot differentiate well between. An advantage of selecting $\mathcal{P}$ in this manner is $\mathcal{P}$ is selected independent of the query object and is chosen offline.

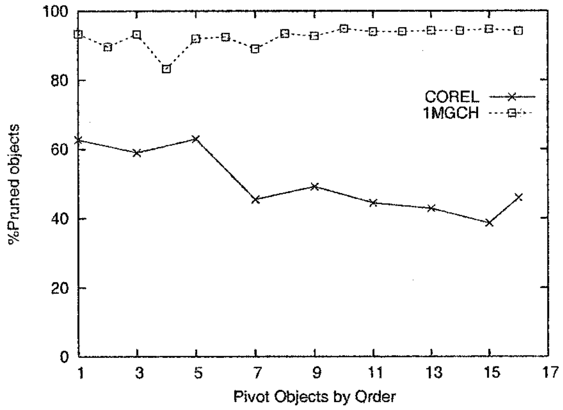


Figure 3.5: Pruning power of pivots selected using PCA and the pivots are sorted by the order they are created

3.2.2 Restructuring the OMNI-file

The OMNI-file contains the OMNI-coordinates for every object in the database. In order to benefit from $\mathcal{P}$ having better pruning power, the OMNI-file has to be restructured to prevent a complete sequential scan of the whole file as the OSEQ method does. The improved OMNI$^+$-file stores each of the OMNI-coordinates of an object on separate disk pages. This allows for one OMNI-coordinate of an object to be independently accessed from disk without retrieving the entire OMNI-vector for each object. This prevents a complete

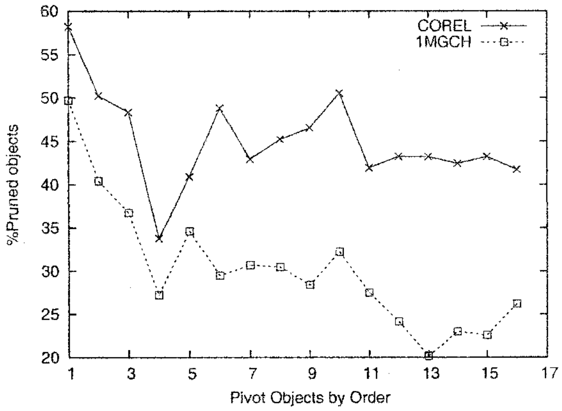


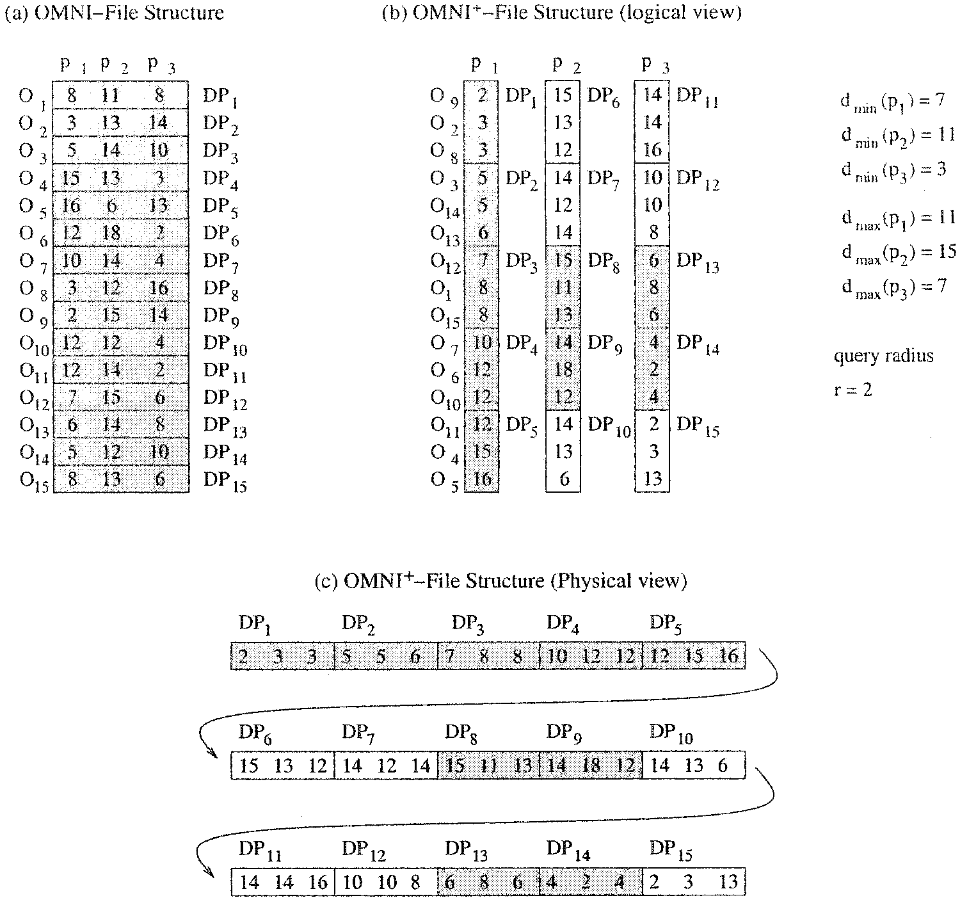
Figure 3.6: Pruning power of pivots selected with the Incremental Selection algorithm and the pivots are sorted in the order they are selected

sequential scan of the OMNI$^+$-file for each query.

The OMNI-coordinates are also sorted by each object's distance to $\mathcal{P}$. This causes the OMNI-coordinates of all objects not pruned by $\mathcal{P}$ to occur adjacent to each other on disk because all the objects which can not be pruned by $\mathcal{P}$ will occur in the range $d(\mathcal{P}, q) \pm r$. This allows the OMNI-coordinates relating to the remaining pivots to be read sequentially.

Figures 3.7(a) and (b) show an example of the original OMNI-file structure and the logical view of the improved OMNI$^+$-file structure. The shaded pages refer to the disk pages which will be accessed for the example query which refers to the objects in Figure 3.8. The original OMNI-file structure stores all the OMNI-coordinates of an object on the same disk page. This results in the complete OMNI-file needing to be read (15 pages accesses in the example of Figure 3.7(a)) for the OSEQ method because at least one OMNI-coordinate of each object must be compared for every query. The improved OMNI$^+$-file structure avoids reading every page containing the remaining OMNI-coordinates because the OMNI-coordinates are stored together by pivots, not by object. Thus, one OMNI-coordinate of an object can be read without reading every disk page containing OMNI-coordinates of that object from disk. This allows us to access the OMNI-coordinates individually for each object and not access many OMNI-coordinates which can not help prune additional objects.

The OMNI-coordinates in the OMNI$^+$-file are presorted by their distance to



(a) OMNI–File Structure

	p_1	p_2	p_3	
O_1	8	11	8	DP_1
O_2	3	13	14	DP_2
O_3	5	14	10	DP_3
O_4	15	13	3	DP_4
O_5	16	6	13	DP_5
O_6	12	18	2	DP_6
O_7	10	14	4	DP_7
O_8	3	12	16	DP_8
O_9	2	15	14	DP_9
O_{10}	12	12	4	DP_{10}
O_{11}	12	14	2	DP_{11}
O_{12}	7	15	6	DP_{12}
O_{13}	6	14	8	DP_{13}
O_{14}	5	12	10	DP_{14}
O_{15}	8	13	6	DP_{15}

(b) OMNI$^+$–File Structure (logical view)

	p_1		p_2		p_3	
O_9	2	DP_1	15	DP_6	14	DP_{11}
O_2	3		13		14	
O_8	3		12		16	
O_3	5	DP_2	14	DP_7	10	DP_{12}
O_{14}	5		12		10	
O_{13}	6		14		8	
O_{12}	7	DP_3	15	DP_8	6	DP_{13}
O_1	8		11		8	
O_{15}	8		13		6	
O_7	10	DP_4	14	DP_9	4	DP_{14}
O_6	12		18		2	
O_{10}	12		12		4	
O_{11}	12	DP_5	14	DP_{10}	2	DP_{15}
O_4	15		13		3	
O_5	16		6		13	

$d_{min}(p_1) = 7$
$d_{min}(p_2) = 11$
$d_{min}(p_3) = 3$

$d_{max}(p_1) = 11$
$d_{max}(p_2) = 15$
$d_{max}(p_3) = 7$

query radius
$r = 2$

(c) OMNI$^+$–File Structure (Physical view)

DP_1	DP_2	DP_3	DP_4	DP_5
2 3 3	5 5 6	7 8 8	10 12 12	12 15 16

DP_6	DP_7	DP_8	DP_9	DP_{10}
15 13 12	14 12 14	15 11 13	14 18 12	14 13 6

DP_{11}	DP_{12}	DP_{13}	DP_{14}	DP_{15}
14 14 16	10 10 8	6 8 6	4 2 4	2 3 13

Figure 3.7: The (Improved) OMNI$^+$-File Structure

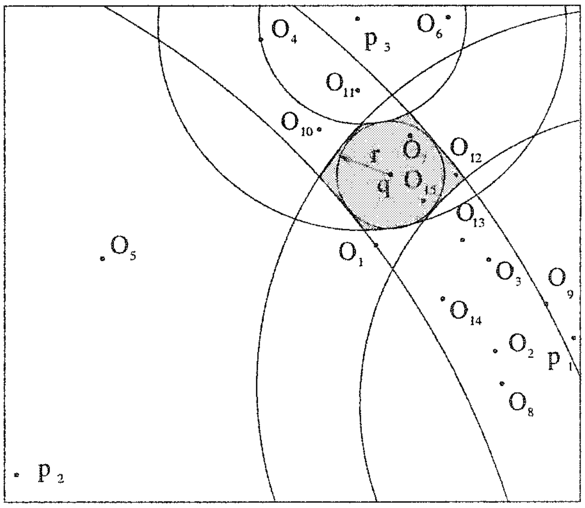


Figure 3.8: An example query using 3 pivots to prune objects

p_1 (which is also denoted by $\mathcal{P}$). All objects O_i where $d_{min}(p_1) > d(p_1, O_i)$ or $d_{max}(p_1) < d(p_1, O_i)$, i.e., all objects which reside outside of p_1's pruning ring, are pruned and the rest of the OMNI-coordinates for the pruned objects are not required. In Figure 3.7(b), by using the improved OMNI$^+$-file structure, disk pages $DP_6, DP_7, DP_{10}, DP_{11}, DP_{12}$, and DP_{15} will never be read from disk using the OSEQ$^+$ Access method because the disk pages only contain OMNI-coordinates for objects which have already been pruned. By examining Figure 3.8, we can see objects O_{12}, O_{15} and O_7 cannot be pruned by any of the pivots as they are located in the intersection of the three pruning rings (shaded area). Their actual distances must be computed to determine whether they belong to the answer set. Even in this trivial example, the improved OMNI$^+$-file structure reads only 9 pages of OMNI-coordinates from disk compared to the original OMNI-file structure which requires reading all 15 disk pages. The OMNI$^+$-file allows many less pages to be read and the only additional cost it incurs is one random disk access is required to jump to the start of the OMNI-coordinates of the objects not yet pruned for each pivot object. This is more efficient than reading the entire OMNI$^+$-file because there are many pages in the OMNI$^+$-file which will not have to be read. In the example in Figure 3.7(b), a random disk access would be required to jump from DP_5 to DP_8 and from DP_9 to DP_{13}. Since the number of pivot objects is small, this cost is negligible. Hence, the OSEQ$^+$ algorithm potentially (and typically, as our experiments show) avoids a full scan of the OMNI$^+$-file and still performs all but a few I/Os sequentially. In Figure 3.7(c) we can see a physical view of the OMNI$^+$-file. On disk the OMNI-coordinates pertaining to p_1 occur sequentially followed by the OMNI-coordinates of p_2 and p_3. The OSEQ$^+$ in the example reads the sets of pages $DP_1 - DP_5, DP_8 - DP_9$ and $DP_{13} - DP_{14}$.

