MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Indexing Structures for Searching in Metric Spaces

PH.D. THESIS

Vlastislav Dohnal

Brno, February 2004

Acknowledgement

I would like to thank my supervisor Pavel Zezula for guidance, insight and patience during this research. I would also acknowledge my colleagues Claudio Gennaro and Pasquale Savino for their contributions and comments on this work. I also thank my parents, my wife and my colleagues for their patience, emotional support and encouragement throughout the entire process. Above all else, I would like to thank Queen for composing very inspiring music.

Abstract

This Ph.D. thesis concerns the problem of indexing techniques for searching in metric spaces. We propose two novel index structures for similarity queries, we have compared them with other approaches and executed numerous experiments to verify their properties.

In order to speedup retrieval in large data collections, index structures partition the data into subsets so that query requests can be evaluated without examining the entire collection. As the complexity of modern data types grows, metric spaces have become a popular paradigm for similarity retrieval. We propose a new index structure, called D-Index, that combines a novel *clustering* technique and the *pivot-based* distance searching strategy to speed up execution of similarity range and nearest neighbor queries for large files with objects stored in disk memories. We have qualitatively analyzed D-Index and verified its properties on actual implementation. We have also compared D-Index with other index structures and demonstrated its superiority on several real-life data sets. Contrary to tree organizations, the D-Index structure is suitable for dynamic environments with a high rate of delete/insert operations.

In the next part, we analyze the problem of similarity join which are often applied in many areas, such as data cleaning or copy detection. Though this approach is not able to eliminate the intrinsic quadratic complexity of similarity joins, significant performance improvements are confirmed by experiments.

Supervisor: prof. Ing. Pavel Zezula, CSc.

Keywords

metric space

metric data

similarity search

index structures

range query

nearest neighbor query

similarity join

performance evaluation

D-Index

eD-Index

Contents

1 Introduction **11**

2 Background **15**

2.1 Metric Spaces . . . 15
2.2 Distance Functions . . . 17
Minkowski Distances . . . 17
Quadratic Form Distance . . . 18
Edit Distance . . . 18
Tree Edit Distance . . . 19
Hausdorff Distance . . . 20
Jacard Coefficient . . . 21
Time Complexity . . . 21
2.3 Similarity Queries . . . 21
Range Query . . . 22
Nearest Neighbor Query . . . 23
Reverse Nearest Neighbor Query . . . 23
Similarity Join Query . . . 24
Combinations . . . 25
2.4 Search Strategies . . . 25
2.5 Partitioning Algorithms . . . 31
2.6 Filtering Algorithms . . . 33
2.7 Choosing Pivots . . . 35

3 Metric Index Structures **37**

3.1 Ball Partitioning Methods . . . 37
3.1.1 Burkhard-Keller Tree . . . 37
3.1.2 Fixed Queries Tree . . . 39
3.1.3 Fixed Queries Array . . . 41
3.1.4 Vantage Point Tree . . . 42
3.1.5 Multi Vantage Point Tree . . . 46

3.1.6 Excluded Middle Vantage Point Forest 47
3.2 Generalized Hyperplane Partitioning Methods 50
3.2.1 Bisector Tree 50
3.2.2 Generalized Hyperplane Tree 51
3.2.3 GNAT 52
3.3 M-Tree 55
3.3.1 Slim Trees 59
3.4 Spatial Approximation Tree 60
3.5 Distance Matrix Methods 63
3.5.1 AESA 64
3.5.2 LAESA 65
3.5.3 Other Methods 66
3.6 Mapping-Based Approaches 67

4 D-Index 69
4.1 Clustering Through Separable Partitioning 69
4.2 Pivot-Based Filtering 72
4.3 Illustration of the idea 72
4.4 System Structure and Algorithms 76
4.4.1 Storage Architecture 76
4.4.2 Insertion and Search Strategies 78
4.4.3 Generic Search Algorithms 80
4.4.4 Range Query Algorithm 83
4.4.5 k-Nearest Neighbors Search 84
4.4.6 Choosing references 86
4.4.7 Implementation of D-Index 90
4.4.8 Properties of D-Index 91

5 D-Index Evaluation 93
5.1 Data Sets 93
5.2 D-Index Performance Evaluation 95
5.3 D-Index Structure Design 97
5.4 D-Index Performance Comparison 100
5.4.1 Search efficiency for different data sets 101
5.4.2 Search, space, and insertion scalability 107

6 eD-Index **111**
6.1 Algorithm Categories . . . 112
6.1.1 Complexity . . . 112
6.1.2 Filter Based Algorithms . . . 113
6.1.3 Partition Based Algorithms . . . 114
6.1.4 Range Query Algorithms . . . 115
6.2 Implementation and Performance Evaluation . . . 116
6.2.1 Sliding Window Algorithm . . . 116
6.2.2 Pivot Based Filtering . . . 117
Dynamic Pivots . . . 118
6.2.3 Comparison . . . 119
6.3 Range Query Join Algorithm . . . 121
6.4 eD-Index Access Structure . . . 123
6.4.1 Range Query Algorithm . . . 125
6.4.2 Nearest Neighbors Search . . . 126
6.4.3 Similarity Self Join Algorithm . . . 126

7 eD-Index Evaluation **131**
7.1 Performance Evaluation . . . 131
7.1.1 Join-cost ratio . . . 132
7.1.2 Scalability . . . 135

8 Conclusion **139**
8.1 Summary . . . 139
8.2 Contribution . . . 141
8.3 Research Directions . . . 143

Bibliography **145**

List of Figures

2.1 The sets of points at the same distance from the center point for different L_p distance measures. 18

2.2 (a) Range query $R(q, r)$ and (b) nearest neighbor query 3–$NN(q)$ where the objects o_1, o_3 are at the same distance equal to 3.3 and o_1 has been chosen as the 3^{rd} neighbor at random. Qualifying objects are filled. . . . 22

2.3 (a) Reverse nearest neighbor query 2–$RNN(q)$ and (b) similarity self join query $SJ(2.5)$. Qualifying objects are filled. 24

2.4 Illustration of Lemma 2.4.1: (a) the general case, (b) the lower bound, and (c) the upper bound. 26

2.5 Illustration of Lemma 2.4.2 with three different positions of the query object; above, below, and within the range $[r_l, r_h]$. We can bound $d(q, o)$, providing that $r_l \leq d(p, o) \leq r_h$ and $d(q, p)$. The dotted and dashed lines denote lower and upper bounds, respectively. . . 28

2.6 Illustration of Lemma 2.4.3, $d(p, o)$ is within the range $[r_l, r_h]$, and $d(q, p)$ is within $[r'_l, r'_h]$. The lower bound on $d(q, o)$ is expressed in (a), the upper bound on $d(q, o)$ is presented in (b). The lower and upper bounds are emphasized with dotted and dashed lines, respectively. 29

2.7 Illustration of Lemma 2.4.4. As the query object moves along the dashed equidistant line (i.e. to q') the lower bound shrinks. The lower bound $\frac{d(q,p_1)-d(q,p_2)}{2}$ is denoted by dotted arrow line. 30

2.8 Examples of ball partitioning. The basic bp function (a), more spherical cuts (b), and combination of two functions (c). 32

2.9 Illustration of hyperplane partitioning. The basic hp function (a) and general partitioning (b). The clustering principle is depicted in (c). 33
2.10 Illustration of filtering algorithm. The lower bound principle (a), both the upper bound (dotted circle) and the lower bound are applied (b), and filtering using two distinct pivots. 34

3.1 An example of a metric space and a range query (a). BKT tree (b) is built on the space. 39
3.2 Examples of FQT tree (a) and FHFQT tree (b) that are built on the given space in Figure 3.1(a). 40
3.3 Example of the FHFQT tree (a). FQA is built from FHFQT in (b). 41
3.4 An example of range queries, in (a), S_0 need not to be accessed during the search and both the set must be visited in (b). 44
3.5 An example of VPT with two pivots p_1 and p_2 in (a). The corresponding representation of the tree in (b). . . 45
3.6 The difference between VPT and MVPT structures, VPT uses three pivots for partitioning to four sets (a), MVPT collapses two levels of VPT into one node and uses two pivots only (b). 48
3.7 An example of bp_ρ function in figure (a), all filled points form the exclusion set S_2, which is then eliminated from the process of partitioning. VPF consisting of two trees is presented in (b), the second tree is built from all exclusion sets of the first tree. 49
3.8 Generalized Hyperplane Tree (GHT): (a) hyperplane partitioning and both the partitions must be accessed to answer the given range query, (b) the corresponding structure of the tree. 52
3.9 An example of partitioning used in the Geometric Near-neighbor Access Tree in (a) and the corresponding tree in (b). 53

3.10 Example of the pruning effect of ranges. The query $R(q_1, r_1)$ needs only the range $[r_l^{ij}, r_h^{ij}]$ to eliminate the subtree of p_j. As for the query $R(q_2, r_2)$, the range $[r_l^{ij}, r_h^{ij}]$ is not sufficient for skipping the subtree of p_j and the range $[r_l^{jj}, r_h^{jj}]$ is also used and p_j is then eliminated. 54
3.11 Example of M-Tree (MT) that consists of 3 levels. Above, the 2-D representation of partitioning where pivots are denoted with crosses and the circles around pivots correspond to values of covering radii. Observe that balls around pivots can intersect. The dotted circles represent the minimum values of covering radii. If they are used no overlap is achieved. Below, the corresponding tree structure is shown. 57
3.12 SAT tree is built over a dataset in (a). In (b), SAT tree is depicted where o_1 is selected as root. 62
3.13 A sample of SAT with the root t and the query object q are provided. At the current node p, the range search algorithm selects the object p as the closest object to q while s_2 is closer to q than p. An improved variant of search algorithm selects the closest object from all ancestors and their neighbors, i.e. s_2 is selected instead of p. The dashed lines represent the boundaries between Voronoi cell of the first level of SAT tree. The dotted lines depict the same for the second level. 64

4.1 The ball partitioning based on the excluded middle partitioning 74
4.2 Combination of two first-order ρ split functions applied on the two-dimensional space. 75
4.3 Example of pivots behavior. 76
4.4 Example of D-Index structure. 79
4.5 Example of use of the function G. 81
4.6 Different choices for pivot p to divide the unit square. . 87
4.7 Distance distribution for two pivots, one is the center while the other is a corner object, in the unit cube of the 20-dimensional Euclidean space. 88

5.1 Distance densities for VEC, URL, and STR. 94

5.2 D-Index organization parameters 95
5.3 Search efficiency for VEC in the number of distance computations and in the number of read blocks. 96
5.4 Search efficiency for URL in the number of distance computations and in the number of read blocks. 96
5.5 Search efficiency for TEXT in the number of distance computations and in the number of read blocks. 97
5.6 Nearest neighbors search on sentences using different D-Index structures. 98
5.7 Range search on sentences using different D-Index structures 99
5.8 Search efficiency versus the block size 100
5.9 Comparison of the range search efficiency in the number of distance computations for VEC, URL, and STR. . 102
5.10 Comparison of the range search efficiency in the number of block reads for VEC, URL, and STR. 103
5.11 Comparison of the nearest neighbor search efficiency in the number of distance computations for VEC, URL, and STR. 104
5.12 Comparison of the nearest neighbor search efficiency in the number of block reads for VEC, URL, and STR. . 105
5.13 Range search scalability. 107
5.14 Nearest neighbor search scalability. 108

6.1 Illustration of the Sliding Algorithm. 116
6.2 Example of pivots behavior. 118
6.3 Performance of the Pivot Based Filtering algorithm on the 10 and 20 dimensional data sets. 120
6.4 Performance of the Sliding Window algorithm on the 10 and 20 dimensional data sets. 120
6.5 The modified *bps* split function: (a) original ρ-split function; (b) modified ρ-split function. 123
6.6 The Sliding Window algorithm. 127
6.7 Coloring technique applied in the eD-Index. 129

7.1 Join queries on the text data set. 132
7.2 Join queries on vectors. 133
7.3 RJ algorithm scalability. 136
7.4 OJ algorithm scalability. 137

List of Tables

7.1 Join Speedup for text 134
7.2 Join Speedup for vectors 134
7.3 Join Speedup for text with building costs included . . . 135
7.4 Join Speedup for vectors with building costs included . 136

Chapter 1

Introduction

Searching has always been one of the most prominent data processing operations because of its useful purpose of delivering required information efficiently. However, exact match retrieval, typical for traditional databases, is not sufficient or feasible for data types of present digital age, that is image databases, text documents, audio and video collections, DNA or protein sequences, to name a few. What seems to be more useful, if not necessary, is to base the search paradigm on a form of *proximity*, *similarity*, or *dissimilarity* of a query and data objects. Roughly speaking, objects that are close to a given query object form the query response set. The similarity is often measured by a certain distance function. In this place, the notion of mathematical *metric space* [56] provides a useful abstraction for the nearness.

For illustration, consider the text data as the most common data type used in information retrieval. Since text is typically represented as a character string, pairs of strings can be compared and the *exact match* decided. However, the longer the strings are the less significant the exact match is: the text strings can contain errors of any kind and even the correct strings may have small differences. According to [60], text typically contains about 2% of typing and spelling errors. This gives a motivation for a search allowing errors, or *similarity search*.

Consider a database of sentences for which translations to other languages are known. When a sentence is to be translated, such a database can suggest a possible translation provided the sentence or its close approximation already exists in the database. For two strings of length n and m available in main memory, there are several dynamic programming algorithms to compute the edit distance of the strings in $O(nm)$ time. Refer to [68] for an excellent overview of the work and additional references.

In some applications, the metric space turns out to be of a particular type called vector space, where the elements consist of k coordinates (often termed *feature vectors*). For example, images are often represented by color and shape histograms [38, 48], typically consisting of 64 or 256 values. A lot of work[29, 6, 7, 47, 76, 10] have been done on vector spaces by exploiting their geometric properties. However, index and search techniques for vector spaces usually work well only for low-dimensional spaces. Even indexing methods specifically designed for higher dimensions (the X-tree [10], LSD^h-tree [50], and hybrid-tree [18]) typically stop being efficient, i.e. accessing nearly all objects, when the dimensionality exceeds twenty. Moreover, approaches to vector spaces cannot be extended to general metric spaces where the only available information is the distance among objects. In this general case, furthermore, the distance function is normally quite expensive to evaluate (e.g. the edit distance), thus, the general objective is to reduce the number of distance computations. On the contrary, the operations performed in vector spaces tend to be simple and hence the goal is to diminish I/O. Unfortunately, some applications have difficulties in transforming their problems to vector spaces. For this reason, several authors resort to general metric spaces.

Several storage structures, such as [87, 14, 25, 13], have been designed to support efficient similarity search execution over large collections of metric data where only a distance measure for pairs of objects is possible to quantify. However, the performance is still not satisfactory and further investigation is needed. Though all of the indexes are trees, many tree branches have to be traversed to solve a query, because similarity queries are basically range queries that typically cover data from several tree nodes or some other content-specific partitions. In principle, if many data partitions need to be accessed for a query region, they are not *contrasted* enough and such partitioning is not useful from the search point of view.

In general, there are two common principles to index a data set. The former is based on trees, where a tree structure is built and objects of the data set are accommodated either in leaves only or both in internal nodes and leaves. At the query time, the tree is traversed and the result set is constructed. The latter concerns direct access structures – hashing. This Ph.D. thesis concentrates on applying hashing principles in metric spaces. As the result, a novel access structure the D-Index has been developed.

The development of Internet services often requires an integration of heterogeneous sources of data. Such sources are typically unstructured whereas the intended services often require structured data. The main challenge is to provide consistent and error-free data, which implies the data cleaning, typically implemented by a sort of similarity join. In order to perform such tasks, similarity rules are specified to decide whether specific pieces of data may actually be the same things or not. However, when the database is large, the data cleaning can take a long time, so the processing time (or the performance) is the most critical factor that can only be reduced by means of convenient similarity search indexes. Since the similarity join queries are not efficiently implemented by existing metric index structures we have focused on this type of queries and extended the D-Index to support such queries, which is the second major contribution of this work.

This thesis is organized as follows. Chapter 2 embraces background and definitions of similarity searching in metric spaces. In Chapter 3, we provide a survey of existing index structures for metric spaces. We categorize methods into four categories: ball partitioning based, hyperplane partitioning based methods, distance matrix methods, and mapping-based approaches. In the next stage, we propose a novel access structure called D-Index that synergistically combines the clustering technique and the pivot-based distance searching methods into one system. It supports disk memories, thus, it is able to handle large archives. It also provides easy insertion and bounded search costs for range queries with radius up to ρ. Next, we experimentally evaluate properties of the D-Index and compare with other techniques which also supports a disk storage. In chapter 6, we focus on the problem of similarity joins and propose an extension of the D-Index, called eD-Index. Then, we provide a deep evaluation of the eD-Index. We conclude in Chapter 8 and outline the future research directions.

Chapter 2

Background

In this chapter, we provide the reader with necessities for understanding of similarity searching in metric spaces. We also present possible types of similarity queries used in information retrieval and give examples of such queries. Finally, we define selected metric distance functions.

2.1 Metric Spaces

A metric space $\mathcal{M}$ is defined as a pair $\mathcal{M} = (\mathcal{D}, d)$. The set $\mathcal{D}$ denotes the domain (universe) of valid objects (elements, points) of the metric space. A finite subset X of the domain $\mathcal{D}$, $X \subseteq \mathcal{D}$, is the set of objects where we search. X is usually called the *database*. The function

$$d : \mathcal{D} \times \mathcal{D} \mapsto \mathbb{R}$$

denotes a measure of distance between objects. The smaller the distance is the closer or more similar the objects are. The distance function is *metric* [56, 45] and satisfies following properties:

- $\forall x, y \in \mathcal{D}, d(x, y) \geq 0$
- $\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$
- $\forall x, y \in \mathcal{D}, x = y \Leftrightarrow d(x, y) = 0$
- $\forall x, y, z \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$

Some authors may call metric as *metric function*. There are some variations of the metric, which satisfy weaker or stronger conditions. Before we introduce them we decompose some properties of metric to simplify characterizations of metric variations:

(p1) $\forall x, y \in \mathcal{D}, d(x, y) \geq 0$	non-negativity,
(p2) $\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$	symmetry,
(p3) $\forall x \in \mathcal{D}, d(x, x) = 0$	reflexivity,
(p4) $\forall x, y \in \mathcal{D}, x \neq y \Rightarrow d(x, y) > 0$	positiveness,
(p5) $\forall x, y, z \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality.