3.2.3 Sorting the Data File

The data file for the OSEQ$^+$ method is also sorted by each object's distance to $\mathcal{P}$ and therefore is in the same order as the OMNI$^+$-file. Sorting the data file improves the performance of the OSEQ$^+$ considerably by reducing the number of disk pages accessed and not using random disk I/Os to access the data file. The reason this results in reducing the cost of accessing the candidates' feature vectors is the candidates occur closer together in the data file and do not occur randomly. This allows the objects to be read sequentially from disk as opposed to using more expensive random disk I/Os for each object. In the example in Figure 3.7(b), objects O_{12}, O_{15}, and O_7 cannot be pruned by any pivot object and therefore the actual feature vectors of these objects have to be retrieved from disk. The OSEQ method will require one random disk access to retrieve the feature vectors of O_{12}, O_{15} and O_7, whereas the OSEQ$^+$ would read O_{12}, O_{15}, O_1 and O_7 sequentially from disk. Even while some objects that are already pruned will be read, i.e., O_1, and not be needed, it is more efficient to do so than to force random disk accesses for each object not pruned because sequential disk I/Os are much less expensive than random disk I/Os. Unless the number of objects not pruned is extremely small, the OSEQ method will not be efficient.

3.2.4 Building and Searching - OSeq$^+$ Method

The OSEQ$^+$ build algorithm (Figure 3.9), builds the OMNI$^+$-file offline and the pivots can be chosen using any pivot selection technique such as the HF algorithm, the Incremental Selection algorithm or using PCA in vector spaces. $\mathcal{P}$ (p_1) is selected as the most important pivot object, i.e., the closest pivot on average to every object in the database for the HF algorithm or for PCA and the Incremental Selection algorithm, the first pivot selected by the algorithm. The OMNI$^+$-file and the data file are sorted by each object's distance to $\mathcal{P}$. As mentioned previously, one of the main advantages of the OSEQ$^+$ method compared to the OSEQ method is it uses sequential disk I/Os to access the OMNI-coordinates and to access the data objects in the refinement step. This is possible because the objects are sorted by their distance to $\mathcal{P}$, therefore the objects occur close together in the data file and can be read sequentially.

The search algorithm for the OSEQ$^+$ (Figure 3.10) is similar to the OSEQ search algorithm. Its main advantages are the format of the OMNI$^+$-file and the data file, and how both files are accessed. The search algorithm computes the distance from each pivot object to the query object and calculates $d_{min}(p_j)$

```
OSeq$^+$ Build Algorithm: Building the OMNI$^+$-file
Input  : The data set $s_i \in S$, the number of pivots $k$
Output: Pivot set $p_j \in P$, the OMNI$^+$-file, the sorted data file

Select pivot objects with a pivot selection algorithm
Select $p_1$ as the most important pivot depending on the pivot selection algorithm
foreach $p_j$ do
    foreach $s_i$ do
        compute $d(p_j, s_i)$ and store in OMNI$^+$-file
Sort the OMNI$^+$-file by each object's distance to $p_1$
Sort the data file by each object's distance to $p_1$
```

Figure 3.9: OSEQ$^+$ Build Algorithm

and $d_{max}(p_j)$ for each p_j. The algorithm prunes by $\mathcal{P}$ and sequentially reads the OMNI-coordinates pertaining to $\mathcal{P}$. It should be noted that objects can also be pruned by $\mathcal{P}$ using a B$^+$-tree to find $d_{min}(\mathcal{P})$ and $d_{max}(\mathcal{P})$. All the objects not pruned by $\mathcal{P}$ occur adjacent to each other in the OMNI$^+$-file because the OMNI$^+$-file is sorted by the objects' distances to $\mathcal{P}$. This allows the algorithm to jump to the first candidate object not pruned by previous pivots with one random disk access for each pivot and read the OMNI-coordinates of the objects not pruned by previous pivots sequentially. This may lead to reading some OMNI-coordinates of objects already pruned, but it is more efficient to use sequential rather than random disk I/Os and read a few unnecessary OMNI-coordinates. After all the objects have been pruned by each pivot, the actual distance is computed between each candidate object and the query object. The feature vectors of the candidate objects are read sequentially because the data file is sorted. One random disk access is required to find the feature vector of the first object not pruned.

The OSEQ$^+$ method performs all but k ($k << D$) of its disk I/Os sequentially to avoid expensive random disk accesses. The number of random I/Os is equal to the number of pivot objects used, which is much less than the size of the candidate set, making the OSEQ$^+$ method more efficient than the OSEQ method. As can be seen in Figure 3.7(c), the data pages are read sequentially,

```
OSeq^+ Search Algorithm: Find all answers to a given range query
Input  : The data set $s_i \in S$ sorted by $\mathcal{P}$, Set of pivots $p_j \in P$, OMNI^+-file
         sorted by $\mathcal{P}$, query object $q$ and query radius $r$
Output: Answer Set $A$ to the range query

Initialize Candidate Set $C$ to contain all $s_i \in S$
foreach $p_j \in P$ do
    calculate $d(p_j, q)$
    $d_{min}(p_j) = d(p_j, q) - r$
    $d_{max}(p_j) = d(p_j, q) + r$
foreach $p_j \in P$ do
    foreach $s_i \in C$ do
        if $d(p_j, s_i) < d_{min}(p_j)$ or $d(p_j, s_i) > d_{max}(p_j)$ then
            remove $s_i$ from $C$
foreach $s_i \in C$ do
    calculate $d(s_i, q)$
    if $d(s_i, q) \leq r$ then
        place $s_i$ in Answer set $A$
return A
```

Figure 3.10: OSEQ$^+$ Search Algorithm

only one random disk access is required to find the OMNI-coordinate of the first object which has not been pruned for each remaining pivot object. This makes the OSEQ$^+$ method very fast and yet simple to implement.

3.3 Using Only a Few Good Pivots – OSEQ*

The OSEQ$^+$ method chooses $\mathcal{P}$ such that the first pivot on average has higher pruning power than the other pivot objects. However, the OSEQ$^+$ method does not address the problem that the order the pivots are selected by the HF algorithm and the Incremental Selection algorithm does not indicate they will be equally good for all queries, i.e., reading their OMNI-coordinates is not justified by their pruning ability for every query object. A solution to this problem is to select more pivots than are needed for pruning (oversampling the pivot set) and at query time only choose a small number of pivots, those which are closest to the query object. As mentioned in Section 3.2, by using more pivot objects closer to the query object, the pivot objects used better approximate the position of the query object. Choosing pivot objects closer to the query object also results in reducing the hyper-volume of the intersection of the pruning rings of the selected pivot objects. This allows more objects to be pruned and results in less disk accesses and less distance calculations needing to be computed to answer similarity queries. In Figure 3.11 we can see the advantage of having pivot objects closer to the query object. The area of the intersection of the pruning rings of pivot objects A and B is much smaller than that of pivot objects C and D. To illustrate this, consider again Figure 3.7, and assume that in addition to $\mathcal{P} = p_1$, only one additional pivot is used. Since p_3 is closer to q than p_2, p_3 is used and all pages related to p_2 $(DP_6 - DP_{10})$ will never be accessed, reducing even further the number of pages accessed. The choice of better pivot objects per query reduces the candidate set size and increases the efficiency of processing similarity queries, this is confirmed by our experimental results. If we again consider Figure 3.4, we can see the pivot objects selected with the HF algorithm which are closer on average to other objects in the database have higher pruning power. For PCA, the pivots are created artificially on the principal axes of the data set outside the data space. Therefore the OSEQ* method will be the same as the OSEQ$^+$ method when selecting pivots using PCA because the notion of closeness does not apply because as explained in Section 2.2, all pivots are created the same distance outside the data space. For the Incremental Selection algorithm, the same result occurs as with the HF algorithm. In Figure 3.12, we can see the pivot objects selected with ISA that are on average closer to other objects in the data set have higher pruning power. The OSEQ* attempts to select the pivots with higher pruning power to be used to prune the candidate set for each query. The additional cost of this method compared to the OSEQ method and the OSEQ$^+$ method is that there is an overhead of extra disk space to store the OMNI-coordinates of the extra pivots. However, no additional cost is incurred at query time because the same number of pivots or less are used for pruning as with the OSEQ$^+$ algorithm, the efficiency is actually increased because pivots with higher pruning power are used for pruning.

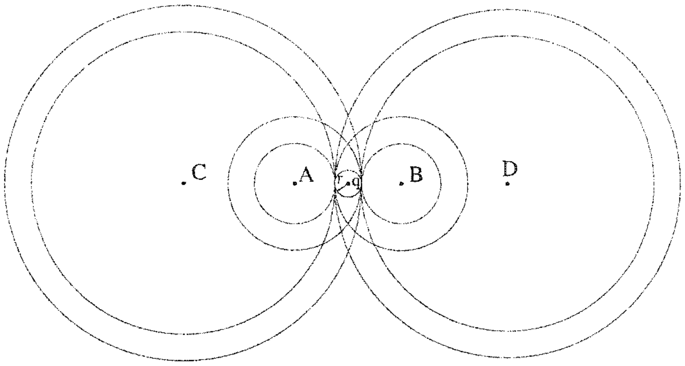


Figure 3.11: The effect of having pivots closer to the query object

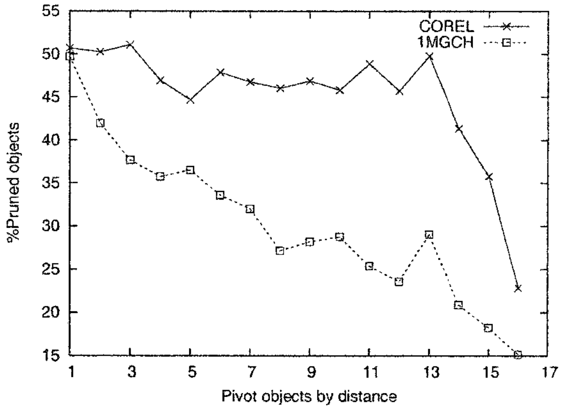


Figure 3.12: Pruning power of pivots selected with the Incremental Selection Algorithm and the pivots are sorted by the pivots' average distance to all objects in the database

The build algorithm for the OSEQ* method is exactly the same as for the OSEQ$^+$ method. The difference is the build algorithm for the OSEQ* method selects more pivots than will be used at query time and oversamples the pivot set, while the OSEQ$^+$ build algorithm selects the same number of pivots as the search algorithm uses for similarity search.