If the distance function does not satisfy the positiveness property (p4) then the function is called *pseudo-metric*. We do not consider pseudo-metric functions in this work since we can adapt them by identifying all the objects at distance zero as a single object. Such the transformation is correct: if the triangle inequality (p5) holds we can prove that $d(x, y) = 0 \Rightarrow \forall z \in \mathcal{D}, d(x, z) = d(y, z)$. Specifically, by combining the following triangle inequalities

$$d(x, z) \leq d(x, y) + d(y, z),$$

$$d(y, z) \leq d(x, y) + d(x, z)$$

and assuming $d(x, y) = 0$, we get the desired formula. If the symmetry property (p2) does not hold we talk about a *quasi-metric*. For instance, the objects are places in a city and the distance corresponds to distance traveled by car from one place to another. The existence of one-way streets implies the asymmetric behavior of such the distance function. There are techniques to make these distances symmetric from the asymmetric ones, for example, $d_{sym}(x, y) = d_{asym}(x, y) + d_{asym}(y, x)$. Some metrics satisfy a stronger version of the triangle inequality:

$$\forall x, y, z \in \mathcal{D}, d(x, z) \leq max\{d(x, y), d(y, z)\}.$$

These metrics are called *super-metrics* or *ultra-metrics* and are indeed metrics. An equivalent condition is that every triangle has at least two equal sides.

In the rest of this thesis, we refer to a metric function as the distance function or, shortly, the distance. We also assume that the maximum distance never exceeds d^+, thus we consider a *bounded metric space*.

2.2 Distance Functions

In this stage, we provide some examples of distance functions, which can be applied on various types of data. Distance functions are often tailored to a specific application or to a domain of possible applications. Thus, the distance is specified by an expert, however, the definition of distance function is not restricted to any type of queries that can be asked.

In general, distance functions can be divided into two groups with regards to the returned value:

- **discrete** – a distance function returns only a small set of values,
- **continuous** – the cardinality of set of values returned by a distance function is infinite or very large.

An example of the continuous case is the Euclidean distance and an illustration of the discrete function can be the edit distance applied on words. We will refer to this categorization in Chapter 3.

Minkowski Distances

Minkowski distances form the whole family of metric functions, so-called L_p metrics. These distances are defined on vectors of real numbers as follows:

$$L_p[(x_1, \ldots, x_n), (y_1, \ldots, y_n)] = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}.$$

L_1 metric is called Manhattan distance or city distance, L_2 denotes well-known Euclidean distance, and $L_\infty = max_{i=1}^{n}|x_i - y_i|$ is called maximum distance or chessboard distance. Figure 2.1 presents some members of L_p family, where the curves denote points of 2-D space at the same distance from the central point. A convenient application of

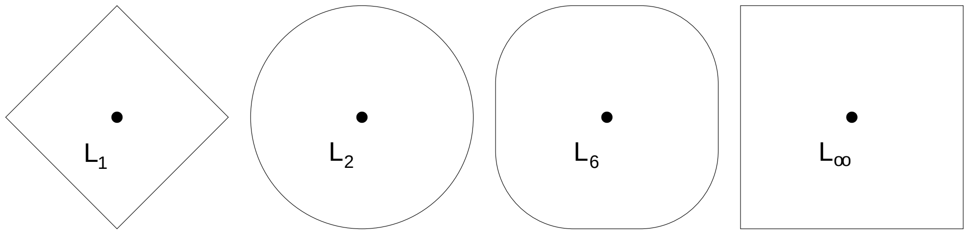


Figure 2.1: The sets of points at the same distance from the center point for different L_p distance measures.

L_p metrics is the computation of distance between color histograms or between vectors of features extracted from images.

Quadratic Form Distance

The Euclidean distance measure L_2 does not provide any correlation of features of color histograms. A distance model that has been successfully applied to image databases [38] and that has the power to model dependencies between different components of features or histogram vectors is provided by the class of quadratic form distance functions [48, 78]. Thereby, the distance measure of two n-dimensional vectors is based on an $n \times n$ matrix $M = [m_{ij}]$ where the weights m_{ij} denote the similarity between the components i and j of the vectors $\vec{x}$ and $\vec{y}$, respectively:

$$d_M(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y}) \cdot M \cdot (\vec{x} - \vec{y})^T}$$

This definition also includes the Euclidean distance when the matrix M is equal to the identity matrix. The quadratic form distance measure is computationally expensive because the color image histograms are typically high-dimensional vectors, consisting of 64 or 256 distinct colors.

Edit Distance

Textual documents or strings are usually compared by the edit distance function, so-called Levenshtein distance [62]. The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of atomic operations needed to transform the string x into y. The atomic operations are:

- insert the character c into the string x at position i
  $ins(x, i, c) = x_1 x_2 \cdots x_i c x_{i+1} \cdots x_n$
- delete the character at position i from the string x
  $del(x, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$
- replace the character at position i in the string x with the new character c
  $replace(x, i, c) = x_1 x_2 \cdots x_{i-1} c x_{i+1} \cdots x_n$

The generalized edit distance function is extended by weights (positive real numbers) assigned to individual atomic operation. Hence, the distance between strings x, y is the minimum value of the sum of weights of atomic operations needed to transform x into y. If the weight of insert and delete operations differ, the edit distance is not symmetric (see property (p2)) and therefore it is not a metric. For convenience, see the following example of such the edit distance:

$w_{ins} = 2$	weight of insert
$w_{del} = 1$	weight of delete
$w_{replace} = 1$	weight of replace

$d_{ed}("combine", "combination") = 9$
– replacement $e \to a$, insertion of t, i, o, n
$d_{ed}("combination", "combine") = 5$
– replacement $a \to e$, deletion of t, i, o, n

In this work, we deal only with metrics, thus, the weights of insert and delete operations have to be the same, the weight of replace may differ. Usually, the edit distance with weights equal to one is used. The excellent survey about string matching is in [68].

Tree Edit Distance

The most well-known distance metrics for trees is the tree edit distance, which has been extensively studied in [46, 28]. The tree edit distance function defines the distance between two trees as the minimal summation of costs needed to convert the source tree to the target tree using the predefined set of edit operations such as insert or delete. In fact, the problem of computing the distance between two trees is a generalization of the problem of computing the distance between two strings to labeled trees. Individual costs of edit operations (atomic operations) can be the same or they can vary with the level in the tree at which the operation is applied. This is supported by the hypothesis that inserting one node near the root may be more significant that inserting a node near the leaf nodes. This, of course, depends on the application domain. Several strategies of setting the costs and computing the tree edit distance are described in the doctoral thesis by Lee [61]. Since XML documents can be easily modeled as trees we can apply the tree edit distance to measure distance between XML documents.

Hausdorff Distance

Another example of distance that can be applied on histograms is the Hausdorff metric [52]. This metric measures the distance between two sets A,B and it is defined as follows:

$$\begin{aligned}
d_p(x, B) &= \inf_{y \in B} d(x, y) \\
d_p(A, y) &= \inf_{x \in A} d(x, y) \\
d_s(A, B) &= \sup_{x \in A} d_p(x, B) \\
d_s(B, A) &= \sup_{y \in B} d_p(A, y).
\end{aligned}$$

Then the Hausdorff distance is:

$$d(A, B) = \max\{d_s(A, B), d_s(B, A)\}.$$

The distance $d(x, y)$ between two elements of sets can be arbitrary, e.g. L_2 metric. In other words, the Hausdorff distance measures the extent to which each point of a "model" set A lies near some point of an "image" set B and vice versa. Apparently, the Hausdorff distance is a very time-consuming operation since its time complexity is $\mathcal{O}(n \times m)$ for sets of size n and m, this can be improved to $\mathcal{O}((n+m)log(n+m))$ [1].

Jacard Coefficient

The distance function applicable to comparing two sets is the Jacard coefficient and is defined as follows:

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

For example, an application may be interested in resemblances between users of an information system. The behavior of each user is expressed by the set of visited addresses (URLs). Thus, the similarity between users can then be measured with the Jacard coefficient.

Time Complexity

In this part, we sum up the time requirements of metric distances. The L_p norms (metrics) are computed linearly in time. However, the quadratic form distance is much more expensive because it uses multiplications by a matrix. Thus, it is evaluated in $\mathcal{O}(n^3)$ time. The dynamic programming algorithms which computes the edit distance on strings have the time complexity $\mathcal{O}(n^2)$. Moreover, the tree edit distance is computed in $\mathcal{O}(n^4)$, more details can be found in [61]. As for metrics for similarity between sets, they are considered as time-consuming operations, as well. In summary, we assume without any particular knowledge about the distance used in a certain application that metric distance functions are generally time-consuming operations. Consequently, almost all indexing techniques for searching in metric spaces minimizes the number of distance evaluations, see Chapter 3.

2.3 Similarity Queries

In general, the problem of indexing metric spaces can be defined as follows.

Problem 2.3.1 *Let $\mathcal{D}$ be a set, d a distance function defined on $\mathcal{D}$, and $\mathcal{M} = (\mathcal{D}, d)$ a metric space. Given a set $X \subseteq \mathcal{D}$, preprocess or structure the elements of X so that similarity queries can be answered efficiently.*

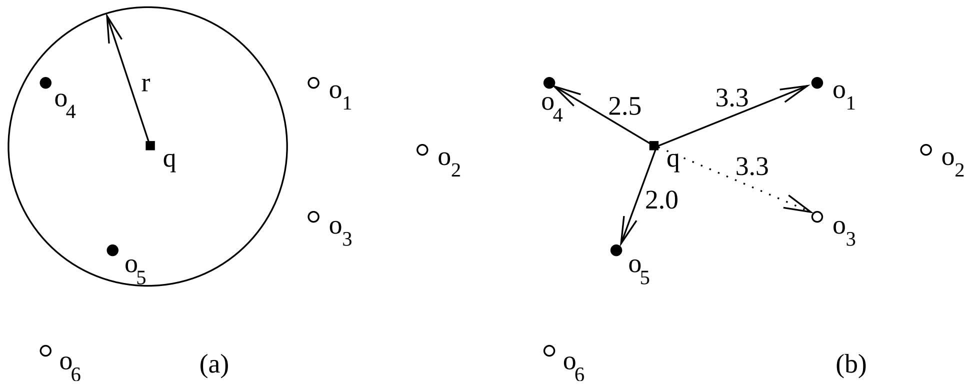


Figure 2.2: (a) Range query $R(q, r)$ and (b) nearest neighbor query 3–$NN(q)$ where the objects o_1, o_3 are at the same distance equal to 3.3 and o_1 has been chosen as the 3^{rd} neighbor at random. Qualifying objects are filled.

A similarity query is usually defined by query object (point) and some selectivity. The answer to the query contains all objects which are close to the given query object and satisfy the selective condition. In the remaining, we provide the reader with definitions of the best-known similarity queries.

Range Query

This is probably the most common type of query that is meaningful almost in every application. The query $R(q, r)$ is specified by the query object q and the query radius r and retrieves all objects which are within the distance of r from q, formally:

$$R(q, r) = \{o \in X, d(o, q) \leq r\}.$$

Notice that a query object q does not necessarily have to belong to the database of objects, i.e. $q \notin X$. However, the query object must be included in the domain of metric space $\mathcal{D}$, i.e. $q \in \mathcal{D}$, this statement applies to all possible types of queries. For convenience, Figure 2.2a shows an example of the range query. Another example of the range query asked to a map system can be: *Give me all museums within the distance of 2km from my hotel.*

Nearest Neighbor Query

The elementary version of this query finds one nearest neighbor, that is, the object closest to the given query object. We give here only the definition of the general case where we look for k nearest neighbors. Precisely, k–$NN(q)$ query retrieves k nearest neighbors to the object q, if a database consists of less than k objects the query returns the whole database.

$$k\text{–}NN(q) = \{R \subseteq X, |R| = k \wedge \forall x \in R, y \in X - R : d(q, x) \leq d(q, y)\}.$$

In the case of ties, if there are more objects than one at the same distance as the k^{th} nearest neighbor and which do not fit the answer we choose the k^{th} nearest neighbor at random. Figure 2.2b depicts the situation with two objects at the same distance. This type of query is more user-friendly in comparison to the range query because a user does not have to know any details about the underlying distance function users only specify the number of objects they want to get. A following request can arise: *Tell me which two museums are the closest to my hotel.*

Reverse Nearest Neighbor Query

In many situations, it is important to know how you are perceived or ranked by others or who consider you as their nearest neighbor. In fact, this is an inverted nearest neighbor search. The basic variant is 1–RNN, it returns all objects which have a given query object as their nearest neighbor. An illustration is provided in Figure 2.3a, the dotted circles denote the distance to the second nearest neighbor of an object o_i. All object satisfying the given 2–$RNN(q)$ query are filled, these objects have q among their two nearest neighbors. Recent works [58, 82, 86, 81, 57] have highlighted the importance of reverse nearest neighbor (1–RNN) queries in decision support systems, profile-based marketing, document repositories, and management of mobile devices. General k–RNN query is defined as follows:

$$\begin{aligned} k\text{–}RNN(q) \quad = \quad & \{R \subseteq X, \forall x \in R : q \in k\text{–}NN(x) \wedge \\ & \forall x \in X - R : q \notin k\text{–}NN(x)\}. \end{aligned}$$

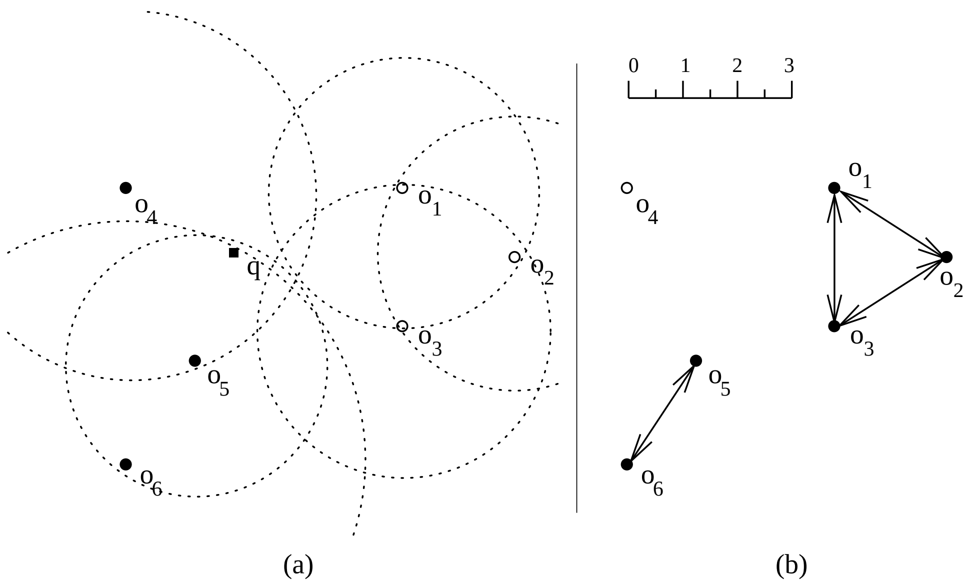


Figure 2.3: (a) Reverse nearest neighbor query $2{-}RNN(q)$ and (b) similarity self join query $SJ(2.5)$. Qualifying objects are filled.

The reader can observe that although an object is far away from the query object q it could still belong to the $k{-}RNN(q)$ response set. Contrary, an object that is near q does not necessarily be in the $k{-}RNN(q)$. This property of reverse nearest neighbors search is called as the *non-locality* property.

Similarity Join Query

The development of Internet services often requires an integration of heterogeneous sources of data. Such sources are typically unstructured whereas the intended services often require structured data. The main challenge is to provide consistent and error-free data, which implies the data cleaning, typically implemented by a sort of similarity join. The similarity join query between two sets $X \subseteq \mathcal{D}$ and $Y \subseteq \mathcal{D}$ retrieves all pairs of objects (x, y) which are at the distance of the given threshold at maximum. Specifically, the *similarity join* query $J(X, Y, \mu)$ is defined as follows:

$$J(X, Y, \mu) = \{(x, y) \in X \times Y : d(x, y) \leq \mu\}, \tag{2.1}$$

where $0 \leq \mu < d^{+}$. If the sets X, Y coincide, i.e. $X = Y$, we talk about *similarity self join* and we denote it as $SJ(\mu) = J(X, X, \mu)$, where X is the database where we search. Figure 2.3b presents an example of similarity self join query $SJ(2.5)$. For example, consider a document collection of books and a collection of compact disk documents. A possible search request can require to find all pairs of books and compact disks which have similar titles.

Combinations

Besides the presented queries, we can define other types of queries as combinations of previous ones. We can compound the range query and the nearest neighbor query to get k–$NN(q, r)$:

$$\{R \subseteq X, |R| = k \wedge \forall x \in R, y \in X - R : d(q, x) \leq d(q, y) \wedge d(q, x) \leq r\}.$$

In fact, we perform the nearest neighbor search but limit the distance to the k^{th} nearest neighbor. Similarly, we can combine the similarity self join and the nearest neighbor search – we limit the number of returned pairs by the similarity self join operation to the value of k. The k closest pairs query is the composition of ideas incorporated in the nearest neighbor query and the similarity join query.