As with the OSEQ$^+$ Search algorithm, the OSEQ* Search algorithm (Figure 3.13) requires that the OMNI$^+$-file is presorted by $\mathcal{P}$, where $\mathcal{P}$ is still chosen to be the pivot which is most important. Unlike the OSEQ$^+$ method, the OSEQ* method oversamples the pivot set and selects more pivots than it will use at query time (K pivots). At query time, the algorithm computes the distance between the query object and each of the K pivot objects. Besides

$\mathcal{P}$, the OSEQ* search algorithm selects the $k-1$ pivots ($k-1 < K$) which are closest to the query object. Note that unlike the case for $\mathcal{P}$, these pivots are query-dependent, i.e., they are chosen based on the query object. The remainder of the OSEQ* Search algorithm is similar to the OSEQ$^+$ Search algorithm once the $k-1$ pivot objects are selected.

```
OSeq* Search Algorithm: Find all answers to a given range query
Input  : The data set $s_i \in S$ sorted by $\mathcal{P}$, set of pivots $p_j \in P$ ($|P| = K$,
         oversampled), OMNI$^+$-file sorted by $\mathcal{P}$, query object $q$, query
         radius $r$, number of pivots to use $k$
Output: Answer Set $A$ to the range query

Initialize Candidate Set $C$ to contain all $s_i \in S$
foreach $p_j \in P$ do
    calculate $d(p_j, q)$
Choose the $k-1$ closest $p_j \in \{P - \mathcal{P}\}$ to $q$ and place in $PC$ ($pc_j \in PC$)
Sort $PC$ by distance to $q$ and place $\mathcal{P}$ in $PC$ as $pc_1$
foreach $pc_j \in PC$ do
    $d_{min}(pc_j) = d(pc_j, q) - r$
    $d_{max}(pc_j) = d(pc_j, q) + r$
Find all $s_i \in S$ such that $d_{min}(\mathcal{P}) \leq d(\mathcal{P}, s_i) \leq d_{max}(\mathcal{P})$, place $s_i$ which
satisfies the above inequality in Candidate Set $C$
foreach $pc_j \in PC$ do
    foreach $s_i \in C$ do
        if $d(pc_j, s_i) < d_{min}(pc_j)$ or $d(pc_j, s_i) > d_{max}(pc_j)$ then
            remove $s_i$ from $C$
foreach $s_i \in C$ do
    calculate $d(s_i, q)$
    if $d(s_i, q) \leq r$ then
        place $s_i$ in Answer set $A$
return A
```

Figure 3.13: OSEQ* Search Algorithm

3.4 Experimental Results

We test the efficiency of the OSEQ$^+$ and the OSEQ* methods using the same two data sets described in the previous section. The 1MGCH data set is created to test the scalability of our algorithms on large data sets. We select a sample of 100 random query objects from each of the data sets to use in our experiments. Our measure of efficiency is the speedup in terms of I/Os and simulated time[2] required to process a range query. To calculate the simulated time, we perform experiments using the facilities offered by our experimental environment[3] to obtain the cost (in seconds) of distance calculations, comparisons, absolute values, sequential disk I/Os and random disk I/Os. The results are shown in Table 3.2. Although others have used the number of distance calculations as a measure of query cost [9, 16, 19], we choose time because it combines the total cost of answering similarity queries, i.e., random and sequential disk accesses, distance calculations, absolute values and comparisons.

Table 3.2: Costs

1 64D distance calculation	655.4 *nsecs*
1 comparison	1.3 *nsecs*
1 absolute value	1.3 *nsecs*
1 sequential disk I/O	105,300 *nsecs*
1 random disk I/O	1,852,000 *nsecs*

We use the L_1 metric distance because it has been shown to be more effective in high-dimensional spaces than the Euclidean distance (L_2 norm) [1]. As opposed to NN-queries, a range query cannot guarantee an answer size, thus, for our experiments, we fix r so that an average answer size between 15 and 45 is obtained. The page size is set to 4Kb.

Initially we designed an experiment to verify our intuition about the existence of a preferential pivot ($\mathcal{P}$) with higher pruning capability for pivots selected by the HF algorithm, Incremental Selection algorithm and PCA. The

[2] Due to caching management issues, we were unable to accurately measure real time, therefore we report simulated time

[3] 1600 MHz PC, AMD Athlon processor, Seagate ST360021A disk, RedHat Linux 9 and gcc-3.4.0

results for these experiments are detailed in Section 3.2.1 and provide evidence that it is beneficial to explore the notion of a preferential pivot, and that such a pivot should indeed be the one which is closest on average to every other object in the data set if there is no order of importance imposed by the pivot selection algorithm (HF algorithm). If pivots are selected with an order of importance (ISA and PCA), then $\mathcal{P}$ should be chosen as the most important pivot. Recall that $\mathcal{P}$ can be found a priori yielding no overhead at query time.

Figures 3.14 and 3.15 show the performance of the OSEQ$^+$ method and the OSEQ* method compared to the OSEQ method and a sequential scan (SeqScan) of the data set. The HF algorithm is used to select the pivot objects for the OSEQ, OSEQ$^+$ and OSEQ* methods. It should be noted that the pivot objects used by the OSEQ$^+$ method are the first k pivot objects selected by the HF algorithm, $\mathcal{P}$ is chosen from these k pivots. For the OSEQ* method, 16 pivot objects are always selected by the HF algorithm, this makes the OMNI$^+$-file 25% of the size of the original data file. From these 16 pivot objects, $\mathcal{P}$ is selected and in addition to $\mathcal{P}$, the $k-1$ pivot objects closest to the query object are also selected for each query. In Figures 3.14(a) and (b) the OSEQ$^+$ method processes range queries faster than the OSEQ and a sequential scan (SeqScan) of the data set (curves SeqScan/OSEQ$^+$ and OSEQ/OSEQ$^+$). The curve SeqScan/OSEQ$^+$ represents the number of disk accesses of the SeqScan divided by the total number of I/Os of the OSEQ$^+$ to obtain the speedup of the OSEQ$^+$ compared to the SeqScan. The OSEQ* method requires the least disk accesses and query processing time. The speedup of the OSEQ* method compared to the OSEQ$^+$ is on top of the savings the OSEQ$^+$ already achieves over the OSEQ and SeqScan. This is expected as both methods use the preferential pivot $\mathcal{P}$, the OMNI$^+$-file and sort the data file by $\mathcal{P}$. This means fewer OMNI-coordinates have to be accessed for the OSEQ$^+$ and OSEQ* compared to the OSEQ method and the actual feature vectors of candidate objects occur closer together in the data file. This results in accessing up to 60% and 80% fewer data pages from disk than the OSEQ for the OSEQ$^+$ and OSEQ* methods respectively. In Figure 3.15 we can see the time for the OSEQ$^+$ method and OSEQ* method compared to the OSEQ and SeqScan to process range queries. The actual number of disk accesses for all methods follows a similar curve because each of these methods are I/O bound.

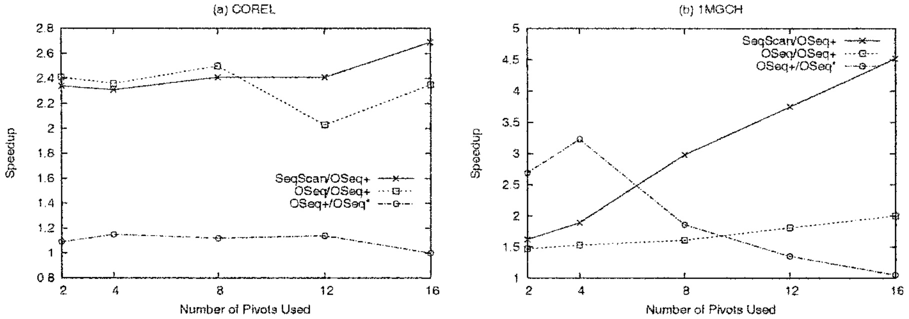


Figure 3.14: The speedup in terms of the number of disk accesses of the proposed sequential metric access methods

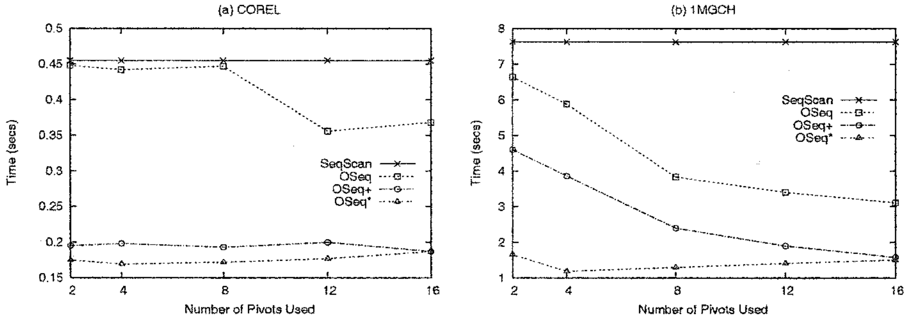


Figure 3.15: The query performance of the proposed sequential metric access methods in terms of simulated time

All the OMNI-based methods use k pivots at query time. The difference is the OSEQ and OSEQ$^+$ methods use the first k pivot objects initially selected by the HF algorithm, whereas the OSEQ* selects K pivots (in our experiments $K = 16$) with the HF algorithm and in addition to $\mathcal{P}$, at query time it chooses the $k-1$ pivots ($k-1 < K$) which are closest to the query object. We can see in Figure 3.14 that the OSEQ* accesses fewer disk pages as the closer pivot objects prune more candidates. When more pivot objects are used, the methods are using a larger common set of pivots, this is why the curves approach each other as k increases in Figure 3.15. At $k = 16$, both the OSEQ$^+$ and the OSEQ* methods are using the same set of pivots. In Figure 3.15, the curve for the OSEQ* method rises slightly as more pivot objects are used ($k > 4$) because reading the OMNI-coordinates of additional pivot objects is not justified by the number of additional data objects which are pruned. The initial closest pivot objects for the OSEQ* method prune so many objects that the last pivot objects can not prune any further. This observation will also occur for the OSEQ method and OSEQ$^+$ method if a greater number of pivot objects are used.