2.4 Search Strategies

The problem of searching metric spaces, formulated in Section 2.3, requires that similarity queries are processed in an efficient manner. To meet the requirement, we often have to structure objects of a metric space to allow the dispose of some objects from the search. The triangle inequality is the key property of metric function postulates for pruning searched objects when processing queries. However, in order to apply the triangle inequality, we often need to use the symmetry property. Furthermore, the non-negativity allows discarding negative values of distance in formulae. In this section, we provide some lemmata to lower-bound and upper-bound the distances between a query object and objects in the database, assuming that we have some knowledge about distances between the query object and some other objects. These assertions can be derived only from the

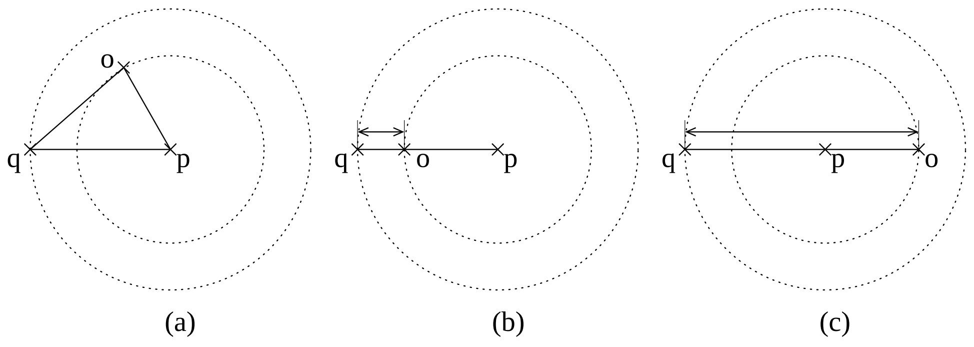


Figure 2.4: Illustration of Lemma 2.4.1: (a) the general case, (b) the lower bound, and (c) the upper bound.

metric postulates. For further details, see an excellent survey by Hjaltason and Samet in [51].

In the first lemma, we describe the situation where we know the distance between the query object q and an object p and the distance between p and an object o, recall that, generally, $q, p, o \in \mathcal{D}$ and $(\mathcal{D}, d)$ is the metric space.

Lemma 2.4.1 *Given a metric space $\mathcal{M} = (\mathcal{D}, d)$ and any three objects $q, p, o \in \mathcal{D}$, we get:*

$$|d(q,p) - d(p,o)| \leq d(q,o) \leq d(q,p) + d(p,o).$$

Consequently, we can bound the distance $d(q,o)$ from below and above by knowing the distances $d(q,p)$ and $d(p,o)$.

Proof: As for the lower bound, we have the following triangle inequalities: $d(p,q) \leq d(p,o) + d(o,q)$ and $d(o,p) \leq d(o,q) + d(q,p)$. They imply $d(p,q) - d(p,o) \leq d(o,q)$ and $d(o,p) - d(q,p) \leq d(o,q)$, respectively. Applying the symmetry property and combining them together, we get $|d(p,q) - d(p,o)| \leq d(q,o)$. The upper bound is the direct implication of the triangle inequality $d(q,o) \leq d(q,p) + d(p,o)$. ■

The situation of Lemma 2.4.1 is depicted in Figure 2.4. The general case is in the (a) figure. The lower bound is present in the (b) figure, specifically, the lower bound is equal to $d(q,p) - d(p,o)$. The second case of the lower bound can be obtained by exchanging the objects q and o. On contrary, the upper bound is depicted in the (c) figure, where the distance $d(q, p) + d(p, o)$ is attained.

In the next lemma, we assume that we know the distance between a query q and an object p. The distance between an object o and p is not known, we only have the interval within the distances $d(p, o)$ are.

Lemma 2.4.2 *Given a metric space $\mathcal{M} = (\mathcal{D}, d)$ and objects $o, p \in \mathcal{D}$ such that $r_l \leq d(o, p) \leq r_h$. Given $q \in \mathcal{D}$ and $d(q, p)$, the distance $d(q, o)$ can be restricted to the range:*

$$max\{d(q, p) - r_h,\ r_l - d(q, p),\ 0\} \leq d(q, o) \leq d(q, p) + r_h.$$

Proof: To prove this lemma, we make use of triangle inequalities. First, we consider the inequality $d(q, o) \leq d(q, p) + d(p, o)$. Using assumption $d(p, o) \leq r_h$ we can substitute r_h for the distance $d(p, o)$ and we get the desired formula of upper bound $d(q, o) \leq d(q, p) + r_h$. Second, we focus on the lower bound that we decompose to three cases. The first case, see Figure 2.5(a), considers the position of the query q_1 above the range $[r_l, r_h]$. We have $d(q, p) \leq d(q, o) + d(o, p)$ and transform it to $d(q, p) - d(o, p) \leq d(q, o)$. Again using the inequality $d(o, p) \leq r_h$ we replace $d(o, p)$ by r_h and we get $d(q, p) - r_h \leq d(q, o)$. As for the second case, the query object q_2 in Figure 2.5(b) is placed below the interval $[r_l, r_h]$. We start with $d(o, p) \leq d(o, q) + d(q, p)$ and we transcribe it to $d(p, o) - d(q, p) \leq d(q, o)$, we have also applied the symmetry property twice. We combine it with the assumption $r_l \leq d(p, o)$ and we get $r_l - d(q, p) \leq d(q, o)$. In Figure 2.5(c), the query object q_3 is within the range $[r_l, r_h]$ which implies that the query object and the object o can be identical, thus, the distance is $d(q, o) \geq 0$. The lower bounds of the first two cases are negative in other positions of query (i.e. in positions that was not specified in the proof). Thus, the overall lower bound is maximum of all three lower bounds, $max\{d(q, p) - r_h,\ r_l - d(q, p),\ 0\} \leq d(q, o)$. ■

Figure 2.5 illustrates three different positions of the query objects and emphasizes lower and upper bounds, denoted by dotted and dashed lines, respectively. The distance $d(q_3, o)$ is lower-bounded by zero because q_3 lies in the range $[r_l, r_h]$ of objects o.

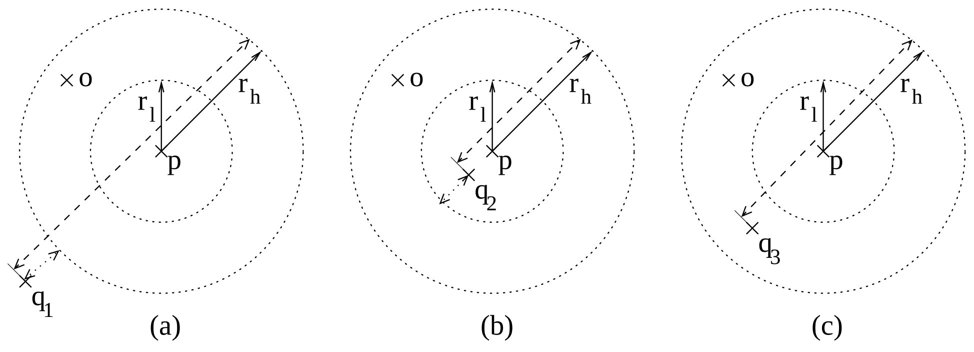


Figure 2.5: Illustration of Lemma 2.4.2 with three different positions of the query object; above, below, and within the range $[r_l, r_h]$. We can bound $d(q, o)$, providing that $r_l \leq d(p, o) \leq r_h$ and $d(q, p)$. The dotted and dashed lines denote lower and upper bounds, respectively.

In the following lemma, we do not know $d(q, p)$ neither $d(p, o)$ exactly. We only have ranges in which the distances are, that is, $d(p, o) \in [r_l, r_h]$ and $d(q, p) \in [r'_l, r'_h]$.

Lemma 2.4.3 *Given a metric space $\mathcal{M} = (\mathcal{D}, d)$ and objects $o, p, q \in \mathcal{D}$ such that $r_l \leq d(p, o) \leq r_h$ and $r'_l \leq d(q, p) \leq r'_h$. The distance $d(q, o)$ can be bounded to the range:*

$$max\{r'_l - r_h,\ r_l - r'_h,\ 0\} \leq d(q, o) \leq r_h + r'_h.$$

Proof: The upper bound is a direct consequence of the upper bound of Lemma 2.4.2 and the assumption $d(q, p) \leq r'_h$, thus, we get the desired by substitution r'_h for $d(q, p)$ in the upper bound in Lemma 2.4.2 because we want to maximize the distance. As for the lower bound, we use the lower bound from Lemma 2.4.2. We want to decrease as much as possible all expression is the max function, i.e. $max\{d(q, p) - r_h,\ r_l - d(q, p),\ 0\}$. Thus, we replace $d(q, p)$ by r'_l in the first expression and by r'_h in the second and get the desired. ■

Figure 2.6 demonstrates the principle of Lemma 2.4.3. The lower bound can be seen as two shells centered in p, one shell is with radius

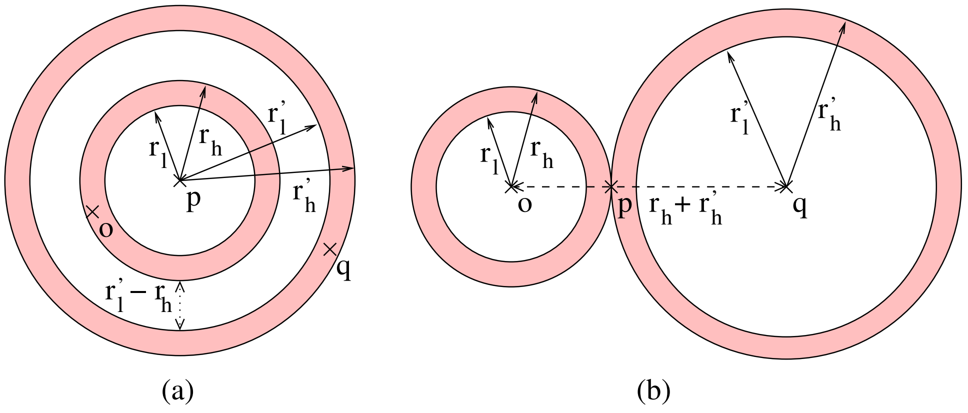


Figure 2.6: Illustration of Lemma 2.4.3, $d(p,o)$ is within the range $[r_l, r_h]$, and $d(q,p)$ is within $[r'_l, r'_h]$. The lower bound on $d(q,o)$ is expressed in (a), the upper bound on $d(q,o)$ is presented in (b). The lower and upper bounds are emphasized with dotted and dashed lines, respectively.

range $[r_l, r_h]$, where o is situated, and the second with $[r'_l, r'_h]$, where q lies. Figure 2.6(a) presents the lower bound $r'_l - r_h$. The second part of lower bound is the symmetric situation where both the shells are swapped. The third part 0 expresses the position when the shells collide. Figure 2.4.3(b) provides the illustration of the upper bound.

In some distance-based indices, objects are partitioned on relative closeness to two or more objects. The following lemma provides a result that can be used to prune some objects in such situations.

Lemma 2.4.4 *Assume a metric space $\mathcal{M} = (\mathcal{D}, d)$ and objects $o, p_1, p_2 \in \mathcal{D}$ such that $d(o,p_1) \leq d(o,p_2)$. Given a query object $q \in \mathcal{D}$ and the distances $d(q,p_1)$ and $d(q,p_2)$, the distance $d(q,o)$ can be lower-bounded as follows:*

$$max\left\{\frac{d(q,p_1) - d(q,p_2)}{2},\ 0\right\} \leq d(q,o).$$

Proof: We have the triangle inequality $d(q,p_1) \leq d(q,o) + d(o,p_1)$ which yields $d(q,p_1) - d(q,o) \leq d(o,p_1)$ and we will use it later on. We combine the triangle inequality $d(p_2,o) \leq d(p_2,q) + d(q,o)$ and the assumption $d(o,p_1) \leq d(o,p_2)$ and we obtain $d(o,p_1) \leq d(p_2,q) +$

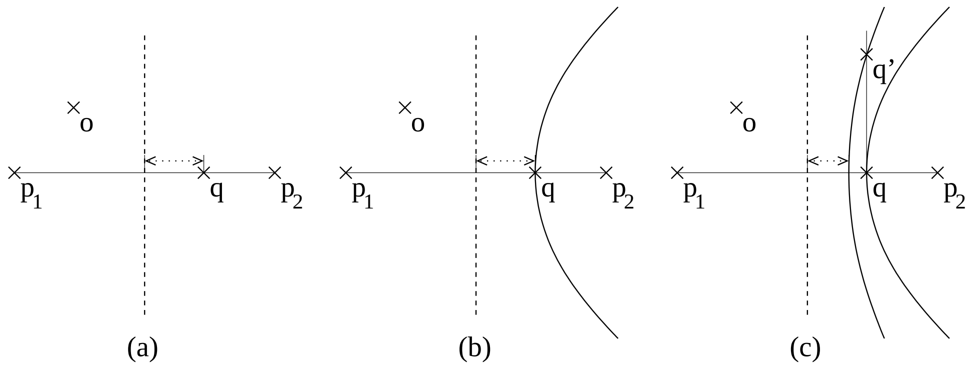


Figure 2.7: Illustration of Lemma 2.4.4. As the query object moves along the dashed equidistant line (i.e. to q') the lower bound shrinks. The lower bound $\frac{d(q,p_1)-d(q,p_2)}{2}$ is denoted by dotted arrow line.

$d(q,o)$. This inequality is combined with the first one and we acquire $d(q,p_1) - d(q,o) \leq d(p_2,q) + d(q,o)$. By rearranging, we get $d(q,p_1) - d(q,p_2) \leq 2d(q,o)$. We assumed that o is closer to p_1. If q is closer to p_2 the derived expression is positive. On contrary, if q is closer to p_1, that is q is on the same side as o, the expression becomes negative. The non-negativity property forbids it, thus the overall assertion is $max\left\{\frac{d(q,p_1)-d(q,p_2)}{2},\ 0\right\} \leq d(q,o)$. ■

Figure 2.7(a) shows the situation and the lower bound is emphasized by the dotted line. If q and o are both on the same side (on the left side in the figure) the lower bound is zero, which is the second parameter of max function. Figures 2.7(b) presents the hyperbolic curve of which points have the same value of $\frac{d(q,p_1)-d(q,p_2)}{2}$. If we move the query point up, see Figure 2.7(c), apparently, we preserve the distance from q to the equidistant dashed line, which consists of points at the same distances from both p_1 and p_2. Nonetheless, the value of $\frac{d(q,p_1)-d(q,p_2)}{2}$ decreases. This corresponds to the movement of the hyperbola in a way that the query object q becomes an element of the curve, i.e. the hyperbola moves closer to the equidistant line. In consequence, the expression $\frac{d(q,p_1)-d(q,p_2)}{2}$ is indeed the lower bound on the distance from q to o. The upper bound cannot be established because o can be arbitrarily far away from p_1 and p_2.

In the next sections, we present two basic algorithms used by nearly all available indexing techniques. The former is based on partitioning paradigm and the latter concerns filtering based on knowledge of distances between objects.

2.5 Partitioning Algorithms

Search structures for vector spaces are called **Spatial Access Methods** (SAM). These methods use the idea of partitioning an indexed space into subsets. In general, we call this principle as dividing into equivalence classes. Specifically, we group objects into one set if they have some property in common, e.g. the first coordinate of a vector is lower than three. For example, **kd-trees** [6, 7] or **quad-trees** [76] use a threshold (value of a coordinate) to divide the given space into subsets. On contrary, **R-Trees** [47] cluster near objects and bound them using the *minimum bounding rectangle*. This approach systematically clusters objects from leaves to the root.

Similar techniques are utilized in the area of metric spaces. The problem of partitioning metric spaces is more difficult because we have no coordinate system that can be used in geometric splits. In metric spaces, we pick one object from given dataset (generally, from the domain of metric space) and promote it to the *pivot*. Using pivot, we can order all remaining objects by the distance. Consequently, we choose a certain value of distance, which plays the role of threshold and we partition objects into two subsets.

In the following, we present concepts based on this idea. The simplest variant that uses one pivot and one radius to split object into two subsets can be formalized as:

$$bp(o,p,r) = \begin{cases} 0 \; if \; d(o,p) \leq r \\ 1 \; if \; d(o,p) > r \end{cases}$$

This type of partitioning is called the *ball partitioning* and was defined by Uhlmann in [83]. In fact, this function returns the identification of the set to which object o belongs. This approach can be extended to use more than one radius. Thus, we define bp_3 function that splits

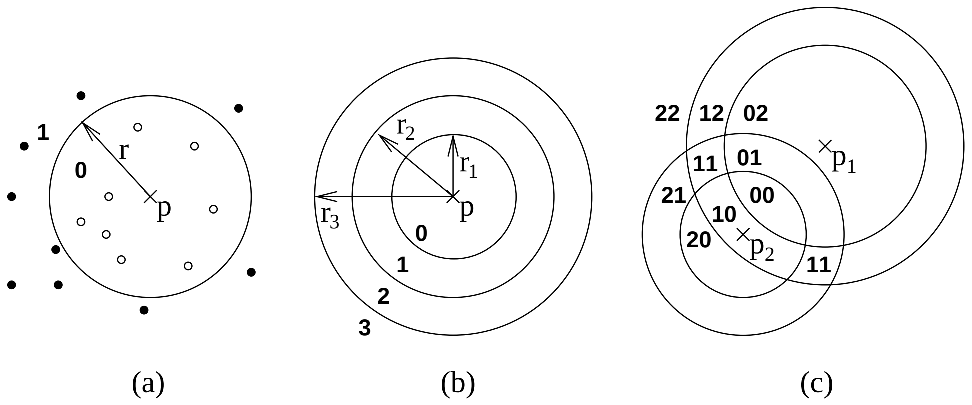


Figure 2.8: Examples of ball partitioning. The basic *bp* function (a), more spherical cuts (b), and combination of two functions (c).

metric space into four spherical cuts:

$$bp_3(o, p, r_1, r_2, r_3) = \begin{cases} 0 \; if \; d(o,p) \le r_1 \\ 1 \; if \; d(o,p) > r_1 \wedge d(o,p) \le r_2 \\ 2 \; if \; d(o,p) > r_2 \wedge d(o,p) \le r_3 \\ 3 \; if \; d(o,p) > r_3 \end{cases}$$

The illustration of *bp* functions is given in Figure 2.8. For example, these functions are used in the Vantage Point Tree (**VPT**) and the multi-way Vantage Point Tree (**mw-VPT**) [13]. In the same paper, Bozkaya and Özsoyoglu defined the combination of *bp* functions, which uses more pivots. An example is presented in Figure 2.8(c) where two bp_2 functions are combined to get partitioning into $3^2 = 9$ subsets.

More pivots can be also used in a different way. Uhlmann [83] has defined the *hyperplane partitioning*. This partitioning uses two pivots to divide objects into two subsets on their relative distances to the pivots. Formally, hyperplane partitioning is defined as follows:

$$hp(o, p_1, p_2) = \begin{cases} 0 \; if \; d(o,p_1) \le d(o,p_2) \\ 1 \; if \; d(o,p_1) > d(o,p_2) \end{cases}$$

Figure 2.9(a) depicts the situation. Principle of *hp* function is applied in the Generalized Hyperplane Tree (**GHT**) [83]. Apparently, this

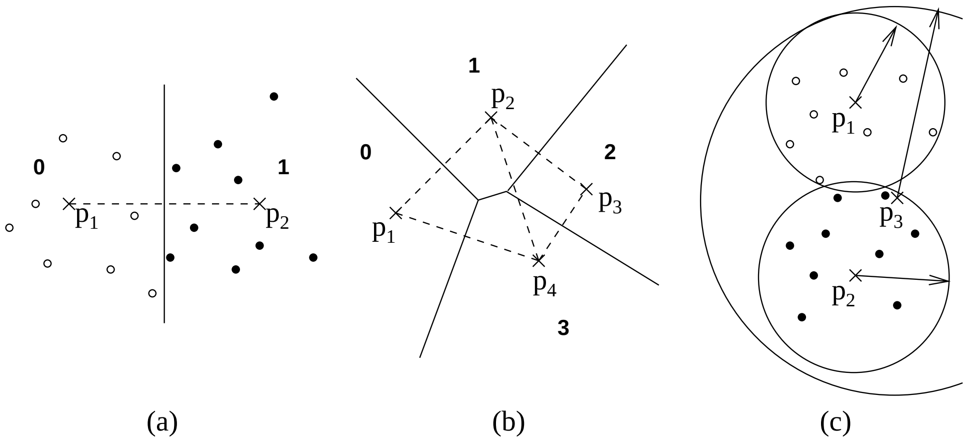


Figure 2.9: Illustration of hyperplane partitioning. The basic hp function (a) and general partitioning (b). The clustering principle is depicted in (c).

idea is related to the Voronoi-like partitioning [2]. Thus, we can generalize hp functions to use more pivots. The following definition uses three pivots:

$$hp_3(o, p_1, p_2, p_3) = \begin{cases} 0 \; if \; d(o,p_1) \leq d(o,p_2) \wedge d(o,p_1) \leq d(o,p_3) \\ 1 \; if \; d(o,p_2) < d(o,p_1) \wedge d(o,p_2) \leq d(o,p_3) \\ 2 \; if \; d(o,p_3) < d(o,p_1) \wedge d(o,p_3) < d(o,p_2) \end{cases}$$

The generalized case is used in the Geometric Near-Access Tree (**GNAT**) [14]. An example of hp_4 function is depicted in Figure 2.9(b).

To complete partitioning methods, we have to remark the clustering approach used in R-trees. In metric spaces, this principle is applied in **M-trees** [25] where objects in leaves are clustered into small groups and these groups are recursively joined into bigger clusters that cover them. For convenience, see Figure 2.9(c).

2.6 Filtering Algorithms

In this section, we introduce second category of algorithms used in indexing metric spaces. Filtering algorithms very often go hand in hand with partitioning algorithm and are used together in a single

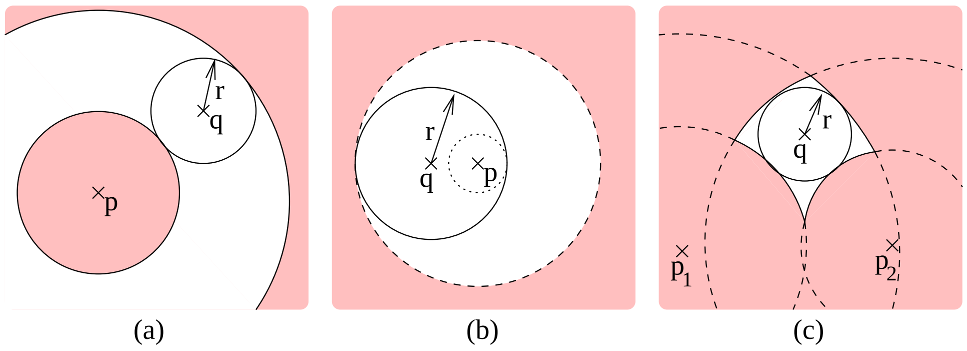


Figure 2.10: Illustration of filtering algorithm. The lower bound principle (a), both the upper bound (dotted circle) and the lower bound are applied (b), and filtering using two distinct pivots.

system. The gist of filtering algorithms is to eliminate some objects without the need to compute their relative distances to a query object, i.e. without accessing them. Such the elimination is feasible if we have some information about the distances from a pivot to other objects and from the query object to the pivot. For this reason, we can apply Lemma 2.4.1 and state lower and upper bounds on distance $d(q, o)$. Considering the range query $R(q, r)$, we can eliminate all objects such that their lower bound $|d(q, p) - d(p, o)|$ of Lemma 2.4.1 is greater than r, that is, when the distance $(q, o) > r$. On the other hand, objects that satisfy $d(q, p) + d(p, o) \leq r$ are included in the result set of the range query actually without computing the distance $d(q, o)$. In fact, this applies when the upper bound of Lemma 2.4.1 is less than or equal to the radius of the range query, thus, it is sure that object satisfies the query. Figure 2.10(a) depicts extremes of the lower bound. The utilization of the upper bound is illustrated by the dotted circle in Figure 2.10(b), objects within the dotted circle are not examined by the metric function (distance $d(q, o)$ is not evaluated) and they are directly included in the response set of the range query. Specifically, objects in the filled area are excluded from searching without computing $d(q, o)$ (the lower bound is greater than the query radii).

By combination of more pivots, we can improve the probability of excluding objects without computing distances to a query object.

We formalize this concept in the following lemma.

Lemma 2.6.1 *Assume a metric space* $\mathcal{M} = (\mathcal{D}, d)$*, a set of pivots* $P = \{p_1, \dots, p_n\}$*, and a mapping* Ψ*:* $(\mathcal{D}, d) \to (\mathbb{R}^n, L_\infty)$*, such that,* $\Psi(o) = (d(o,p_1), d(o,p_2), \dots, d(o,p_n))$*. We can bound the distance* $d(q,o)$ *from below:*

$$L_\infty(\Psi(q), \Psi(o)) \leq d(q,o).$$

Proof: Considering the definition of L_∞ metric function, see page 17, we get $L_\infty(\Psi(q), \Psi(o)) = max_{i=1}^{n}|d(q,p_i) - d(o,p_i)|$. Apparently, we have that all values of $|d(q,p_i) - d(o,p_i)|$ are less than or equal to $d(q,o)$. By multiple usage of Lemma 2.4.1, we get the desired lower bound. ■

The mapping function Ψ used in Lemma 2.6.1 stands in for the precomputed distances between pivots and objects, those distances are known in advance. The mapping function is also applied on the query object q and the characteristic vector of distances $\Psi(q)$ is determined. Once we have $\Psi(q)$ and $\Psi(o)$, we can directly apply the lower bound criterion. For convenience, Figure 2.10(c) shows the filtering with two pivots. The light area represents the objects that cannot be eliminated from searching using the distances to pivots. These objects have to be tested directly against the query object q with the original metric function d.

The mapping Ψ is *contractive*. It means that on no account the distance $L_\infty(\Psi(o_1), \Psi(o_2))$ is greater than the distance $d(o_1, o_2)$ in the original metric space. The outcome of a range query in the projected space $(\mathbb{R}^n, L_\infty)$ is a candidate list, which can contain some spurious objects that do not qualify the query and have to be tested by the original distance function d.

2.7 Choosing Pivots

The problem of choosing reference objects (pivots) is important for any search technique in general metric spaces, because all such structures need, directly or indirectly, some "anchors" for partitioning and search pruning, see Sections 2.5 and 2.6. It is well known that the way in which pivots are selected can affect the performance of search algorithms. This has been recognized and demonstrated by several researchers, e.g. [87], [13], or [79], but specific suggestions for choosing good reference objects are rather vague.

Due to the problem complexity, pivots are often chosen at random, or as space outliers, i.e. having large distances to other objects in the metric space. Obviously, independent outliers are good pivots, but any set of outliers does not ensure that the set is good. Qualitatively, the current experience can be summarized as:

- good pivots are *far away* from the rest of objects of the metric space,
- good pivots are *far away* from each other.

Recently, the problem was systematically studied in [17], and several strategies for selecting pivots have been proposed and tested. We will focus on selecting pivots later in Section 4.4.6.

Chapter 3

Metric Index Structures

In this chapter, we provide an overview of existing indexes for metric spaces. The more complete and more detailed survey is available in [22]. This chapter is organized into three parts. The first stage presents index structures based on the ball partitioning principle described in Section 2.5. Next, methods that use the hyperplane partitioning are concerned. Finally, we focus on techniques that map a metric space into a vector space. All the presented time and space complexities are adopted from [22] and source papers.

3.1 Ball Partitioning Methods

In this section, we start with methods based on the ball partitioning. First, we introduce Burkhard-Keller Tree, which is probably the very first solution for indexing metric spaces. Next, we present Fixed Queries Tree and its further evolution called Fixed Queries Array. We conclude this section with Vantage Point Trees and Multi Vantage Point Trees we also include Excluded Middle Vantage Point Forests, which introduce the concept of excluding the middle area from partitioning.

3.1.1 Burkhard-Keller Tree

Apparently, the first solution to support searching in metric spaces is presented in [16]. A Burkhard-Keller Tree (**BKT**) is suitable for discrete distance functions. The tree is built recursively in the following manner: an arbitrary object $p \in X$ is selected as the root node of the tree, where X is the indexed data set. For each distance $i \geq 0$, a set X_i is defined as a group of all objects at the distance of i from the root p,

$X_i = \{o \in X, d(o,p) = i\}$. A child node of the root p is built for every non-empty set X_i. All child nodes can be recursively repartitioned until no new child node can be created. If a child node is repartitioned the child node contains an object o_i, which is picked from the set X_i as a representative of it. The leaf node is created for every set X_i if X_i is not repartitioned again. In general, objects assigned as roots of subtrees (saved in internal nodes) are called *pivots*.

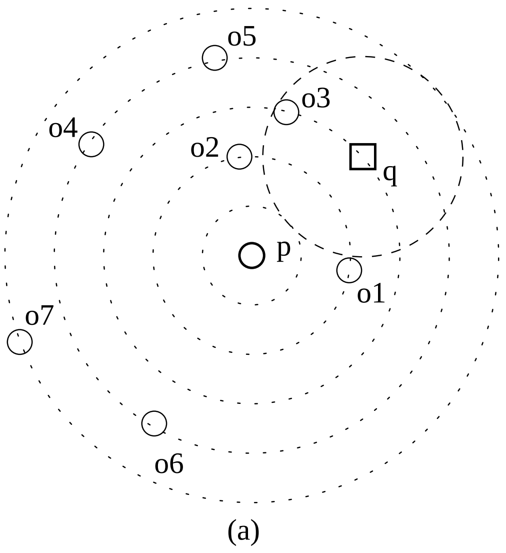
The algorithm for range queries is very simple. The range search $R(q,r)$ starts at the root node p of the tree and compares the object p with the query object q. If p satisfies the query, that is if $d(p,q) \leq r$, the object p is reported. Subsequently, the algorithm enters all child nodes o_i such that

$$max\{d(q,p) - r, 0\} \leq i \leq d(q,p) + r \tag{3.1}$$

and proceed down recursively. Notice that the inequality 3.1 cuts out some branches of the tree. The inequality is a direct consequence of lower bounds provided by Lemma 2.4.2. In particular, by applying the lemma with $r_l = i$ and $r_h = i$, we find that the distance from q to an object o in the inspected tree branch is at least $max\{d(q,p) - i, i - d(q,p), 0\}$. Thus, we visit the branch i if and only if $max\{d(q,p) - i, i - d(q,p), 0\} \leq r$.

Figure 3.1(b) shows an example where the BKT tree is constructed from objects of the space that is illustrated in the figure (a). Objects p, o_1, and o_4 are selected as roots of subtrees, so-called pivots. The range query is also given and specified by the object q and radius $r = 2$. The search algorithm evaluating the range query $R(q,2)$ discards some branches and the accessed branches are emphasized. Obviously, if the radius of range query grows the number of accessed subtrees (branches) increases. This leads to higher search costs, which are usually measured in terms of the number of distance computations needed to answer a query. During the range query elaboration the algorithm traverses the tree and evaluates distances to pivots, which are accommodated in internal nodes. Thus, the increasing number of accessed subtrees leads to the growing number of distance computations due to the different pivots in individual branches. The following structure, fixed-queries trees (FQT), reduces such the costs.

The BKT trees are linear in space $\mathcal{O}(n)$. The construction complexity is $\mathcal{O}(n \log n)$ and is measured in terms of the number of dis-



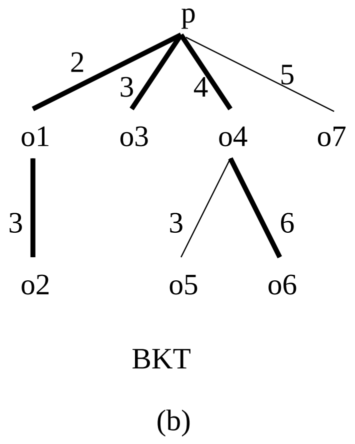


Figure 3.1: An example of a metric space and a range query (a). BKT tree (b) is built on the space.

tance computations needed to construct the tree. The search time complexity is $\mathcal{O}(n^{\alpha})$ where α is a real number satisfying $0 < \alpha < 1$ and depends on the search radius and the structure of the tree [22]. The search complexity is also measured in terms of distance computations.

3.1.2 Fixed Queries Tree

The **FQT** (Fixed Queries Tree), originally presented in [4], is an extension of BKT tree. Fixed Queries Trees use the same pivot in all nodes at the same level, that is in contrast to BKT trees where several pivots per level are used, see Figures 3.1(b) and 3.2(a). All objects of a given dataset X are stored in leaves only and internal nodes are used only for navigation during the search. The range search algorithm is the same as for the BKT. The advantage of this structure is following, if more than one subtree has to be accessed only one distance computation between the query object and the pivot per level is evaluated because all nodes at the same level share the same pivot. The experiments, presented in [4], reveal that FQTs perform less distance computations than BKTs.

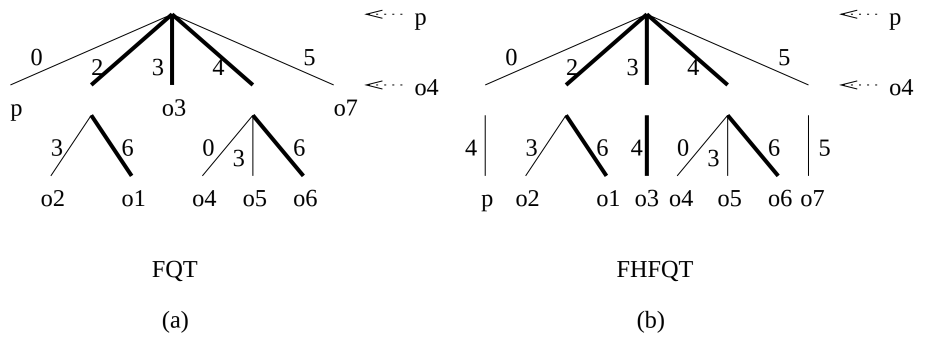


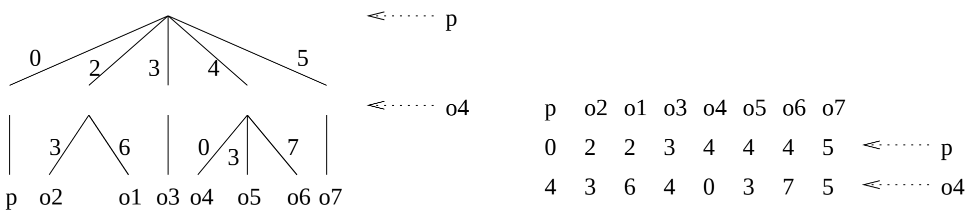
Figure 3.2: Examples of FQT tree (a) and FHFQT tree (b) that are built on the given space in Figure 3.1(a).

An example of FQT tree is demonstrated in Figure 3.2(a). The tree is built from objects depicted in Figure 3.1(a), where objects p and o_4 are pivots of corresponding levels. Notice that all objects are stored in leaves, including objects selected as pivots. The range query $R(q, 2)$ is defined and accessed branches of the tree are highlighted.

The space complexity is superlinear because the objects selected as pivots are duplicated. The complexity varies from $\mathcal{O}(n)$ to $\mathcal{O}(n \log n)$. The number of distance computations required to build the tree is $\mathcal{O}(n \log n)$. The search complexity is $\mathcal{O}(n^{\alpha})$ where $0 < \alpha < 1$ depending on the query radius and an indexed space.