For the case of the Euclidean space, we want to test the effectiveness of selecting pivots using PCA compared to the HF algorithm and ISA. Figure 3.16(b) shows that the OSEQ$^+$ saves up to 27 times the number of disk accesses by selecting pivots with PCA compared to using pivots selected with the HF algorithm and saves up to 20 times the number disk I/Os compared to using pivots selected with the Incremental Selection algorithm. The savings in time are displayed in Figure 3.17. Pivots selected with PCA prune more objects as pivot objects are selected to lie on the principal axes of the data set and are orthogonal to each other as shown by the authors in [10]. We do not show results for the OSEQ* method using PCA to choose the pivot objects because the pivot objects selected with PCA are all created the same distance outside the data set so the notion of closest does not apply. Figures 3.16 and 3.17 also show the pivot objects selected by the Incremental Selection algorithm are better than the pivots selected with the HF algorithm. The OSEQ$^+$ using ISA to select the pivots is up to 1.6 times faster. This provides evidence that for general metric spaces it will be beneficial to use the Incremental Selection algorithm compared to the HF algorithm.

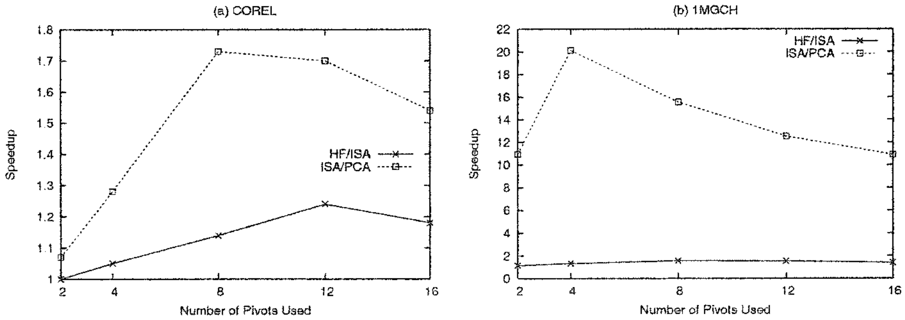


Figure 3.16: The speedup in terms of the number of disk accesses of the OSEQ$^+$ method using different pivot selection algorithms

Next we compare the OSEQ$^+$ method using PCA to select the pivots

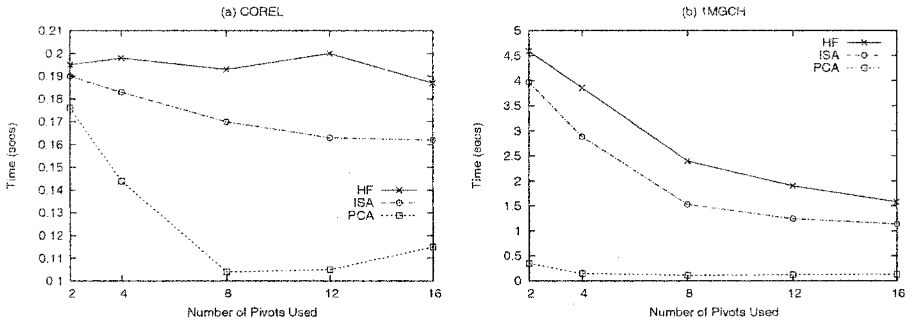


Figure 3.17: The elapsed time of the OSEQ$^+$ method using different pivot selection algorithms

(OSEQ$^+$ (PCA)), the OSEQ* method using ISA to select the pivots (OSEQ* (ISA)) and the VA-File in Euclidean space. Figure 3.18 shows the speedup of the OSEQ$^+$ (PCA) method compared to the SeqScan, the VA-File and the OSEQ* (ISA) in terms of I/Os and Figure 3.19 shows the time required to answer range queries for the OSEQ$^+$ (PCA) method, the OSEQ* (ISA), the VA-File method and the sequential scan. The OSEQ$^+$ (PCA) method is up to 24 times faster than the VA-File method in terms of time. While it is true that more efficient implementations of absolute values, comparisons and distance calculations would benefit the VA-File more than the OSEQ$^+$ (PCA) method, Figure 3.18 shows that the OSEQ$^+$ (PCA) method requires fewer disk accesses for both data sets when the number of pivots is greater than 8. This means that no matter how much faster the implementation, the VA-File will still be less efficient than the OSEQ$^+$ (PCA) method because both methods use sequential access patterns and avoid random disk I/Os, while the VA-File requires a substantial amount of CPU time. In Figure 3.18 we can also see the OSEQ* (ISA) and the VA-File have similar performances in terms of the number of disk accesses. In terms of time (Figure 3.19), the OSEQ* (ISA) is faster than the VA-File for both data sets because of the large amount of CPU time required for the VA-File. Figures 3.18 and 3.19 also show that in Euclidean space, the OSEQ$^+$ (PCA) is more efficient than the OSEQ* (ISA). The OSEQ* (ISA) outperforms the OSEQ$^+$ (ISA), we can see the speedups in Figure 3.16 are larger for the OSEQ$^+$ (PCA) compared to the OSEQ$^+$ (ISA) then in Figure 3.18 for the OSEQ$^+$ (PCA) compared to the OSEQ* (ISA).

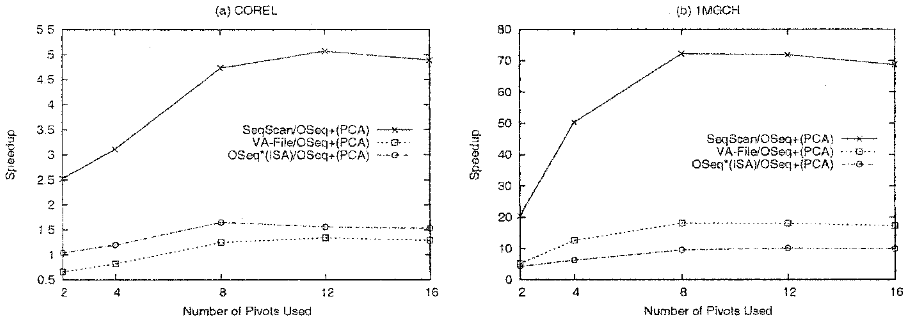


Figure 3.18: The performance of the SeqScan, the VA-File and the OSEQ* (ISA) in terms of disk accesses compared to the OSEQ$^+$ (PCA)

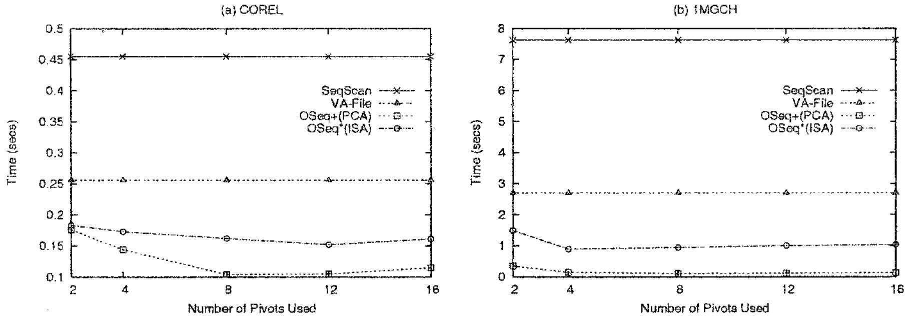


Figure 3.19: The performance of the SeqScan, the VA-File, the OSEQ* (ISA) and the OSEQ$^+$ (PCA) in terms of time (secs)

It is interesting to note that even though the sequential scan of the data set accesses approximately four times more disk pages than the VA-File, the VA-File is not even twice as fast in terms of time as the sequential scan for the COREL data set and only about 2.8 times faster for the 1MGCH data set. This supports our argument that high-dimensional index structures must be efficient not only with respect to disk accesses but CPU operations as well.

The number of random disk I/Os increases rapidly for the VA-File as the radius of the range query increases. In Figures 3.20(a) and (b), we can see that the time required for the VA-File to process range queries increases quickly due to the increase in the number of random disk I/Os in the refinement step. This occurs because as the radius increases, the number of objects returned and the number of objects which can not be pruned based on their approximations for the VA-File increases, while the VA-File requires one random disk access for each candidate object in the refinement step. For this reason we do not show the speedup in terms of disk accesses for these experiments. The comparison in terms of the number of disk accesses is unfair because the VA-File is no longer primarily a sequential access method as the radius of a range query increases. In contrast, the OSEQ$^+$ (PCA) method uses sequential disk I/Os in the refinement step. Therefore, as the radius of the range query increases, the time required for the OSEQ$^+$ (PCA) method to process range queries increases at a much smaller rate.

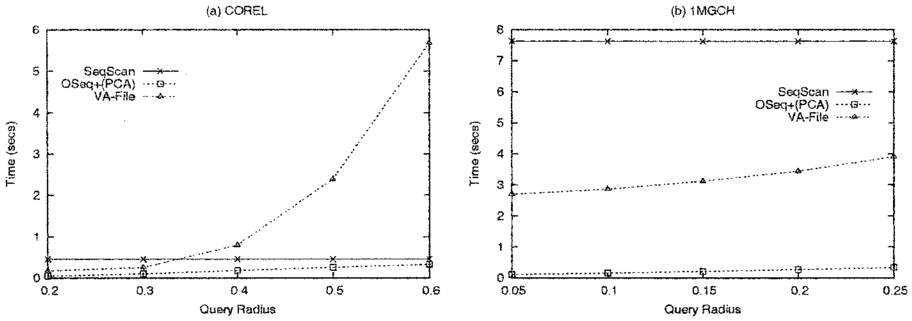


Figure 3.20: The performance of the access methods in terms of time (secs) as the radius of the range query increases

Chapter 4

The Metric Grid

It is a known result, e.g., [4, 11], that when processing similarity queries in a general metric space, objects can be pruned using the triangular inequality property. All one needs is a set of pivots, the distance from all objects in the database to the pivots and a query range. That is, the actual distance from the query object to the objects in the data set need not be computed for a potentially large set of pruned objects. Given that the actual distance can be computationally expensive, this is an important gain.

The VA-File partitions the data space by dividing it into a grid of cells. This allows objects to be approximated by the cells where they lie. These approximations require less storage than the original feature vectors of the objects. One drawback of the VA-File is it can only be applied to vector data spaces because each dimension is divided into slices to form a grid. For general metric data, the only information available is the distances between objects, there may be no concept of dimensions, thus we cannot partition each dimension into slices to form cells. Nevertheless, the idea of creating a pseudo-grid can be generalized to a metric data space. An actual grid cannot be created in all metric spaces because there may be no actual dimensions, but a grid-like structure can be constructed. Instead of dividing each dimension into slices, pivots in the data set can be selected and rings can be formed based on the distances from the pivots. In Figure 4.1 the pivot objects are p_1 and p_2 and the rings are created based on the radii distances to the pivots.