In [4, 3], the authors propose a variant of FQT trees, called **FHFQT** (Fixed-Height Fixed Queries Tree). This structure has all leaf nodes at the same level, i.e. leaves are at the same depth h. In other words, leaves are replaced with paths. The enlargement of the tree can actually improve the search performance. If we increase the height of the tree by thirty we add only thirty more distance computations for the entire similarity search. We may introduce many new node traversals but they cost much less. However, thirty more pivots will filter out many more objects, so the final candidate set will be smaller – this idea is explained in Section 2.6. For convenience, see Figure 3.2(b) where an example of the fixed-height FQT is provided.

The space complexity of FHFQT is superlinear and is somewhere between $\mathcal{O}(n)$ and $\mathcal{O}(nh)$, where h is the height of the tree, this is the same as for FQT trees. The FHFQT tree is constructed in $\mathcal{O}(nh)$ dis-



FHFQT (a) FQA (b)

Figure 3.3: Example of the FHFQT tree (a). FQA is built from FHFQT in (b).

tance computations. The claimed search complexity is constant $\mathcal{O}(h)$, which is the number of distances evaluations computed to pivots. However, the extra CPU time, i.e. the number of traversed nodes, remains $\mathcal{O}(n^\alpha)$, where $0 < \alpha < 1$ depending on the query radius and an indexed space. The extra CPU time is spent by comparing values (integers) of distances and by traversing the tree. In practice, the optimal height $h = \log n$ cannot be achieved because of space limitations.

3.1.3 Fixed Queries Array

The Fixed Queries Array (**FQA**) is presented in [21, 20]. The structure of FQA is strongly related to FHFQT but it is not a tree structure. First, the FHFQT tree with fixed height h is built on a given dataset X. If the leaves of the FHFQT are traversed in order from left to right and put in an array the outcome is exactly the FQA array. Each element of the array consists of h numbers representing distances to every pivot utilized in the FHFQT, in fact, the sequence of h numbers is the path from the root of FHFQT to the leaf. The FQA structure simply stores the database objects lexicographically sorted by this sequence of distances. Thus, the objects are first sorted to the first pivot and those at the same distance to the first pivot are sorted to the second pivot and so forth. For illustration, Figure 3.3 shows the FQA array constructed from the FHFQT tree.

The range search algorithm is inherited from the FHFQT tree.

Each internal node of FHFQT corresponds to a range of elements in the FQA. If a node is a child of a parent node its range of elements is a subrange of the parent's range in the array. There is a similarity between FQA approach and suffix trees and suffix arrays [40]. The movement in the algorithm of FHFQT is simulated by the binary searching the new range inside the current one.

The FQA array is able to use more pivots than FHFQT, this improves the efficiency and search pruning. The authors of [21] show that the FQA outperform the FHFQT. The space requirements are nhb bits, where b is the number of bits used to store one distance. The number of distance computations evaluated during the search is $\mathcal{O}(h)$. Authors in [5] proved that FHFQT has $\mathcal{O}(n^\alpha)$ extra CPU complexity. FQA has $\mathcal{O}(n^\alpha \log n)$ extra complexity, where $0 < \alpha < 1$. The extra CPU time is expended in the binary search of the array.

All previously presented structures (BKT, FQT, FHFQT) are designed for discrete metric functions (see Section 2.2 for details). FQA also do not work well if the distance is continuous. The matter is that we cannot have a separate child for any value of the continuous distance function. If we do the tree degenerates to a flat tree of height two. Thus, the search algorithm in such the tree would perform almost like a sequential scan.

For the continuous case, we have to segment the range of possible values of the distance into a small set of subranges. Authors of [20, 21] introduce two discretizing schemata for the FQA. The former divides the range of possible values into identical slices, in terms of the width of slice. The result is the **Fixed Slices Fixed Queries Array**. Such the partitioning would lead to empty slices where no database object is accommodated. This motivates the latter approach that partitions the entire range into slices with the same number of database objects. In other words, we divide that range into fixed quantiles. The resulting FQA is called the **Fixed Quantiles Fixed Queries Array**.

3.1.4 Vantage Point Tree

The Vantage Point Tree (**VPT**) is a metric tree suitable to use with continuous distance functions. Discrete distance functions are also supported with virtually no modifications. VPT trees [87] are based on the same idea as a previous work – **Metric Tree** [83].

In VPT tree, we pick an object $p \in X$ and promote it to a pivot, in [87] termed the vantage point, and the median d_m of the distances between objects $o \in X$ and p. The pivot and the median are used to divide the remaining objects into two groups as follows:

$$S_1 = \{o \in X | d(o,p) \leq d_m\},$$

$$S_2 = \{o \in X | d(o,p) > d_m\}.$$

Both the sets are roughly equal sized. The objects in S_1 are inside the ball of radius d_m around p and the objects in S_2 are outside the ball. The sets S_1 and S_2 form the left and the right branch of the node with pivot p, respectively. Applying this principle recursively we build a binary tree. In the leaf nodes, there can be stored one or more objects depending on the capacity. The example of the partitioning is presented in Section 2.5 and referred as the ball partitioning. Using this procedure, we produce a balanced binary tree. The application of median to divide a dataset into two subsets can be replaced with a different strategy which uses the mean of the distances between the objects in $X \setminus \{p\}$. This method, so-called the middle point, can yield better results for high dimensional datasets [22]. The disadvantage is that the middle point produces unbalanced tree and consequently deteriorates the search algorithm efficiency.

The search algorithm for a range query $R(q,r)$ traverses the VPT tree from the root to leaves. In each internal node, it evaluates the distance between the pivot p of the node and the query object q, that is $d(q,p)$. If $d(q,p) \leq r$ holds the pivot p is reported to output. In internal nodes, the algorithm also must decide which subtrees are accessed. To do so, we must establish lower bounds on the distances from q to objects in the left and right branches. If the query radius r is less than the lower bound the algorithm does not visit the corresponding subtree. Figure 3.4(a) provides an example of situation where the inner ball need not to be accessed whereas Figure 3.4(b) shows the example where both the subtrees must be examined. The lower bounds are established using Lemma 2.4.2. Precisely, by applying the equation with $r_l = 0$ and $r_h = d_m$ we expose that the distance from q to any object in the left branch is at least $max\{d(q,p) - d_m, 0\}$. Likewise, by applying the equation with

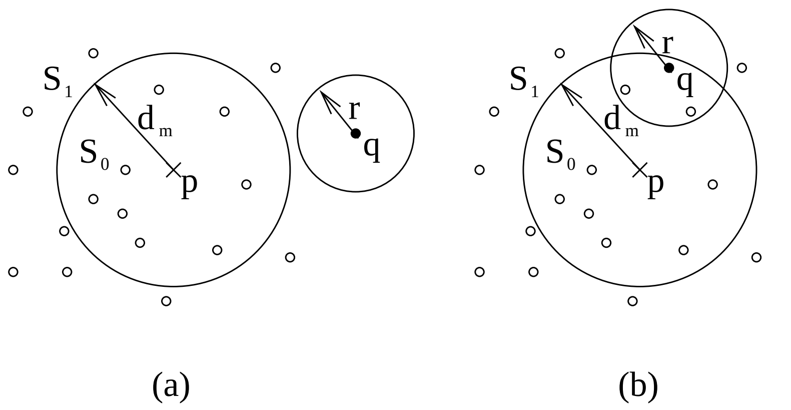


Figure 3.4: An example of range queries, in (a), S_0 need not to be accessed during the search and both the set must be visited in (b).

$r_l = d_m$ and $r_h = \infty$ we get that the distance from q to an object in the right subtree is at least $max\{d_m - d(q,p), 0\}$. Thus, we enter the left branch if $max\{d(q,p) - d_m, 0\} \leq r$ and the right branch if $max\{d_m - d(q,p), 0\} \leq r$. Notice that both subtrees can be visited.

The ball partitioning principle applied in VPTs does not guarantee that the ball around the pivot p_2 is completely inside the ball around the pivot p_1, which is the parent of p_2. For convenience, see Figure 3.5 where the situation is depicted and a query object q is given. In general, it is possible that the lower bound from q to a child node is smaller than the lower bound from q to the child's parent node, that is

$$max\{d(q,p_2) - d_m^2, 0\} < max\{d(q,p_1) - d_m^1, 0\}.$$

Nonetheless, this does not affect the behavior and correctness of the search algorithm since objects rooted in the subtree of p_2 are not closer than $max\{d(q,p_1) - d_m^1, 0\}$, even though the lower bounds claim the opposite. In other words, the objects in the left subtree of p_2 (the set S_1) are somewhere in the white area inside the ball of p_2 and not in the shaded region (see Figure 3.5). On the other hand, the objects of the right branch (the set S_2) must be in the dashed area and not outside the ball around p_1.

During the building of VPT many distance computations between pivots and objects are evaluated. These distances are com-

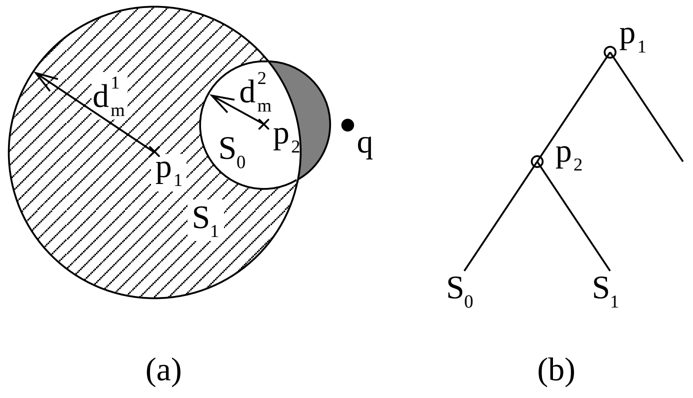


Figure 3.5: An example of VPT with two pivots p_1 and p_2 in (a). The corresponding representation of the tree in (b).

puted for every object o in a leaf to each pivot p on the path from the leaf to the root. We can take advantage of this information during the search algorithm to prune some objects. This strategy is used in **VP**^{s} trees, the variant of VPTs, proposed in [87]. These distances are remembered and stored in the structure of VP^s tree. They are then used in the range search algorithm in the following way:

- if $|d(q, p) - d(p, o)| > r$ holds we discard the object o without actually computing the distance $d(q, o)$,
- if $d(q, p) + d(p, o) \leq r$ holds we directly include the object o to the answer to the range query $R(q, r)$, again without computing the distance $d(q, o)$.

Given the distances $d(q, p)$ and $d(p, o)$, Lemma 2.4.1 forms the lower and upper bounds on the distance between q and o:

$$|d(q, p) - d(p, o)| \leq d(q, o) \leq d(q, p) + d(p, o).$$

The previous two pruning conditions are direct consequences of Lemma 2.4.1. Another variant of VPT tree is **VP**^{sb} tree [87]. This tree comes out from the VP^s tree where a leaf node can store more that one object and forms the whole bucket.

Figure 3.4(b) show the situation when the search algorithm must enter both subtrees. Thus, no object can be excluded from examining.

In this case, the performance of searching deteriorates. In [12], this issue is handled by extending the VPT tree to an k-ary tree. This variant uses $k - 1$ thresholds (percentiles) $d_m^1, \cdots, d_m^{k-1}$ instead of one median d_m to partition the dataset into k spherical cuts, where $k > 2$. These modified trees are called **Multi-Way Vantage Point Trees** (mw-VPTs) and presented in [12, 13]. Unfortunately, the experiments reveal that the performance of mw-VPT trees is not very satisfactory because the spherical cuts become too thin for high-dimensional domains [12]. Consequently, it leads to access more branches of the tree during the search elaboration. If a search path descents down to i of k children of a node then i distance computations are evaluated at the next level because the distance between the query object q and each pivot of the accessed children has to be measured. This is caused by the fact that the VPT keeps a different pivot for each internal node at the same level.

Another extension of VPT trees called **Optimistic Vantage Point Tree** is presented in [23]. Authors formulate algorithms for nearest neighbor queries and exhaustively test the performance on an image database.

The VPT trees take $\mathcal{O}(n)$ space. The construction time for balanced tree is $\mathcal{O}(n \log n)$. The search time complexity is $\mathcal{O}(\log n)$. The authors of [87] claim that this is true only for very small query radii – too small to become interesting. The construction time of mw-VPT trees is $\mathcal{O}(n \log_k n)$ distance computations. The space complexity is the same, i.e. $\mathcal{O}(n)$. Similarly, the search time complexity is $\mathcal{O}(\log_k n)$.

3.1.5 Multi Vantage Point Tree

The **MVPT** (Multi Vantage Point Tree) [12, 13] is the extension of the VPT tree. The motivation behind the MVPT is to decrease the number of pivots used to construct a tree because distances between a query object and pivots form significant search costs. FQT trees also introduces a similar technique for reducing these costs, see Section 3.1.2. Another interesting idea, which contributes to the shrinkage of the number of distance evaluated during the search is based on storing distances between pivots and objects in leaf nodes, notice that these distances are computed during the tree construction time. This extra information kept in leaves is then exploited by a sort of filtering algorithm, explained in detail in Section 2.6. The filtering algorithm dramatically decreases the number of distance computations needed to answer similarity queries.

The MVPT uses two pivots in each internal node, instead of one in VPT. Thus, each internal node can be viewed as two levels of VPT collapsed into one node. However, there is one crucial difference: VPT trees use different pivots at the lower level, on contrary, MVPT applies only one, that is, all children at the lower level use the same pivot. This allows to have less pivots whereas the number of partitions is preserved. Figure 3.6 depicts a situation where a VPT tree is collapsed into a MVPT tree. We can easily observe that some sets are partitioned using pivots that are not members of the sets, this does not occur in VPTs. In this case, each pivot has two partitions (fanouts) that implies that the number of fanouts of MVPT's node is 2^2. A pivot can generally partition into m subsets and MVPT tree can employ k pivots in each internal node. As a result, each non-leaf node has m^k partitions. Moreover, each object in the leaf node has associated a list of distances to the first p pivots, which are used for additional pruning in the query time.

The space complexity is $\mathcal{O}(n)$ since no objects are duplicated, objects chosen as pivots are stored in internal nodes and are not duplicated into leaves. However, MVPT needs some extra space since we keep p precomputed distances for each object in leaves. The construction time complexity is $\mathcal{O}(nk \log_{m^k} n)$, where $\log_{m^k} n$ is the height of the balanced tree. The search complexity is $\mathcal{O}(k \log_{m^k} n)$ but it is true for very small query radii. In the worst case, the search complexity is $\mathcal{O}(n)$. The authors of [13] show experimentally that MVPT trees outperform VPT trees, while a larger improvement is obtained by using more pivots in internal nodes than increasing m, the number of fanouts. The largest performance boost is achieved by storing more precomputed distances in leaves.

3.1.6 Excluded Middle Vantage Point Forest

The **VPF** (Excluded Middle Vantage Point Forest), presented in [88], is another member of the family of VPT trees. The motivation for VPF is that VPT [87] and k-d trees [7, 8, 9] have sublinear search time for nearest neighbor queries but the performance depends not only

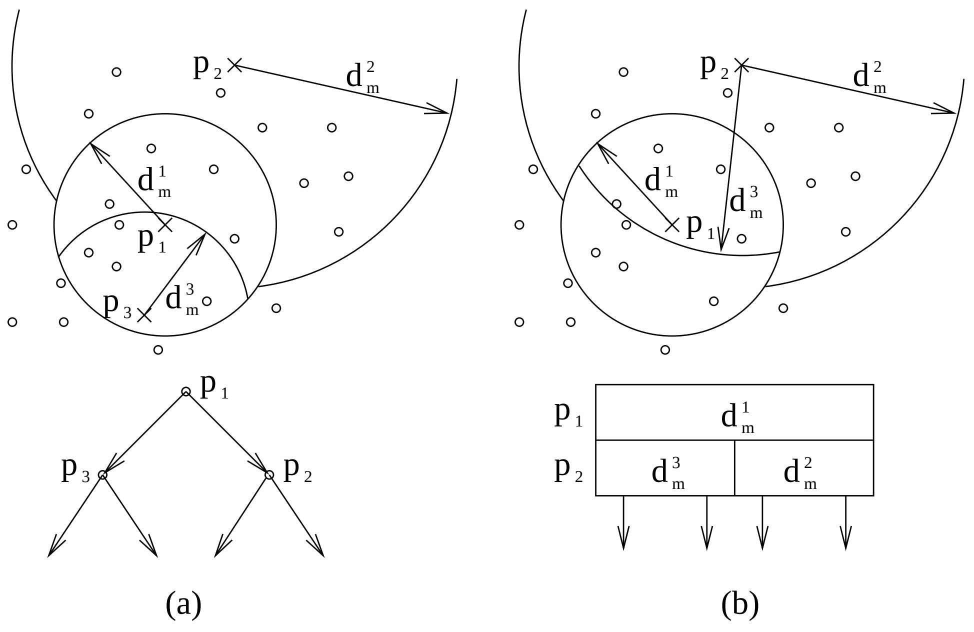


Figure 3.6: The difference between VPT and MVPT structures, VPT uses three pivots for partitioning to four sets (a), MVPT collapses two levels of VPT into one node and uses two pivots only (b).

on the dataset but also on the distribution of queries. The VPF structure supports the worst-case sublinear search time for queries with a fixed radius up to ρ – the performance does not depend on the query distribution. The VPF introduces a new concept of excluding the middle distances. The structure uses a modified ball partitioning technique that eliminates the middle distances from partitioning. The new principle is defined as follows:

$$bp_\rho(o,p,d_m,\rho) = \begin{cases} 0\ if\ d(o,p) \leq d_m - \rho \\ 1\ if\ d(o,p) > d_m + \rho \\ 2\ otherwise \end{cases} \tag{3.2}$$

For convenience, Figure 3.7(a) depicts an example of bp_ρ function. The function divides a dataset into two sets S_0, S_1 and an exclusion set S_2 which is expelled from the partitioning process. A binary tree is built by recursive repartitioning of the set S_0 and S_1. All resulting

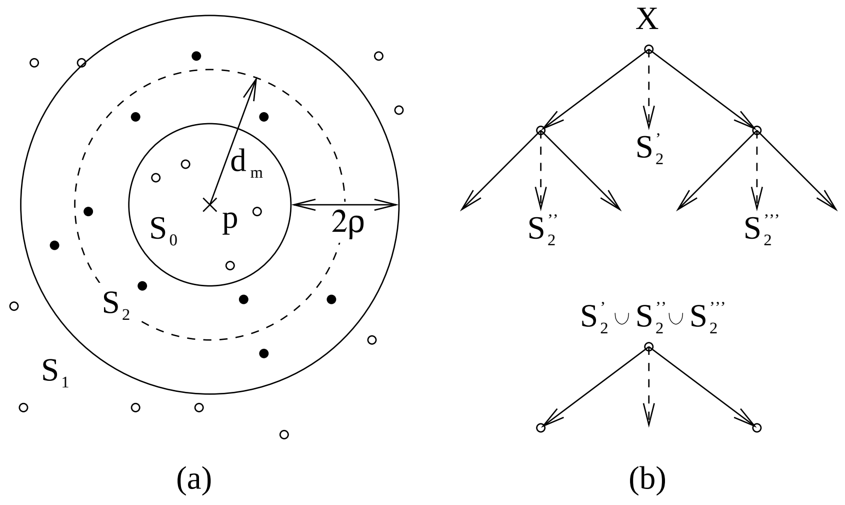


Figure 3.7: An example of bp_ρ function in figure (a), all filled points form the exclusion set S_2, which is then eliminated from the process of partitioning. VPF consisting of two trees is presented in (b), the second tree is built from all exclusion sets of the first tree.

exclusion sets S_2 are used to create another binary tree. This procedure is repeated and a forest of VPT trees is produced. Figure 3.7(b) provides an example of VPF. The first tree is built on the dataset X. All exclusion sets of the first tree, i.e. $\{S_{2,1}, S_{2,2}, S_{2,3}\}$, are organized in the second tree. This process continues until the exclusion sets are not empty.

The idea of excluding the middle distances eliminates the examination of more than one branch if the query radius is less than or equal to ρ. The next tree is searched if and only if the excluded area must be visited. The search algorithm which enters only one branch (left or right) is correct since every pair (x, y) of objects, such that x belongs to the left branch and y lies in the right one, is distant at least 2ρ, that is, $d(x, y) > 2\rho$.

Proof: We have to prove that $\forall x, y \in X$, $bp_\rho(x, p, d_m, \rho) = 0$, $bp_\rho(y, p, d_m, \rho) = 1 : d(x, y) > 2\rho$. If we consider the definition of bp_ρ function in Equation 3.2, we can write $d(x, p) \leq d_m - \rho$ and $d(y, p) > d_m + \rho$. Since the triangle inequality among x, y, p holds, we get $d(x, y) + d(x, p) \geq d(y, p)$. If we combine these inequalities and simplify the expression, we get $d(x, y) > 2\rho$. ■

The VPF is linear in space $\mathcal{O}(n)$ and the construction time is $\mathcal{O}(n^{2-\alpha})$ where $\mathcal{O}(n^{1-\alpha})$ is the number of trees in the VPF forest. Similarity queries are answered in $\mathcal{O}(n^{1-\alpha} \log n)$ distance computations. Having a parallel environment with $\mathcal{O}(n^{1-\alpha})$ processors the search complexity is logarithmic $\mathcal{O}(\log n)$. The parameter $0 < \alpha < 1$ depends on ρ, the dataset, and the distance function. Unfortunately, to achieve greater value of α the parameter ρ has to be quite small.

3.2 Generalized Hyperplane Partitioning Methods

In this stage, methods based on an orthogonal approach to the ball partitioning are concerned. Specifically, we focus on Bisector trees and their variants Monotonous Bisector Trees and Voronoi Trees. Next, we discuss properties of Generalized Hyperplane Trees and Geometric Near-neighbor Access Trees. All these techniques share the same idea, that is, they apply the hyperplane partitioning.

3.2.1 Bisector Tree

Probably the first solution which uses hyperplane partitioning, see Section 2.5, was **BST** (Bisector Tree), proposed in [54]. BST is a binary tree recursively build over the dataset X as follows. Two pivots p_1, p_2 are selected at each node and hyperplane partitioning is applied, that is, objects near the pivot p_1 form the left subtree while objects closer to the pivot p_2 go to the right subtree. For each of the pivots the *covering radius* is established and stored in the node. The covering radius is the maximum distance between the pivot and any object in its subtree. The search algorithm enters a tree-branch if $d(q, p_i) - r$ is not greater than the covering radius r_i^c of p_i. Thus, we can prune a branch if the query does not intersect the ball centered in p_i with the covering radius r_i^c. The pruning condition $d(q, p_i) - r \leq r_i^c$ is correct since its modification $d(q, p_i) - r_i^c \leq r$ is a direct consequence of lower bound of Lemma 2.4.2 with substitutions $r_l = 0$ and $r_h = r_i^c$, besides $d(q, o)$ is upper-bounded by the query radius r (from the definition of the range query).

A variant of BST is proposed in [73, 72] and called **Monotonous Bisector Tree** (MBT). The idea behind this structure is that one of pivots of each internal node, except the root node, of course, is inherited from its parent node. In other words, pivots representing the left and right subtree are copied to the corresponding child nodes. This technique leads to usage of less pivots which inherently brings less distance computations during the search algorithm elaboration[1].

The BST trees are linear in space $\mathcal{O}(n)$ and they need $\mathcal{O}(n \log n)$ distance computations to construct the tree. Search complexity was not analyzed by authors.

An improvement of BST trees called **VT** (Voronoi Tree) is proposed in [31]. VT uses 2 or 3 pivots in each internal node and has the property that the covering radii are reduced as we move downwards in the tree. This provides better packing of objects in subtrees. Authors of [71] show that balanced VT trees can be obtained by assimilating similar insertion algorithm used in B-Trees [29]. This technique is exploited in M-Trees [25], see Section 3.3.

3.2.2 Generalized Hyperplane Tree

The Generalized Hyperplane Tree (**GHT**) proposed in [83] is nearly the same as BST trees in construction. GHT recursively partitions the given dataset by applying the generalized hyperplane partitioning principle. The difference is that it does not use covering radii as a pruning criterion during the search operation, instead of it, GHT uses the hyperplane between two pivots p_1, p_2 to decide which subtrees must be visited. Figure 3.8 depicts an example of GHT tree, in (a) the partitioning is presented and a range query is specified. The corresponding tree structure is shown in (b). At the search time, we traverse the left subtree if $d(q, p_1) - r \leq d(q, p_2) + r$. The right subtree is visited if $d(q, p_1) + r \geq d(q, p_2) - r$ holds. Again, note that it is possible to enter both the subtrees. The first inequality is from Lemma 2.4.4 and from the fact that $d(q, o) \leq r$, i.e. the constraint of range query. The second inequality is based on the same prerequisites, however, Lemma 2.4.4 is used the other way around, that is,

[1]Assuming that the distance to the pivot in the parent node is not computed again in the child node.

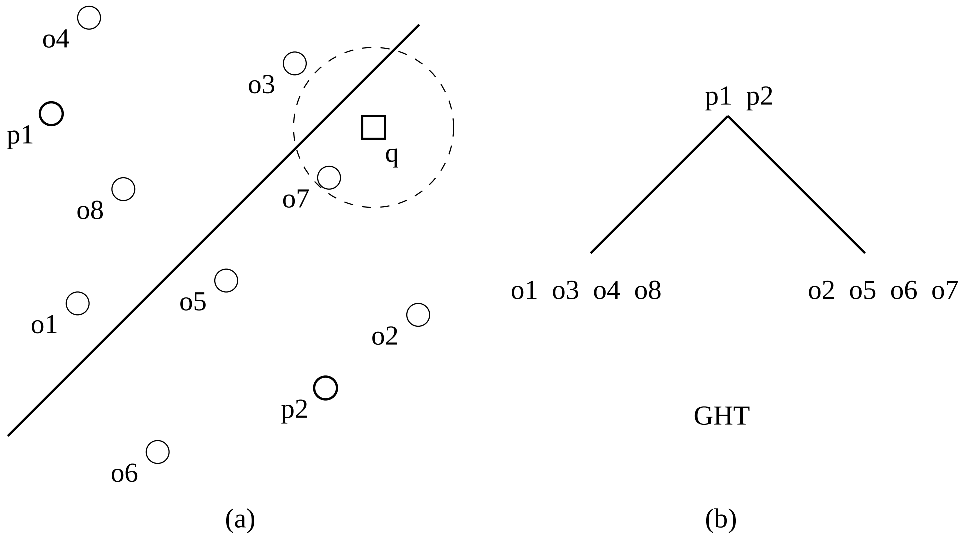


Figure 3.8: Generalized Hyperplane Tree (GHT): (a) hyperplane partitioning and both the partitions must be accessed to answer the given range query, (b) the corresponding structure of the tree.

the assumption on position of o is $d(o, p_1) \geq d(o, p_2)$.

A modification of GHT, which adopts the idea of reusing one pivot from the parent node, applied in MBT trees, is presented in [15].

The space complexity of GHT trees is $\mathcal{O}(n)$ and they need $\mathcal{O}(n \log n)$ distance computations to construct the tree, it is the same as BST trees. Unfortunately, the search complexity was not analyzed by authors. Uhlmann in [83] argues the GHT trees could work better than VPT trees in high dimensions.

3.2.3 GNAT

The essential idea of GHT is extended in the next work [14] that presents **GNAT** (Geometric Near-neighbor Access Tree). GNAT is a generalization of GHT that uses m pivots in each internal node. In particular, we pick a set of pivots $P = \{p_1, \ldots, p_m\}$ and split the dataset X into $S_1, \ldots, S_m$ subsets depending on the shortest distance to a pivot in P. In other words, for any object $o \in X - P$, o is a member of the set S_i iff $d(p_i, o) \leq d(p_j, o)$ for all $j = 1, \ldots, m$. Thus, applying

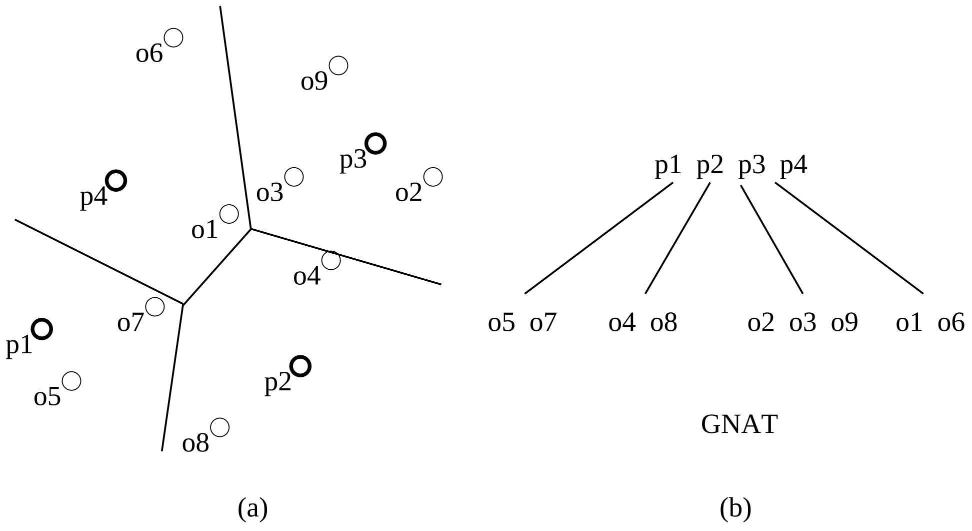


Figure 3.9: An example of partitioning used in the Geometric Near-neighbor Access Tree in (a) and the corresponding tree in (b).

this procedure recursively we build an m-ary tree. Figure 3.9 shows a simple example of the first level of GNAT. Remark the relationship between this idea and a Voronoi-like partitioning of vector spaces [2]. Each subset S_i corresponds to a cell in the Voronoi diagram, Brin [14] calls this cell as the Dirichlet domain. Brin also suggests that the parameter m can vary with the level in the tree and describes that the number of children (m) depends proportionally on the number of data objects allocated in the node.

Besides the m-ary partitioning principle, GNAT also stores information about distances between pivots and objects in subtrees. This enables further pruning during the search and implies that the range search algorithm is quite different from GHT trees. In each internal node, a table $m \times m$ consisting of ranges of distances is stored. In particular, ranges store the minimum and maximum of distances between each pivot p_i and each subset S_j. Formally, the range $[r_l^{ij}, r_h^{ij}]$, where $i, j = 1, \ldots, m$, is defined as follows:

$$r_l^{ij} = \min_{o \in S_j \cup \{p_j\}} d(p_i, o),$$

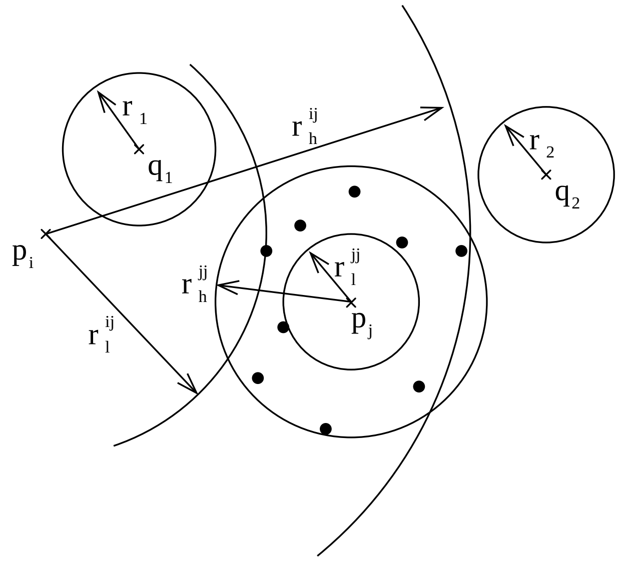


Figure 3.10: Example of the pruning effect of ranges. The query $R(q_1, r_1)$ needs only the range $[r_l^{ij}, r_h^{ij}]$ to eliminate the subtree of p_j. As for the query $R(q_2, r_2)$, the range $[r_l^{ij}, r_h^{ij}]$ is not sufficient for skipping the subtree of p_j and the range $[r_l^{jj}, r_h^{jj}]$ is also used and p_j is then eliminated.

$$r_h^{ij} = \max_{o \in S_j \cup \{p_j\}} d(p_i, o).$$

Figure 3.10 illustrates two ranges, the first $[r_l^{ij}, r_h^{ij}]$ is defined for pivots p_i and p_j while the second $[r_l^{jj}, r_h^{jj}]$ for pivot p_j itself.

The range search algorithm for a range query $R(q, r)$ specified in [14] proceeds in a depth-first manner. In each internal node n, the distances between q and pivots of n are computed gradually and some subtrees can be eliminated in each step. After all distances from q to pivots have been computed, the algorithm visits all subtrees that were not eliminated. In detail, the procedure applied in each internal node can be described in the following steps. We start with the set P consisting of all pivots of the examined node. Next, we pick one pivot p_i from P repeatedly (but we do not pick the same pivot twice) and compute the distance $d(p_i, q)$. If $d(p_i, q) \leq r$ holds the pivot p_i is reported to the query result. Afterwards, for all $p_j \in P$ we remove p_j from P if $d(q, p_i) - r > r_h^{ij}$ or $d(q, p_i) + r < r_l^{ij}$. Those inequalities are direct consequences of the lower bound $max\{d(q,p_i) - r_h^{ij}, r_l^{ij} - d(q,p_i)\} \leq r$ of Lemma 2.4.2 with $d(q,o) \leq r$. When all pivots in P have been examined, the subtrees of the node n that correspond to remaining pivots in P are visited. Note that a pivot p_j can be discarded from P before its distance to q is evaluated. Figure 3.10 depicts the situation where two range queries $R(q_1,r_1), R(q_2,r_2)$ are given. Only the range $[r_l^{ij}, r_h^{ij}]$ is sufficient for the query $R(q_1,r_1)$ to discard p_j. However, the other query needs additional range, that is $[r_l^{jj}, r_h^{jj}]$, to prune the subtree around p_j.

The space complexity of GNAT index structure is $\mathcal{O}(nm^2)$ since tables consisting of m^2 elements are stored in each internal node. GNAT is built in $\mathcal{O}(nm \log_m n)$. The search complexity was not analyzed by authors but experiments [14] reveal the fact that GNAT outperforms GHT and VPT.

3.3 M-Tree

All indexing methods described earlier are either static, unbalanced, or both. Thus, they are not suitable in dynamic environments, where data are subject of permanent changes, or in the area of large data warehouses, where disk-based techniques are necessary. Authors of [25] present **MT** (M-Tree) that is designed as a dynamic and balanced index structure with the capability of disk storages. The idea of R-Trees [47], obviously inherited from B-Trees, i.e. tree grows from its leaves to the root, is also utilized in M-Trees. This conception provides the balanced tree structure regardless how many insertions or deletions were made and has impacts on the search performance.

In general, MT behaves like the R-Tree, that is, all objects are stored or referenced in leaf nodes while internal nodes keep pointers to nodes at the next lower level along with additional information about their subtrees. Recall that R-Trees store minimum bounding rectangles in non-leaf nodes that cover subtrees. In general metric spaces, we cannot define bounding rectangles, hence, MT trees use a pivot and a covering radii to form a covering ball. In MT, pivots play similar roles as in GNAT, however, unlike GNAT, all objects are stored in leaves. An object can be referenced multiple times in the tree, that is, once in a leaf node and once or more in internal nodes as a pivot.