For the VA-File, each object is approximated by the cell in which it lies,

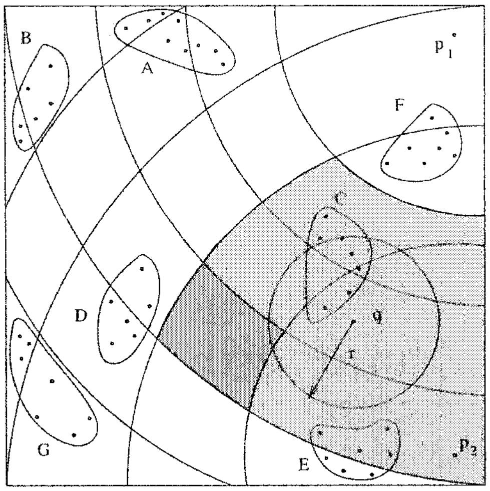


Figure 4.1: The rings splitting up the data space into a grid-like structure

the size of these approximations is smaller than the original feature vectors. Objects cannot be approximated for general metric data because there are often no feature vectors for objects, but the object's position on a pseudo-grid can be approximated by its distance to each of the pivots. In effect, the data set is mapped into a low-dimensional vector space and the objects can be clustered based on their positions in the grid. The intuition is that objects close together in the original data space will also be close together on the grid in the low-dimensional mapped space. Using this notion, we propose to cluster the data objects by their distances to the pivots. Any clustering algorithm can be used because the distances to the pivots are numerical (vectorial). Clustering high-dimensional data and metric data is typically a complex and expensive task. Using a small number of distances to the pivots to cluster the data objects makes the clustering much more efficient.

Objects can be pruned using their distances to pivots and the triangular inequality property. Further, we can think of clusters as objects. Clusters can be pruned in the same manner as objects using the triangular inequality property, that is, clusters that do not touch pruning rings can be safely discarded. In Figure 4.1, for query q and radius r, the shaded cells intersect the query region, i.e., the region defined by the circle around q. By employing the notion of a grid, a cluster can be pruned if the cluster does not intersect any cells which also intersect the query region. Only clusters C and E intersect cells intersecting the query region and therefore clusters A, B, D, F and G can be pruned from the answer set without computing any of the actual distances between objects in the cluster and the query object.

Given this rationale we can now propose a new access method for both vector data (under a metric distance) and general metric data. We call this method the M-Grid. The M-Grid is a dynamic, metric access structure and is designed to reduce the processing time of similarity search queries for high-dimensional data sets and metric data sets, in particular for data sets that present some inherent clustered structure. The M-Grid clusters objects by their positions in a pseudo-grid based on their distances to the pivot objects. The pivots are selected from the data set because it is often not possible to artificially create pivots for metric data sets. For vector space however, we can create artificial pivots with higher pruning power using PCA to select pivots on the major axis of the data set, e.g., [10]. Cells in the grid are determined by the distance the rings are from the pivots (Figure 4.1) which allows a cluster to be pruned if the cluster does not occupy any cells that intersect the query region. In order to take advantage of sequential disk accesses, the M-Grid performs most of its disk accesses sequentially within the clusters, being an I/O efficient metric access method. In addition the M-Grid is computationally inexpensive as the computation of the actual, and potentially expensive, distance functions are avoided for most objects.

4.1 Building the M-Grid

The algorithm to build the M-Grid is shown in Figure 4.3. To begin building the M-Grid, the pivot objects must first be selected. Pivots can be selected in a variety of different ways for metric data, i.e., selecting objects randomly from the data set, selecting objects with the HF algorithm [16], using the Incremental Selection algorithm [12], etc. We have found (Section 3.4) that a very effective way to select pivots for general metric spaces is with the Incremental Selection algorithm.

The Incremental Selection algorithm selects pivots that best distinguish the distances between pairs of randomly chosen objects from the data set. The intuition behind this is if the pivots can distinguish well the distances between arbitrarily selected objects in the data set, they will better be able to distinguish the distances between the query object and the objects in the data set. Being able to distinguish the distance between the query object and objects in the data set is important because only the triangular inequality property can be used to prune objects in metric spaces which are non-vectorial. If the distance between the query object and the pivot is the same as an object's distance to the pivot, the object will not be able to be pruned from the answer set.

The distance from each object in the data set to all the pivots must be computed. The rings are selected so there is an equal number of objects in each ring for each pivot, this allows more dense regions of the data space to be covered by more rings and increases the number of objects pruned during similarity search. The intersection of the rings form cells in the low-dimensional mapped space, a pseudo-grid for metric data (see Figure 4.1 for a 2-D example).

The objects are clustered using the K-means clustering algorithm [18] for simplicity. It is important to note however, that this is not a requirement. Any clustering algorithm for vector data can be used because the distances from objects to pivots are numerical (vector), hence forming a feature vector. Since the number of pivots tends to be small, the data set can be clustered efficiently. Objects are clustered based on their distances to pivots so objects occurring in the same rings will be placed in the same clusters. The clusters closest to the cell containing the query object are retrieved during similarity search and the clusters which do not intersect the query region are not accessed.

The main structure of the Metric Grid consists of three levels. The first level of the M-Grid contains the pivots, the distances the rings are from each pivot and the mean of each cluster. The number of pivots, k, tends to be small, in the range of δ to 2δ where δ is equal to the fractal (intrinsic) dimensionality of the data set [16]. As a direct consequence the data set can be clustered

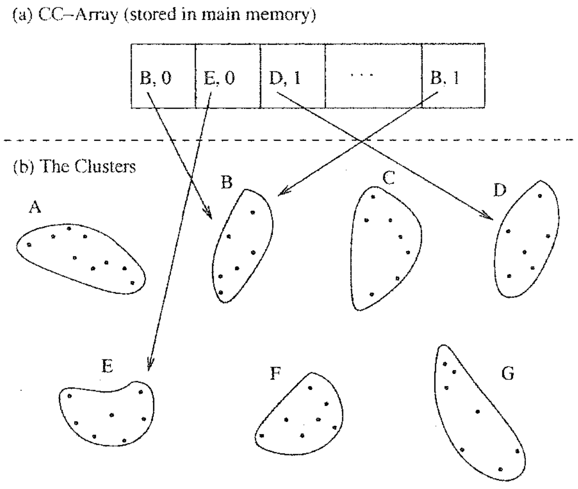


Figure 4.2: The storage of the M-Grid on disk (a) CC-Array (b) The clusters are stored contiguously on disk

efficiently. The data objects of the pivots, the distances to the rings from the pivots and the mean of each cluster on the metric grid can be stored in a small number of disk pages. Since the storage space for this level is small, it is assumed that it can be stored in main memory.

The second level consists of the Cluster Cell-Array (CC-Array) (Figure 4.2). Each cell is represented in the CC-Array by a structure $(Cell[].ptr, Cell[].O_{bit})$ where $Cell[].ptr$ is a pointer to the cluster the cell belongs to and $Cell[].O_{bit}$ is one bit which indicates if the cell is empty or contains objects (Figure 4.2(a)). This level is also assumed to fit in main memory. In Section 4.2 we show good performance is obtained using four pivots and ten rings per pivot which is equivalent to 20 Kb, which can be easily stored in main memory.

The closest cluster to each cell is determined by computing the distance from the center of each cell to the mean of each cluster. The structure for empty cells also point to the nearest clusters because the first cluster to visit is determined by the cell the query object is in, even if the cell is empty. Also, the pointers are used for the insertion and deletion algorithms to determine the cluster to insert or delete the object. In Figure 4.2(a) we can see the structures in the CC-Array contain pointers to one cluster and one bit indicating if the cell is empty. In the example, Cell 1 contains zero objects but points to cluster B, while the $Cell[].O_{bit}$ for Cell 3 is equal to 1, indicating Cell 3 contains at least one object and the pointer for the cell points to cluster D. Storing whether each cell contains any objects (or is empty), is useful to prevent visiting clusters based on a cell that does not contain any objects but intersects the query region and points to a cluster that would not otherwise be visited. In Figure 4.1, the dark-shaded cell is empty but intersects the query region and its pointer will point to Cluster D (its nearest cluster). By using 1 bit for the cell to indicate the cell is empty, we can avoid visiting Cluster D in this example. It should be noted that each cell can only point to one cluster, i.e., the cluster it is closest to in the metric grid. While a cell can only belong to one cluster, several different cells can point to the same cluster, there is not just one cluster stored for each cell in the grid. The actual number of objects in each cell is stored separately on disk and is only used to aid inserting and deleting objects from the M-Grid.

The third level of the M-Grid contains the original objects in the data set. The data for each object in the same cluster is stored sequentially on disk. This is to allow clusters to be accessed using sequential disk accesses during the NN search algorithm.

The pseudo-code for the algorithm to build the M-Grid is shown in Figure 4.3.

4.1.1 Similarity Search in the M-Grid

The K-Nearest Neighbor Search algorithm for the M-Grid visits clusters in order depending on their distance to the query's position on the metric grid with the closest clusters being visited first. A good initial radius can be found quickly visiting only a few clusters as the closest clusters usually contain the closest objects to the query object. This current nearest neighbor radius is then used to prune clusters which are farther from the query object and do not intersect the query region.

The K-Nearest Neighbor search algorithm for the M-Grid is shown in Figure 4.4. The first step is to compute the distance between the query object and each pivot. The cell containing the query object (*q_cell*) is determined by calculating which ring for each pivot that the query object belongs to by its

```
Build Algorithm: Constructing the M-Grid
Input  : The number of pivots, the data set of objects ($s_i \in S$), the number of rings $no\_rings$
Output: The set of pivots $p_j \in P$, the ring distances, the clusters, the cell pointers, the number of objects in each cell $num\_per\_cell[]$

Select the set of pivots ($p_j \in P$)
foreach $s_i \in S$ do
    foreach $p_j \in P$ do
        compute $d(s_i, p_j)$
Determine the ring distances for each pivot such that each ring contains the same number of objects
Cluster the objects using the cells in the grid to represent the objects
Compute the mean of each cluster
foreach cell do
    foreach cluster ($cl_i \in C$) do
        compute $d(cell, cl_i)$
        Set cell pointer to nearest cluster
    Count the number of objects in each cell
```

Figure 4.3: The algorithm to build the M-Grid

distance to each of the pivots. The pointer for the cell containing the query object is accessed (in the CC-Array) to determine the first cluster to visit. The algorithm calls the Visit_Cluster() function (Figure 4.5) and sequentially retrieves all the objects in the first cluster. The distance between the query object and each object in the cluster is computed. Using the distance from the query object to the Kth nearest object found so far (KNN_dist[K]), the rings intersecting the query region are identified. Any clusters not intersecting cells which intersect the query region, are pruned and never visited.