In details, each node in MT consists of m entries. An entry of a non-leaf node is a tuple $(p, r^c, d(p, p^p), ptr)$, where p is a pivot, r^c is the corresponding covering radius, p^p is the parent pivot of p (superscript ^{p} denotes parent), $d(p, p^p)$ is the distance from p to the parent pivot p^p, and finally, ptr is the pointer to the child node of pivot p. All objects o in the subtree rooted in ptr are within the distance r^c from p, i.e. $d(o, p) \leq r^c$. In the next, we will show that distances to parent objects enhance the pruning effect. Similarly, an entry of a leaf node consists of a tuple $(o, d(o, o^p))$, where o is a database object, $d(o, o^p)$ is the distance between o and its parent object (pivot). Figure 3.11 depicts MT consisting of three levels. Observe that the covering radius of a non-leaf entry (e.g. root entries in the figure) is not necessarily the minimum radius for the objects in the corresponding subtree. The minimum radius is emphasized with a dotted circle. If minimum values of covering radii are used the overlap of entries is minimized, furthermore, this results in a more efficient search. If a bulk-load algorithm [24] is used for building MT the covering radii are set to their minimum values.

MT is a dynamic structure, thus, we can build the tree gradually as new data come. The insertion algorithm looks for the best leaf node to insert a new object o_n and inserts it. The heuristic that finds the suitable leaf works as follows. The algorithm descends along a subtree for which no enlargement of covering radius r^c is needed, i.e. $d(o_n, p) \leq r^c$. If multiple subtrees with this property exist, the one for which object o_n is closest to its pivot is chosen. Figure 3.11 shows the situation where the object o_{11} can be inserted into two subtrees around pivots o_7, o_2, finally, o_{11} is inserted into the subtree of o_7. If there is no pivot for which no enlargement is need the algorithm's choice is to minimize the increase of the covering radius. At last, the new object o_n is inserted into selected leaf node. The insertion into the leaf may cause overflow, the leaf splits and a new pivot is selected. Note that the overflow may propagate to the root node and the tree grows one level in the bottom-up fashion. A number of heuristics for choosing the most suitable leaf node and for splitting nodes is considered in [25].

The range search algorithm for $R(q, r)$ traverses the tree in a depth-first manner, initiated in the root. During the search we make

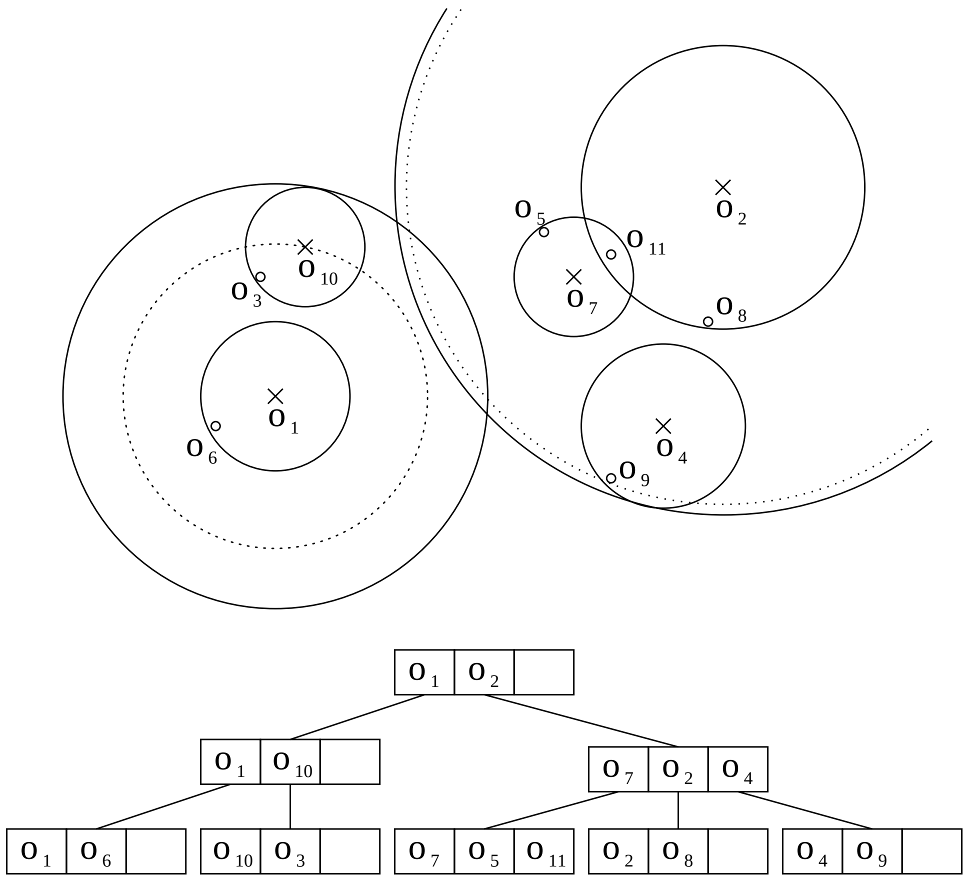


Figure 3.11: Example of M-Tree (MT) that consists of 3 levels. Above, the 2-D representation of partitioning where pivots are denoted with crosses and the circles around pivots correspond to values of covering radii. Observe that balls around pivots can intersect. The dotted circles represent the minimum values of covering radii. If they are used no overlap is achieved. Below, the corresponding tree structure is shown.

use of all stored distances to parent objects. Assume that the current node n is an internal node, we consider all entries $(p, r^c, d(p, p^p), ptr)$ of n.

- If $|d(q, p^p) - d(p, p^p)| - r^c > r$ the subtree pointed ptr does not need to be visited and the entry is pruned. This pruning criterion is based on the fact that the expression $|d(q, p^p) - d(p, p^p)| - r^c$ is a lower bound on distance $d(q, o)$, where o is any object in the subtree ptr. Thus, if the lower bound is greater than the query radius r the subtree need not to be visited because no object in the subtree can qualify the range query.

  Proof: We have to prove that the lower bound $|d(q, p^p) - d(p, p^p)| - r^c \leq d(q, o)$ is correct. The lower bound is a consequence of Lemma 2.4.1 and Lemma 2.4.3. Lemma 2.4.1 provides us with the lower and upper bound on the distance $d(q, p)$. If we use p and p^p as o and p in the lemma, respectively, we get the lower bound $|d(q, p^p) - d(p, p^p)| \leq d(q, p)$ and the upper bound $d(q, p) \leq d(q, p^p) + d(p, p^p)$. The distances $d(p, o)$ are in the range $[0, r^c]$ because all objects o in the subtree ptr are within the covering ball established by pivot p with the covering radius r^c. Now, using Lemma 2.4.3 we obtain the lower on the distance $d(q, o)$. If we substitute $r_l = 0$, $r_h = r^c$ and $r'_l = |d(q, p^p) - d(p, p^p)|$, $r'_h = d(q, p^p) + d(p, p^p)$ we get the lower bound $\max\{|d(q, p^p) - d(p, p^p)| - r^c, 0 - (d(q, p^p) + d(p, p^p)), 0\} \leq d(q, o)$. The expression $0 - (d(q, p^p) + d(p, p^p))$ is always negative, thus, the lower bound is $|d(q, p^p) - d(p, p^p)| - r^c \leq d(q, o)$. ■

- If $|d(q, p^p) - d(p, p^p)| - r^c \leq r$ holds we cannot avoid computing the distance $d(q, p)$. Having the value of $d(q, p)$ we can still prune some branches using the following criterion: $d(q, p) - r^c > r$. This pruning criterion is a direct result of the lower bound in Lemma 2.4.2 with the substitution $r_l = 0$ and $r_h = r^c$ (the lower and upper bounds on the distance $d(p, o)$).

- All non-pruned entries are recursively searched.

Leaf nodes are processed similarly. Each entry $(o, d(o, o^p))$ is tested on the pruning condition $|d(q, o^p) - d(o, o^p)| > r$. If it holds the entry can be safely pruned. This pruning criterion is the lower bound in Lemma 2.4.1. If the entry cannot be discarded the distance $d(q, o)$ is evaluated and the object o is reported if $d(q, o) \leq r$. Note that in all three steps where pruning criteria are defined we discard some entries without computing distances of the corresponding objects.

The algorithm for k-nearest neighbor queries is based on the range search algorithm and instead of the query radius r the distance to the k^{th} nearest neighbor is used, for details see [25]. In [27], authors extend MT to support complex similarity queries which are defined through a generic language. In other words, complex queries consist of more than one similarity predicate. An application of complex similarity queries is relevance feedback techniques.

The space complexity is $\mathcal{O}(n + m * N)$ distances, n is the number of distances stored in leaves, N is the number of internal nodes, each node has capacity of m entries. The claimed construction complexity is $\mathcal{O}(n(m \dots m^2) \log_m n)$ distances. The search complexity was not analyzed by authors of [25] but in [26] a cost model of MT is presented. However, the experiments in [25] show that MT is resistant to the dimensionality of the space and is competitive against R*-trees.

3.3.1 Slim Trees

In [53], **ST** (Slim Tree) is proposed. ST is an extension of M-Tree that speeds up insertion and node splitting algorithms while also improving the storage utilization.

The proposed splitting algorithm is based on the minimum spanning tree (MST) [59], which has been successfully used in clustering. Assume a set of n objects of a node which has to be split we consider a fully connected graph consisting of n objects (acting as vertices) and $n * (n - 1)$ edges. The weight of an edge is the distance between the edge's vertices. The algorithm proceeds the next steps:

1. build the minimum spanning tree on the full graph,
2. delete the longest edge,
3. resulting two components of the graph form the new nodes,
4. choose a pivot of each group – the object whose maximum distance to all other objects in the group is the shortest.

Unfortunately, the algorithm does not guarantee the minimal occupation of each node, which is often $\lceil m/2 \rceil$ where m is the node's capacity (maximum number of entries). However, the algorithm tries to minimize the overlap of nodes.

As the post-processing step, authors [53] presents Slim-down algorithm that even reduces the overlap between nodes and increases the storage utilization. The most promising split strategy claimed by authors of M-Tree [25] is minMax splitting algorithms. This algorithm has complexity $\mathcal{O}(n^3)$, where n is the number of objects in a node to be split. On contrary, the proposed MST-splitting method has $\mathcal{O}(n^2 \log n)$ requirements.

Very recently, another extension of building algorithms for M-Trees was proposed [80]. The authors propose two techniques, a multi-way insertion algorithm and a generalized slim-down algorithm. The multi-way method increases node utilization thus it leads to smaller index files. The slim-down algorithm of ST (Slim Tree) is generalized and causes better query performance for both the range and nearest neighbor queries. A combination of these approaches is also considered.

3.4 Spatial Approximation Tree

Previously presented indexes use some partitioning principle to recursively divide the data space into partitions. GHT and GNAT are inspired by the Voronoi-like partitioning. In this section, we introduce **SAT** (sa-tree – Spatial Approximation Tree) [67, 69]. SAT is based on Voronoi diagrams, however, it does not create a hierarchical structure which recursively partitions data into two or more Voronoi cell-like regions. In contrast, SAT trees try to approximate the structure of the *Delaunay graph*. Given a Voronoi diagram, a Delaunay graph defined in [69] is a graph where each node represents one cell of the Voronoi diagram and nodes are connected with an edge if the corresponding Voronoi cells are directly neighboring. In other words, the Delaunay graph is a representation of relations between cells in the Voronoi diagram. Next, we will use the term object for a node of Delaunay graph and vice versa. The search algorithm for the nearest neighbor of a query object q starts with an arbitrary object (node in the Delaunay graph) and proceeds to a neighboring object closer to q as long as it is possible. If we reach an object o where all neighbors of o are further from q than o we have found the nearest neighbor of q – the object o. The correctness of this simple algorithm is guaranteed because of the property of Delaunay graphs that if q is closer to an object o than to any of the neighbors of o then o is the closest object to q. Unfortunately, it is possible to show that without more information about given metric space $\mathcal{M} = (\mathcal{D}, d)$, the knowledge of distances among objects in a finite set $X \subseteq \mathcal{D}$ does not uniquely determine the Delaunay graph for X, for further details see [69, 51]. Thus, the only way to ensure that the search procedure is correct is to use a complete graph, that is, the graph containing all edges between all pairs of objects in X. However, such a graph is not suitable for searching because the decision of which edge should be traversed from the starting object requires computing distances from the query to all remaining objects in X. In fact, this is the same as linear scan of all objects in the database and from the search point of view it is useless.

SAT is defined for a dataset X as follows. An arbitrary object p is selected as a root of the tree and the smallest possible set $N(p)$ of all its neighbors is determined. For every neighbor, a child node is defined and the closest objects to the child are structured in the same way in this subtree. The distance to the furthest object o from p is also stored in each node, i.e. for the root node, $\max_{o \in X} d(p, o)$. In established terminology, it is the covering radius r^c. The set $N(p)$ of neighbors of p is defined as:

$$o \in N(p) \Leftrightarrow \forall o' \in N(p) \setminus \{o\} : d(o, p) < d(o, o').$$

The intuition behind this definition is that, for a valid set $N(p)$ (not necessarily the smallest), each object of $N(p)$ is closer to p than to any other object in $N(p)$ and all objects in $X \setminus N(p)$ are closer to an object in $N(p)$ than to p. Figure 3.12(b) shows an example of SAT built on a dataset depicted in Figure 3.12(a). As the root node the object o_1 is selected, the set of neighbors for o_1 is $N(o_1) = \{o_2, o_3, o_4, o_5\}$. Note that object o_7 cannot be included in $N(o_1)$ since o_7 is closer to o_3 than to o_1.

Unfortunately, the construction of minimum set $N(p)$ is very expensive. Navarro in [69] argues that it is an NP-complete problem

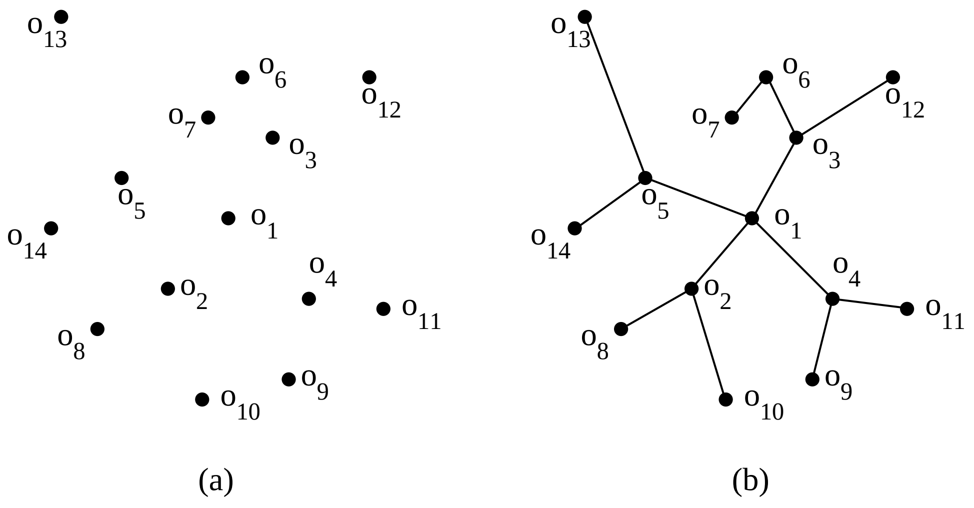


Figure 3.12: SAT tree is built over a dataset in (a). In (b), SAT tree is depicted where o_1 is selected as root.

and proposes a heuristic that constructs the possible set of neighbors, nonetheless, not minimal. The heuristic starts with an object p and a set $S = X \setminus \{p\}$, $N(p)$ is initially empty. First, we sort the members of S by their distance to p. Then, we gradually pick an object o from S and add it to $N(p)$ if is is closer to p than to any other object in $N(p)$. As a result, we have a suitable set of neighbors.

The range search algorithm for the query $R(q, r)$ starts at the root node and traverses the tree visiting all non-discardable subtrees. Specifically, at the node p, we have the set of all neighbors $N(p)$. First, we locate the closest object $o_c \in N(p) \cup \{p\}$ to q. Next, we can discard all subtrees $o_d \in N(p)$ such that

$$d(q, o_d) > 2r + d(q, o_c). \tag{3.3}$$

Proof: The pruning criterion is a consequence of Lemma 2.4.4 with $p_1 = o_d$ and $p_2 = o_c$. In particular, we get $\max\{\frac{d(q,o_d)-d(q,o_c)}{2}, 0\} \leq d(q, o)$ from Lemma 2.4.4. Providing that $d(q, o) \leq r$ (range query constraint) and q is closer to o_c than o_d we have $\frac{d(q,o_d)-d(q,o_c)}{2} \leq r$ from the lower bound of Lemma 2.4.4. We can prune the branch o_d if $\frac{d(q,o_d)-d(q,o_c)}{2} > r$, that is, $d(q, o_d) > 2r + d(q, o_c)$. ■

The closest object o_c to q is selected because we want to maximize the lower bound of Lemma 2.4.4. When the current node is not the root of tree we can even improve the pruning effect. Figure 3.13 depicts a sample of SAT tree with the root t, the current node is p and all its neighbors are p_1, p_2, p_3, the query object q is specified. Above presented algorithm selects p as the closest object to q, even though, the object s_2 is closer. If we choose s_2 as the closest object we further maximize the effect of pruning condition. However, it requires to modify the procedure of picking the closest object as follows: we select o_c from all ancestors of p and their neighbors to get the closest object, i.e. $o_c \in \bigcup_{o \in A(p)} (N(o) \cup \{o\})$, where $A(p)$ consists of all ancestors of p and their neighbors (in the figure, $A(p)$ consists of the root node t and all its neighbors including p). Finally, the covering radius r^c of each node is used to further reduce the search costs. We do not visit a node p if $d(q,p) > r^c + r$. This expression is derived from the lower bound in Lemma 2.4.2 with $r_l = 0, r_h = r^c$ and from the fact that $d(q,o) \leq r$. We must remark that the search algorithm is correct and returns all qualifying objects regardless the strategy of selecting the closest object o_c. In other words, the strategy only influences the efficiency of pruning, see Equation 3.3.