After the first cluster is visited and the clusters are pruned using KNN_dist[K], the K-NN search algorithm computes the distance from *q_cell* to the mean of each cluster which has not been pruned or visited and stores the distances in the Cluster Distances Array (*CDA*). The *CDA* is sorted in ascending order and the clusterIDs of each cluster are placed and sorted in the same order in the Active Cluster List (*Acl*) as the *CDA*. The first cluster in the *Acl* which

```
K-NN Search Algorithm: Searching the M-Grid
Input  : The set of pivots ($p_j \in P$), the ring distances, the query object
         $q$, the clusters, the cluster means $cm_i \in CM$, the cell pointers,
         the number of nearest neighbors $K$
Output: The K-NN objects to $q$

Active Cluster List ($Acl$) = [Null]
Cluster Distances Array ($CDA$) = [Null]
i=1
while $i \leq K$ do
    KNN_dist[i] = $\infty$
    i++
foreach $p_j \in P$ do
    compute $d(q, p_j)$
Calculate the cell the query object lies in (q_cell)
Visit_Cluster(The cluster containing q_cell, KNN_dist[ ], KNN[ ], K) // see Figure 4.5
Prune clusters not intersecting the query region using the KNN_dist[K] and the CC-Array
foreach Cluster not pruned or visited do
    $CDA[i] = d(q\_cell, cm_i)$
Sort the $CDA$ in ascending order
Place the clusterIDs of each cluster in the $Acl$ and sort the $Acl$ in the same order as the $CDA$
curr_KNN_dist = KNN_dist[K]
i = 1
while $i \leq$ number of clusters do
    if $Acl[i]$ is not pruned then
        Visit_Cluster($Acl[i]$, KNN_dist[ ], KNN[ ], K) // see Figure 4.5
        if KNN_dist[K] < curr_KNN_dist then
            curr_KNN_dist = KNN_dist[K]
            Prune clusters not intersecting the query region using the
            KNN_dist[K] and the CC-Array
    i++
```

Figure 4.4: The K-Nearest Neighbor Search algorithm for the M-Grid

```
Visit_Cluster Function: Visit the cluster and compute the distance between the query object and every object in the cluster
Input   : Cluster of objects $s_i \in C$, KNN_dist[ ], KNN[ ], K

foreach $s_i \in C$ do
    if $d(q, s_i) < KNN\_dist[K]$ then
        KNN[K] = $s_i$
        KNN_dist[K] = $d(q, s_i)$
        Place KNN[ ] and KNN_dist[ ] in ascending order of KNN_dist[ ]
```

Figure 4.5: Visit_Cluster Function

has not been pruned is then visited. Like the first cluster visited, the distance between the query object and all objects in the cluster are computed. If a closer object is found than KNN[K], the closer object is placed in KNN[K] and its distance is placed in KNN_dist[K]. KNN[ ] and KNN_dist[ ] are then placed in ascending order by KNN_dist[ ] and clusters are pruned again by determining the cells intersecting the new query region defined by the new KNN_dist[K]. This process continues until every cluster has been either visited or pruned. Note that once a cluster has been pruned, it never has to be considered again because for K-NN queries, the query region can only get smaller. The search algorithm continues to visit clusters in the *Acl* until every cluster has been either pruned or visited to guarantee the exact answer.

The correct answer to each similarity query is guaranteed by the properties of the grid. The triangular inequality can be used to prune objects which are far away from the query object using precomputed distances to pivots. Clusters can be considered large objects and the triangular inequality can be used to prune clusters on the M-Grid. Since we know the cells which intersect (belong to) each cluster from the CC-Array, we only need to identify the cells which intersect the query region. The pointers for these cells can be looked up in the CC-Array and all clusters these cells point to are identified. This allows us to prune only clusters which are outside the cells which intersect the query region and guarantee the correctness of the answer. As mentioned earlier, only non-empty cells are used to identify clusters intersecting the query region. Using empty cells to determine the clusters to visit can lead to visiting additional clusters which do not contain any objects which also intersect the query region. The pointers for empty cells are used to find the first cluster to visit, e.g., if the query object lies in an empty cell. It is also useful to have the pointers for empty cells stored when new objects are inserted into or deleted from the M-Grid, this way we know which cluster to access for the object.

The M-Grid reduces the number of disk accesses of feature vectors of objects and keeps the number of random disk accesses small. Clusters are stored sequentially on disk, so the objects in each cluster are accessed sequentially. Only one random disk access is needed to find the beginning of the cluster on disk, therefore the number of random disk accesses is equal to the number of clusters visited. The M-Grid is designed for K-nearest neighbor queries but can be easily modified to process range queries. To process range queries, the algorithm can be changed so the K-NN radius is equal to the radius of the range query and unlike K-NN queries, the radius is fixed. All clusters intersecting the radius of the range query will be retrieved and all objects less than the radius will be returned as the answer to the range query.

4.1.2 Inserting and Deleting Objects

As mentioned earlier, the M-Grid is a dynamic indexing structure. The algorithm for inserting objects into the M-Grid is shown in Figure 4.6. To insert an object into the M-Grid, the cell containing the object $(Cell[O_{insert}])$ must be identified by its distances to each of the pivots. The cluster to insert the object is determined by looking up in the CC-Array the cluster $Cell[O_{insert}]$ belongs to. If the object bit for the cell containing the object $(Cell[O_{insert}].O_{bit})$ is equal to 0, then the bit must be set to 1 to indicate the cell contains at least one object. The number of objects in the cell is also incremented by 1. The object is inserted into the closest cluster to $Cell[O_{insert}]$ and the mean of the cluster is recomputed.

Objects can be easily deleted from the M-Grid by removing the object from its cluster. Similar to an insert, the cell containing the object to be deleted (O_{delete}) is identified and the cell is looked up in the CC-Array to determine the cluster containing O_{delete}. The number of objects in the cell is decremented by

```
M-Grid Insertion Algorithm: Inserting Objects into the M-Grid
Input  : The set of pivots $p_j \in P$, insertion object $O_{insert}$, the ring distances, the cluster means $cm_i \in CM$, the clusters, the cell pointers, object bits of cells $Cell[].O_{bit}$
Output: Object inserted into M-Grid

foreach $p_j \in P$ do
    compute $d(O_{insert}, p_j)$

Determine the cell ($Cell[O_{insert}]$) containing $O_{insert}$
Find the cluster $Cell[O_{insert}]$ belongs to
Insert $O_{insert}$ into this cluster
if $num\_per\_cell[Cell[O_{insert}]] = 0$ then
    Set $Cell[O_{insert}].O_{bit}$ to 1

$num\_per\_cell[Cell[O_{insert}]] ++$
Recompute mean of the cluster
```

Figure 4.6: M-Grid Insertion Algorithm

1 and if no other objects reside in the cell, i.e., $num\_per\_cell[Cell[O_{delete}]] = 0$, $Cell[O_{delete}].O_{bit}$ is set to 0. The mean of the cluster is then recomputed.

4.2 Experimental Results

We tested the performance of the M-Grid using a variety of synthetic data sets. The data sets are designed to test the scalability of the M-Grid with respect to varying the cardinality ($nobj$) of the data set, the number of clusters (no_cl), the percentage of noise ($noise$), the number of dimensions (dim), the maximum distance (max_d) of objects in each cluster to the seed of the cluster, the number of pivots (no_pivots) and the number of rings (no_rings) used in the M-Grid and the number of nearest neighbors (K) retrieved during similarity search. The default values for the data sets are shown in Table 4.1.

We assigned objects to clusters non-uniformly. It should be noted the M-Grid is designed for data sets which exhibit some clustered structure. Some clusters in the data set contain more objects than others to simulate real data sets. An example of the distribution of how many objects are in each of the 100 clusters is shown in Figure 4.7. Note that changing the number of objects

Table 4.1: Default values of the data sets

nobj	250,000
no_cl	100
noise	20%
dim	64
max_d	1% of dim
no_pivots	4
no_rings	10
K	10

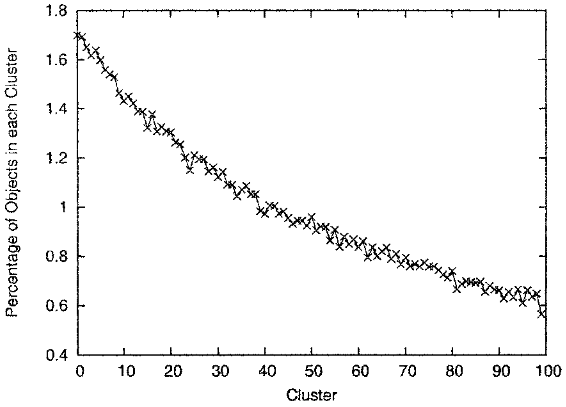


Figure 4.7: An example distribution for the number of objects in each cluster. The average number of objects in each cluster is 2500 or 1% of the total number of objects when using the default values shown in Table 4.1. The distance objects are from the seeds of the clusters follows the same distribution

or the number of clusters in the data set does not affect the distribution of the number of objects in each cluster, it just changes the average number of objects in each of the clusters. The distance the objects are from the seeds of the clusters also follows the distribution in Figure 4.7. We use the L_1 metric distance in our experiments and all data sets are generated in the unit cube. The default value for the maximum distance from the seeds of the clusters is set to 1% of the maximum possible distance objects can be apart in the data set.

For each data set, we generate a sample of 100 queries the same way we generate objects in the data sets to use in our experiments. Our measure of efficiency is the number of disk accesses and the simulated time[1] required to process a K-nearest neighbor query and the page size for all experiments was set to 4KB. To evaluate the efficiency of the M-Grid, we compare it to the VA-File and a sequential scan (SeqScan) of the data set. For each experiment we show the results in a graph where the number of disk accesses for the sequential scan and the VA-File are divided by the number of disk accesses of the M-Grid to show the speedup of the M-Grid, as well, the simulated time is also shown. Using the number of disk accesses as an efficiency measure is fair because the VA-File and the M-Grid both use sequential access patterns and only a few random disk I/Os. For all experiments, 10 data sets are generated the same way (following the same distribution) and the average number of disk accesses is used for the M-Grid and the VA-File. For the sequential scan, the number of disk accesses is constant for each experiment.

The first experiment is designed to test the scalability of the M-Grid while varying the cardinality of the data set. In Figure 4.8(a) we can see the speedup of the M-Grid compared to a sequential scan of the data set is stable at approximately 20 with a slight increase in performance of the M-Grid as the number of objects in the data set is increased. This occurs because as the number of objects in the data set is increased and the number of clusters remains the same, there are more objects in each cluster. This results in a slight increase in performance of the M-Grid because the K-NN distance is reduced slightly for most queries, resulting in accessing slightly fewer clusters. For the same reason, the speedup of the M-Grid compared to the VA-File increases slightly to greater than 5. Figure 4.8(b) indicates the same results. The time for all three access methods to answer K-NN queries increases linearly with increasing the number of objects in the data set.

The next set of experiments test the performance of the M-Grid while varying the number of clusters. As expected, increasing the number of clusters improves the performance of the M-Grid. When decreasing the number of clusters, the number of objects per cluster increases and if any part of the cluster intersects the query region, the entire cluster is retrieved from disk.

[1] We use the same cost values as shown in Table 3.2 to compute the simulated time

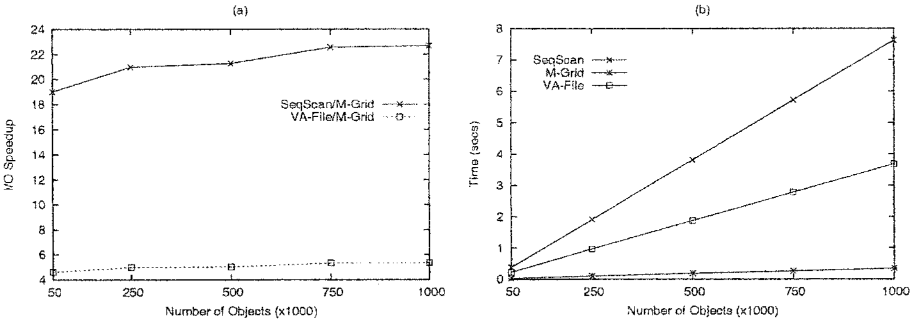


Figure 4.8: The effect of varying the number of objects in the data set

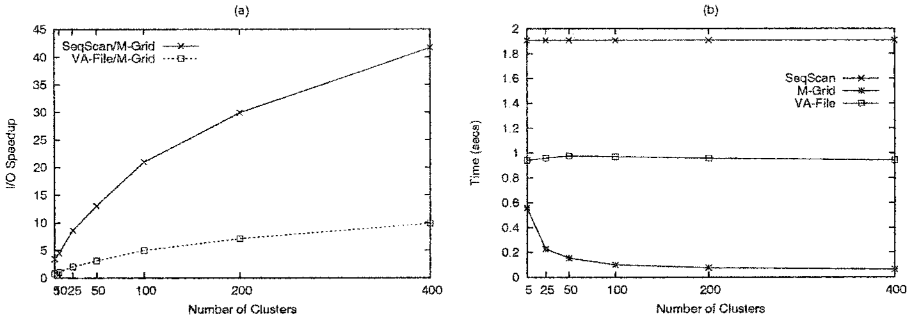


Figure 4.9: The effect of varying the number of clusters in the data set

Dividing a cluster into two smaller clusters can mean retrieving only half the objects if only one of the two clusters now intersects the query region. Dividing clusters into smaller clusters is only beneficial to a certain extent. At some point the same objects will still need to be retrieved, just more clusters (with fewer objects per cluster) will be visited. Figure 4.9(a) shows the M-Grid is more than 40 times faster than a sequential scan of the data set and up to 10 times faster in terms of disk accesses than the VA-File when the number of clusters the M-Grid divides the data set into is increased to 400. In Figure 4.9(b) we can see the performance of the VA-File and the SeqScan remains constant while increasing the number of clusters in the data set, while the performance of the M-Grid increases. In Figure 4.9(b) the M-Grid is up to 15 times faster in terms of time because the M-Grid requires far fewer CPU operations. Note that even when the number of clusters is very small, the M-Grid is still faster than the VA-File.

Noise in the data set are objects which are created randomly in the data set and are not part of any cluster. By using noise, one can investigate the

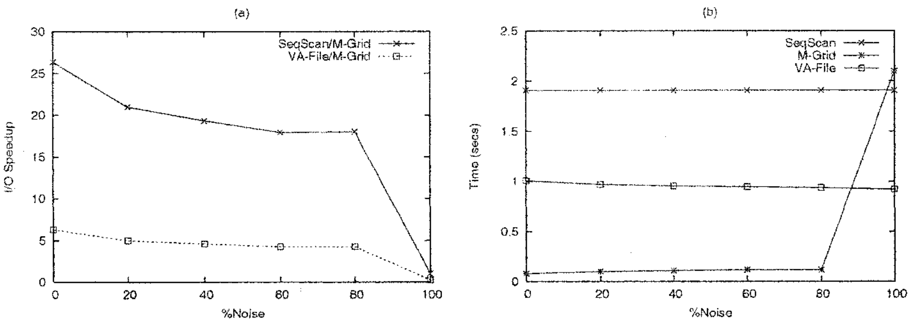


Figure 4.10: The effect of varying the percentage of noise in the data set

resiliency of an access structure. Figures 4.10(a) and (b) show the M-Grid is resistant to increasing the percentage of noise in the data set and the performance of the M-Grid only decreases slightly when the amount of noise in the data set is increased to 80% of the total number of objects in the data set. This means the M-Grid can perform nearest neighbor search efficiently as long as the data set contains some objects which are clustered. At 100% of the data set being noise, the distribution of the data set is uniform as all the objects in the data set are generated randomly. Since the M-Grid is designed for data which has some sort of clustered structure, it cannot improve on the performance yielded by a sequential scan, and it is outperformed by the VA-File. This occurs because the VA-File approximates the actual distance of every object in the data set and actually performs the best for uniform data sets because the objects in the dataset are farther from each other. This reduces the likelihood of objects occurring in the same cell for the VA-File and thus having the same approximations. For a reasonable amount of noise, say 50%, the M-Grid is about 5 times faster than the VA-File and 20 times faster than the sequential scan respectively.

The performance of the M-Grid was also tested against varying the dimensionality of the data set while keeping the remaining parameters of the data set constant. The results for varying the dimensionality between 32 and 512 are shown in Figure 4.11. The maximum distance (max_d) objects can be from the seeds of the clusters also remains constant at 1%. However, when the dimensionality of the data set is increased, 1% of the maximum possible distance between two objects will also increase. Figure 4.11(a) shows the M-

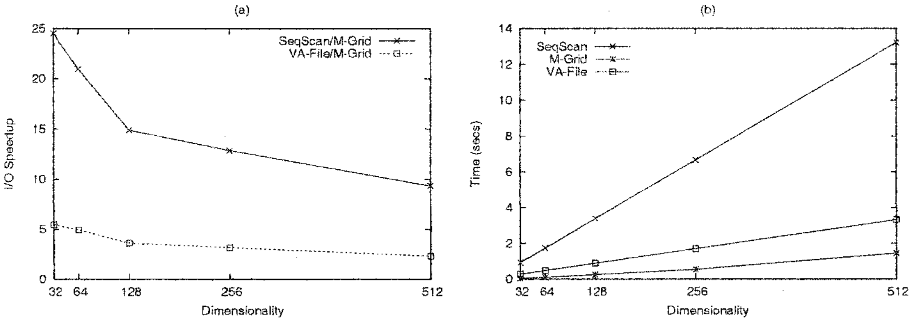


Figure 4.11: The effect of varying the dimensionality of the data set

Grid is more than three times faster than the VA-File when the number of dimensions is 256. For dimensionalities of 32 and 64, the M-Grid is five times faster than the VA-File in terms of disk accesses. The relative performance of the M-Grid decreases slightly as the dimensionality increases compared to a sequential scan of the data set and the VA-File. One reason for this is the maximum distance objects in clusters can be by default is 1%. Therefore the size of the clusters increases when increasing the dimensionality and the total hyper-volume of the clusters increases faster than the maximum distance. Although the total hypervolume of the data space also increases, each dimension is still in the range, zero to one. This causes the chance of clusters overlapping to be higher and increases the chance of visiting additional clusters.

Figure 4.12 shows the performance of the M-Grid decreases as *max_d* increases. The main reason for this is as *max_d* increases, the distribution of the objects in the data set becomes more uniform. This causes the K-nearest neighbor distance to increase significantly, reducing the ability of the pivots to prune clusters. It should be noted that if the data set can be clustered more effectively, the M-Grid would be able to prune more clusters and increase its efficiency. Even when *max_d* reaches 10%, the VA-File and the M-Grid exhibit the same performance and the M-Grid is still a little more than two times faster than a sequential scan of the data set.

The M-Grid has two main input parameters for building the metric grid, the number of pivots and the number of rings. Both of these parameters affect the performance of the M-Grid. Figure 4.13 shows the affect of varying the number of pivots. Increasing the number of pivots results in more clus-

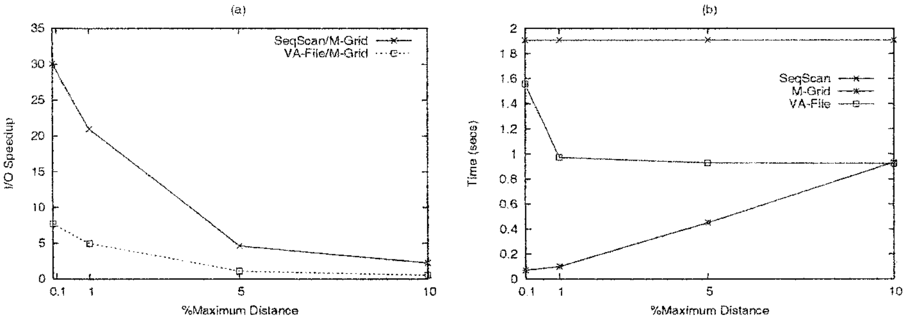


Figure 4.12: The effect of varying the maximum distance objects in a cluster can be from the seed of the cluster

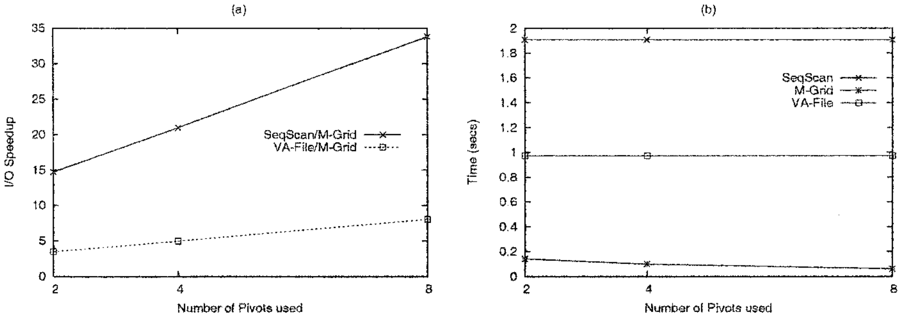


Figure 4.13: The effect of varying the number of pivots used by the M-Grid

ters being pruned. This occurs because the more pivots we use, the greater the probability that clusters will not intersect rings which the query region intersects, for at least one pivot. This results in visiting less clusters and retrieving less objects from disk which reduces the query processing time for the M-Grid (Figure 4.13(b)). Increasing the number of pivots used by the M-Grid can only increase the performance up to a certain extent, at some point using more pivots will not result in pruning more clusters. This occurs because there is always going to be at least one cluster for every query which must be visited and clusters which intersect the query region can not be pruned no matter how many pivots we use. The speedup of the M-Grid compared to a sequential scan of the original data set increases to 33 and to 8 for the VA-File when the number of pivots is equal to 8 (Figure 4.13(a)).

Figure 4.14 shows increasing the number of rings also increases the performance of the M-Grid compared to a sequential scan of the data set and the VA-File. Increasing the number of rings divides the data space into a finer grid which makes approximating objects and clusters by the cells they inter-

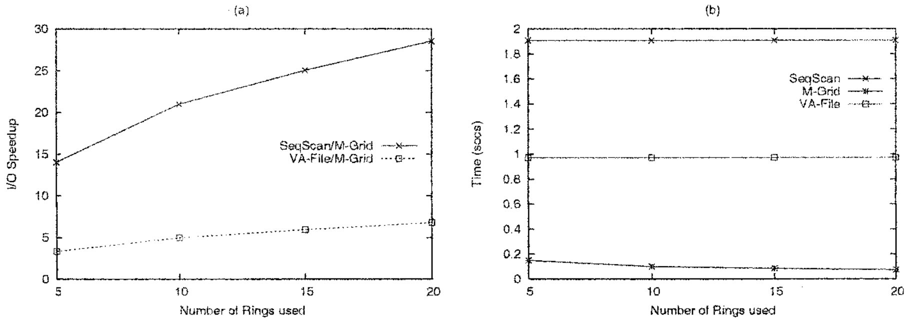


Figure 4.14: The effect of varying the number of rings used by the M-Grid

sect more accurate. This allows some clusters which just border the query region to be pruned and not be visited. Similar to increasing the number of pivots, increasing the number of rings also increases the number of clusters pruned and improves the efficiency of the M-Grid. As noted above for increasing the number of pivots, for every data set there will be some limit at which increasing the number of rings can not result in pruning more clusters as some clusters will have to be visited anyway.

Next, we test the effect of varying the number of nearest neighbors K retrieved. Figure 4.15(a) shows the largest advantage for the M-Grid is achieved when $K = 1$. The reason for this is for small values of K, only 1 or 2 clusters have to be retrieved (or $1\% - 2\%$ of the data set) for the M-Grid. Increasing the number of nearest neighbors retrieved past 10 has very little impact on the performance of the M-Grid because often the additional nearest neighbors retrieved are found in the clusters already accessed and additional clusters do not have to be visited. The K-nearest neighbor distance usually does not increase rapidly when K is larger than 10. This causes the number of disk accesses to increase slowly with increasing K. It should be noted that as K is increased, the additional disk accesses for the VA-File are random disk accesses. This is why in Figure 4.15(b) the curve for the VA-File is increasing at a faster rate than the M-Grid, while in Figure 4.15(a) it appears the performance of the M-Grid is decreasing compared to the VA-File as K is increased. As K is increased, the total number of disk accesses for the VA-File and the M-Grid is comparable, but because these additional disk accesses for the VA-File occur in the refinement step and are random disk I/Os, the performance of the M-

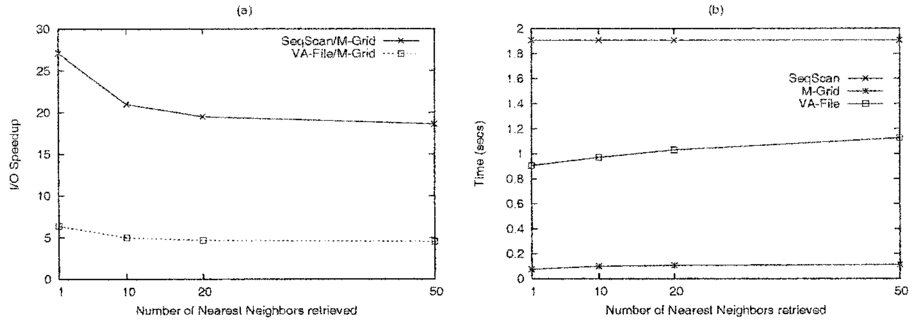


Figure 4.15: The effect of varying the number of nearest neighbors retrieved by the M-Grid

Grid increases compared to the VA-File in actual query processing time. This means the actual time required for the VA-File to perform K-NN search will increase rapidly as K increases, even if the total number of disk accesses does not increase as quickly.

Chapter 5

Conclusion

5.1 Summary and Contributions

In this thesis we have examined the challenges of creating efficient access methods for high-dimensional data and general metric data. We have proposed three new access structures for efficient similarity search. The first two structures, the OSEQ$^+$ method and the OSEQ* method, speedup range query processing with a number of improvements over the OSEQ method. The proposed methods restructure the OMNI-file and employ the notion of a preferential pivot $\mathcal{P}$ to be used as the first pivot to prune objects. The OSEQ$^+$ and OSEQ* methods also sort the OMNI$^+$-file and the data file resulting in substantial savings by allowing the two files to be accessed sequentially and also reduce the total number of disk accesses. In addition, we have shown that it is beneficial to oversample the pivot set when the order the pivots are selected does not indicate their pruning ability, and then at query time, choose a small portion, i.e., those which are closest to the query object.

Our experimental results show that with often less than 25% storage overhead, the OSEQ* method can process similarity queries up to five times faster than the OSEQ for metric data. The OSEQ$^+$ and OSEQ* methods work in high-dimensional vector spaces as well as general metric spaces. We have shown in vector spaces, the OSEQ$^+$ method using the PCA pivot selection technique, clearly outperforms the VA-File by a large margin when processing range queries, up to 24 times faster and is up to 69 times faster than a sequential scan of the data set.

The third access method we have proposed is the M-Grid. We have shown how to construct a pseudo-grid structure for any metric space, including Euclidean space, and use the metric grid to cluster the data objects using any clustering algorithm, even if the original data is not vectorial. The M-Grid performs efficient similarity search in high-dimensional spaces and general metric spaces by searching the clusters sequentially while the properties of the metric grid guarantee the correctness of the answer.

Our experimental results show the M-Grid can perform nearest neighbor search up to 40 times faster than a sequential scan of the data set and up to 15 times faster than the VA-File. In addition, our method works for all metric spaces, not just vector spaces, can be easily implemented and scales well with increasing the number of nearest neighbors retrieved. We have further verified that it is critical to not only limit the number of disk accesses, but to perform most I/O sequentially and to reduce the number of CPU operations to create an efficient access method.

5.2 Future Research

This thesis explores the challenges of creating an efficient access method for both high-dimensional data and general metric data. Some interesting research issues for future work include:

- determining an optimal oversampling rate and the number of pivots to use at query time for the OSEQ* algorithm.
- Modifying the OSEQ$^+$ and OSEQ* algorithms to process nearest neighbor queries, specifically, a good method is needed to estimate the nearest neighbor distance.
- Determining an optimal algorithm for selecting $\mathcal{P}$ independent of the pivot selection algorithm for the OSEQ$^+$ and OSEQ* methods.
- Exploring the effectiveness and efficiency of the clustering using the metric grid and comparing it to other state-of-the-art clustering algorithms.

- Optimally selecting the number of rings and pivots to use for each data set when building the metric grid.

- Determining new ways to store the CC-Array so the number of rings and pivots can be increased without increasing the storage of the CC-Array.

- Investigating if the distribution of the data set changes through insertions and deletions, how will this affect the performance of the M-Grid and how can we detect this problem.

Bibliography

[1] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *Proc. of the 8th Intl. Conf. on Database Theory*, volume 1973, pages 420–434, 2001.

[2] R. Agrawal, C. Faloutsos, and A.N. Swami. Efficient Similarity Search In Sequence Databases. In *Proc. of the 4th Intl. Conf. of Foundations of Data Organization and Algorithms*, pages 69–84, 1993.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering in high dimensional data for data mining applications. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 94–105, 1998.

[4] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. of the 5th Symposium on Combinatorial Pattern Matching*, pages 198–212, 1994.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B.Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM 1990 SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.

[6] S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proc. of the 16th IEEE Intl. Conf. on Data Engineering*, pages 577–588, 2000.

[7] S. Berchtold, D.A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. of the 22nd Intl. Conf. on Very Large Databases*, pages 28–39, 1996.

[8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proc. of the 7th Intl. Conf. on Database Theory*, volume 1540, pages 217–235, 1999.

[9] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 357–368, 1997.

[10] B. Braunmuller, M. Ester, H.-P. Kriegel, and J. Sander. Multiple similarity queries: A basic DBMS operation for mining in metric databases. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):79–95, 2001.

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

[12] B. Bustos, G. Navarro, and E. Chavez. Pivot selection techniques for proximity searching in metric spaces. In *Pattern Recognition Letters*, volume 24, pages 2357–2366, 2003.

[13] E. Chavez, J. L. Marroquin, and G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, pages 57–64, 1999.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of 23rd Intl. Conf. on Very Large Data Bases*, pages 426–435, 1997.

[15] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 2nd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[16] R.F. Santos Filho, A. Traina, C. Traina Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *Proc. of the 17th IEEE Intl. Conf. on Data Engineering*, pages 623–630, 2001.

[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.

[18] J. Han and M. Kamber. *Data Mining: concepts and techniques.* Morgan Kaufmann Publishers, 2001. Page 349–351.

[19] C. Traina Jr., A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization for metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260, 2002.

[20] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 369–380, 1997.

[21] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clindex: Approximate similarity queries in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[22] L. Liao and W. S. Noble. Combining pairwise sequence similarity and support vector machines for remote protein homology detection. In *Proc. of the 6th Intl. Conf. on Research in Computational Molecular Biology*, pages 225–232, 2002.

[23] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional nearest queries. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[24] G. Lu. *Multimedia Database Management Systems.* Artech House, 1999.

[25] M.A. Mico and Oncina J. A new version of the nearest-neighbor approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. In *Pattern Recognition Letters*, volume 15, pages 9–17, 1994.

[26] B.-U. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. 16th Intl. Conf. on Data Engineering*, pages 589–598, 2000.

[27] David Patterson. A conversation with Jim Gray. *ACM Queue tomorrow's computing today*, 1(4), June 2003. `http://www.acmqueue.org`.

[28] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *Intl. Journal of Computer Vision*, 40(2):99–121, 2000.

[29] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 516–526, 2000.

[30] R.O. Stehling, M.A. Nascimento, and A. X. Falcão. MiCRoM: A metric distance to compare segmented images. In *Proc. of the 2002 Visual Information Systems Conf.*, pages 12–23, 2002.

[31] R.O. Stehling, M.A. Nascimento, and A.X. Falcão. Techniques for color-based image retrieval. In C. Djeraba, editor, *Multimedia Mining - A Highway to Intelligent Multimedia Documents*, chapter 4. Kluwer Academics, 2002.

[32] M.J. Swain and D.H. Ballard. Color indexing. *Intl. Journal of Computer Vision*, 7(1):11–32, 1991.

[33] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Inf. Process. Lett*, pages 175–179, 1991.

[34] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of the 24th Intl. Conf. on Very Large Databases*, pages 194–205, 1998.

[35] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of IEEE 12th Intl. Conf. on Data Engineering*, pages 516–523, 1996.