The tree is built in $\mathcal{O}(n \log n / \log \log n)$, takes $\mathcal{O}(n)$ space and the search complexity is $\Theta(n^{1-\Theta(1/\log \log n)})$. For further details, see [69, 67, 51]. SAT is designed as a static structure a dynamic version of SAT is presented in [70].

3.5 Distance Matrix Methods

Finally, we discuss other techniques than presented in previous sections. These methods are based on usage of matrices of distance between objects in metric space. Matrices are then used in search algorithm to prune not qualifying objects. Specifically, we present Approximating and Eliminating Search Algorithm and its linear variant. We also mention other modifications or improvements such as TLAESA, ROAESA, and Spaghettis. At last, we focus on mapping-based approaches that transform a metric space into a vector space and discuss related issues.

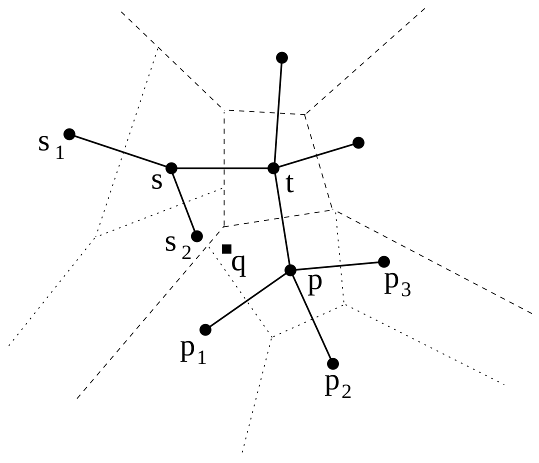


Figure 3.13: A sample of SAT with the root t and the query object q are provided. At the current node p, the range search algorithm selects the object p as the closest object to q while s_2 is closer to q than p. An improved variant of search algorithm selects the closest object from all ancestors and their neighbors, i.e. s_2 is selected instead of p. The dashed lines represent the boundaries between Voronoi cell of the first level of SAT tree. The dotted lines depict the same for the second level.

3.5.1 AESA

AESA (Approximating and Eliminating Search Algorithm) presented in [75, 84] uses a matrix of distances between database objects which were computed during the creation of the AESA structure. The structure is simply a matrix $n \times n$ consisting of distances between all pairs of n database objects. Particularly, due to symmetry property of metric function only a half of the matrix below the diagonal is stored, that is, $n(n-1)/2$ distances. In comparison with previous methods, every object in AESA plays the role of pivot.

The search operation for range query $R(q, r)$ (for the nearest neighbors queries works similarly) picks an object p at random and uses it as a pivot. The distance from q to p is evaluated and is used for pruning some objects. We can prune object o if $|d(p, o) - d(q, p)| > r$, that is, the lower bound in Lemma 2.4.1 is greater than the query radius r. Next, the algorithm chooses next pivot among all non-discarded objects up to now. The choice of pivot is influenced by the lower bound $|d(p,o)-d(q,p)|$ since we want to maximize the pruning effect we have to maximize the lower bound that results in the choice of the closest object p to q [75]. The new pivot is used in the pruning condition for further elimination of some non-discarded objects. This process is repeated until the set of non-discarded objects is not small enough. Finally, the remaining objects are checked directly with q, i.e. distances $d(q,o)$ are evaluated and objects satisfying $d(q,o) \leq r$ are reported.

According to experiments presented in [84], AESA performs an order of magnitude better than other competing methods and is argued that it has a constant query time with respect to the size of database ($\mathcal{O}(1)$). This superior performance is obtained at the expense of quadratic space complexity $\mathcal{O}(n^2)$ and quadratic construction complexity. The extra CPU time is spent in scanning the matrix, the extra time is from $\mathcal{O}(n)$ up to $\mathcal{O}(n^2)$, however, we should note that one distance computation is much more expensive than one scan in matrix. Although the performance is promising, AESA is applicable only for small data sets of at most a few thousand objects. On contrary, if range queries with large radii or nearest neighbors queries with high k are specified AESA tends to to require $\mathcal{O}(n)$ distance computations, the same as a trivial linear scan.

3.5.2 LAESA

The main drawback of AESA approach being quadratic in space is solved in **LAESA** structure (Linear AESA) [64, 65]. This method alleviates this disadvantage by choosing a fixed number k of pivots whose distances to other objects are kept. Thus, the distance matrix is $n \times k$ rather than $n(n-1)/2$ entries in AESA. Usage of only a relatively small number of pivots needs an algorithm for selecting appropriate pivots, which can advance pruning during the search. In [65], the pivot selection algorithm attempts to choose pivots far away from each other as possible.

The search procedure is nearly the same as in AESA except that some objects are not pivots. Thus, we cannot choose next pivot from non-discarded object up now because we might have eliminated some pivots. First, the search algorithm eliminates objects using all pivots. Next, all remaining objects are directly compared to the query object q. More details can be found in [51] as well as other search algorithms (nearest neighbor queries, ranking queries).

The space complexity of LAESA and construction time are $\mathcal{O}(kn)$. The search requires $k + \mathcal{O}(1)$. The extra CPU time is reduced by a modification called **TLAESA** and introduced in [63]. TLAESA builds a GHT tree-like structure using the same k pivots and the extra CPU time is between $\mathcal{O}(\log n)$ and $\mathcal{O}(kn)$. AESA and LAESA approaches are compared in [74].

3.5.3 Other Methods

A similar structure to LAESA is presented in [79] which also stores kn distances in a matrix $n \times k$. However, the search procedure is slightly different. The objects in database $(o_1, \dots o_n)$ are sorted by distance to the first pivot p_1. The search starts with the object o_i whose distance from p_1 is nearly the same as $d(p_1, q)$, i.e. $|d(p_1, o_i) - d(q, p_1)|$ is minimum where $i = 1..n$, note that this is the lower bound in Lemma 2.4.1. The object o_i is checked against all pivots whether the pruning condition $|d(p_j, o_i) - d(q, p_j)| > r$ is valid for any p_j, $j = 1..k$. If o_i cannot be eliminated the distance $d(q, o_i)$ is computed. The search continues with objects $o_{i+1}, o_{i-1}, o_{i+2}, o_{i-2}, \dots$ until the pruning condition $|d(p_1, o_i) - d(q, p_1)| > r$ is valid.

Another improvement of LAESA method is **Spaghettis**, introduced in [19]. This approach also stores kn distances between all n objects and k pivots. Each array of distances to a pivot is sorted according to the distance. Two objects o_i, o_j can have different order in two arrays because their distances to the corresponding pivots differ (e.g. $d(p_1, o_i) < d(p_1, o_j)$ and $d(p_2, o_i) > d(p_2, o_j)$). Thus, we have to store permutations of objects with respect to the preceding array. During the range search, k intervals are defined $[d(q, p_1) - r, d(q, p_1) + r], \dots, [d(q, p_k) - r, d(q, p_k) + r]$. All objects qualifying the query belong to the intersection of all these intervals. Each object in the first interval is checked whether it is a member of other intervals, at this point, the stored permutations are used for traversing through arrays of distances. Finally, non-discarded objects are directly compared with the query object directly. The extra CPU time is reduced to $\mathcal{O}(k \log n)$.

Both AESA and LAESA have an overhead of $\mathcal{O}(n)$ in terms of computations other than distance evaluations (i.e. searching the matrix). In [85], a technique called **ROAESA** (Reduced Overhead AESA) reduces such the cost by using some heuristics to eliminate some unneeded traversals of matrix. However, this technique is applicable only to nearest neighbor searches the range search algorithm is not accelerated.

LAESA can also be used as a $k-NN$ classifier. The variant of LAESA termed **Ak-LAESA** is presented in [66] and performs faster than a standard $k-NN$ classifier in metric spaces while it preserves all features of LAESA.

3.6 Mapping-Based Approaches

In this section, we focus on mapping based approaches to complete the list of techniques for indexing metric spaces. In particular, these methods are based on transforming the original metric space into a multidimensional vector space. In principle, any multidimensional indexing method can be employed to speed up searching in vector spaces. For further details, see surveys [11, 41] of multidimensional access methods.

The transformation is based on a *mapping* function Ψ that transforms a metric space into a vector space:

$$\Psi : (\mathcal{D}, d) \rightarrow (\mathbb{R}^n, \delta).$$

Typically, δ is an L_p metric (Minkowski distance), the most common are the Euclidean metric L_2 or the Chessboard distance L_∞, which is also used in filtering algorithms, see Section 2.6. All database objects o of the metric space are transformed using $\Psi(o)$ into the vector space, the query object q is also transformed $\Psi(q)$. The rationale for doing so is that we replace usually expensive distance d by much less expensive distance δ.

Numerous methods are used for mapping objects into vector spaces, they can be categorized into the following groups: domain specific methods use some special properties of objects, dimensionality reduction methods [30, 39, 55] are applied on the data sets which are already in vector space but the dimensionality is impractically high for use of spatial indexes, and embedding methods are entirely based on the original distance d and the transformation employ a prepicted set of pivots. Of course, the dimensionality reduction techniques can be applied on results of methods of the other categories when the dimensionality is too high.

The search operation in the projected vector space needs the mapping to be *contractive*, it means that $\delta(\Psi(o_1), \Psi(o_2)) \leq d(o_1, o_2), \forall o_1, o_2 \in \mathcal{D}$. In particular, this property ensures that we do not have false dismissals, i.e. the objects qualifying the query that was falsely eliminated. However, the resulting set must be checked with the original function d in the metric space to filter out false positives. Besides, if the mapping function is *proximity preserving* the nearest neighbor queries can be directly performed in the projected vector space without any additional refinement of the answer. The mapping is proximity preserving if $d(o_1, o_2) \leq d(o_1, o_3) \Rightarrow \delta(\Psi(o_1), \Psi(o_2)) \leq \delta(\Psi(o_1), \Psi(o_3)), \forall o_1, o_2, o_3 \in \mathcal{D}$.

Chapter 4

D-Index

Though the number of existing search algorithms is impressive, see [22], the search costs are still high, and there is no algorithm that is able to outperform the others in all situations. Furthermore, most of the algorithms are implemented as main memory structures, thus do not scale up well to deal with large data files.

The access structure we propose, called D-Index [33], synergistically combines more principles into a single system in order to minimize the amount of accessed data, as well as the number of distance computations. In particular, we define a new technique to recursively cluster data in separable partitions of data blocks and we combine it with pivot-based strategies, so-called filtering techniques, to decrease the I/O costs. In the following, we first formally specify the underlying principles of our clustering and pivoting techniques and then we present examples to illustrate the ideas behind the definitions.

4.1 Clustering Through Separable Partitioning

To achieve our objectives, we base our partitioning principles on a mapping function, which we call a *ρ-split function*, where ρ is a real number constrained as $0 \leq \rho < d^+$. The concept of ρ-split functions was originally published by Yianillos [88] and we modify this approach to our needs. In order to gradually explain the concept of ρ-split functions, we first define a first order ρ-split function and its properties.

Definition: Given a metric space $\mathcal{M} = (\mathcal{D}, d)$, a first order ρ-split function $s^{1,\rho}$ is the mapping $s^{1,\rho} : \mathcal{D} \rightarrow \{0, 1, -\}$, such that for arbitrary different objects $x, y \in \mathcal{D}$, $s^{1,\rho}(x) = 0 \wedge s^{1,\rho}(y) = 1 \Rightarrow d(x, y) >$

2ρ (separable property) and $\rho_2 \geq \rho_1 \wedge s^{1,\rho_2}(x) \neq - \wedge s^{1,\rho_1}(y) = - \Rightarrow d(x,y) > \rho_2 - \rho_1$ (symmetry property). ■

In other words, the ρ-split function assigns to each object of the space $\mathcal{D}$ one of the symbols 0, 1, or $-$. Moreover, the function must hold the separable and symmetry properties. The meaning and the importance of these properties will be clarified later.

We can generalize the ρ-split function by concatenating n first order ρ-split functions with the purpose of obtaining a split function of order n.

Definition: Given n first order ρ-split functions $s_1^{1,\rho}, \ldots, s_n^{1,\rho}$ in the metric space $(\mathcal{D}, d)$, a ρ-split function of order n $s^{n,\rho} = (s_1^{1,\rho}, s_2^{1,\rho}, \ldots, s_n^{1,\rho}) : \mathcal{D} \to \{0,1,-\}^n$ is the mapping, such that for arbitrary different objects $x, y \in \mathcal{D}$, $\forall i\ s_i^{1,\rho}(x) \neq - \wedge \forall i\ s_i^{1,\rho}(y) \neq - \wedge s^{n,\rho}(x) \neq s^{n,\rho}(y) \Rightarrow d(x,y) > 2\rho$ (separable property) and $\rho_2 \geq \rho_1 \wedge \forall i\ s_i^{1,\rho_2}(x) \neq - \wedge \exists j\ s_j^{1,\rho_1}(y) = - \Rightarrow d(x,y) > \rho_2 - \rho_1$ (symmetry property). ■

An obvious consequence of the ρ-split function definitions, useful for our purposes, is that by combining n ρ-split functions of the first order $s_1^{1,\rho}, \ldots, s_n^{1,\rho}$, which have the separable and symmetric properties, we obtain a ρ-split function of n^{th} order $s^{n,\rho}$, which also demonstrates the separable and symmetric properties. We often refer to the number of symbols generated by $s^{n,\rho}$, that is the parameter n, as the *order* of the ρ-split function. In order to obtain an addressing scheme, we need another function that transforms the ρ-split strings into integer numbers, which we define as follows.

Definition: Given a string $b = (b_1, \ldots, b_n)$ of n elements 0, 1, or $-$, the function $\langle\cdot\rangle : \{0,1,-\}^n \to [0..2^n]$ is specified as:

$$\langle b \rangle = \begin{cases} [b_1, b_2, \ldots, b_n]_2 = \sum_{j=1}^{n} 2^{n-j} b_j, \text{ if } \forall j\ b_j \neq - \\ 2^n, \text{ otherwise} \end{cases}$$

■

When all the elements are different from '$-$', the function $\langle b \rangle$ simply translates the string b into an integer by interpreting it as a binary number (which is always less than 2^n), otherwise the function returns 2^n.

By means of the ρ-split function and the $\langle \cdot \rangle$ operator, we can assign an integer number i $(0 \leq i \leq 2^n)$ to each object $x \in \mathcal{D}$, i.e., the function can group objects from $\mathcal{D}$ in $2^n + 1$ disjoint subsets.

Assuming a ρ-split function $s^{n,\rho}$, we use the capital letter $S^{n,\rho}$ to designate partitions (subsets) of objects from $X \subseteq \mathcal{D}$, which the function can produce. More precisely, given a ρ-split function $s^{n,\rho}$ and a set of objects X, we define $S^{n,\rho}_{[i]}(X) = \{x \in X \mid \langle s^{n,\rho}(x) \rangle = i\}$. We call the first 2^n sets the *separable sets*. The last set that is the set of objects for which the function $\langle s^{n,\rho}(x) \rangle$ evaluates to 2^n is called the *exclusion set*.

For illustration, given two split functions of order 1, $s^{1,\rho}_1$ and $s^{1,\rho}_2$, a ρ-split function of order 2 produces strings as a concatenation of strings returned by functions $s^{1,\rho}_1$ and $s^{1,\rho}_2$.

The combination of the ρ-split functions can also be seen from the perspective of the object data set. Considering again the illustrative example above, the resulting function of order 2 generates an exclusion set as the union of the exclusion sets of the two first order split functions, i.e. $(S^{1,\rho}_{1[2]}(\mathcal{D}) \cup S^{1,\rho}_{2[2]}(\mathcal{D}))$. Moreover, the four separable sets, which are given by the combination of the separable sets of the original split functions, are determined as follows: $\{S^{1,\rho}_{1[0]}(\mathcal{D}) \cap S^{1,\rho}_{2[0]}(\mathcal{D}),\ S^{1,\rho}_{1[0]}(\mathcal{D}) \cap S^{1,\rho}_{2[1]}(\mathcal{D}),\ S^{1,\rho}_{1[1]}(\mathcal{D}) \cap S^{1,\rho}_{2[0]}(\mathcal{D}),\ S^{1,\rho}_{1[1]}(\mathcal{D}) \cap S^{1,\rho}_{2[1]}(\mathcal{D})\}$.

The separable property of the ρ-split function allows to partition a set of objects X in subsets $S^{n,\rho}_{[i]}(X)$, so that the distance from an object in one subset to another object in a different subset is more than 2ρ, which is true for all $i < 2^n$. We say that such disjoint separation of subsets, or partitioning, is *separable up to* 2ρ. This property will be used during the retrieval, since a range query with radius $r \leq \rho$ requires to access only one of the separable sets and, possibly the exclusion set. For convenience, we denote a set of separable partitions $\{S^{n,\rho}_{[0]}(\mathcal{D}), \ldots, S^{n,\rho}_{[2^n-1]}(\mathcal{D})\}$ as $\{S^{n,\rho}_{[\cdot]}(\mathcal{D})\}$. The last partition $S^{n,\rho}_{[2^n]}(\mathcal{D})$ is the exclusion set.

When the ρ parameter changes, the separable partitions shrink or widen correspondingly. The symmetric property guarantees a uniform reaction in all the partitions. This property is essential in the similarity search phase as detailed in Sections 4.4.2 and 4.4.3.

4.2 Pivot-Based Filtering

In general, the pivot-based algorithms can be viewed as a mapping Ψ from the original metric space $\mathcal{M} = (\mathcal{D}, d)$ to a n-dimensional vector space with the L_∞ distance. The mapping assumes a set $P = \{p_1, p_2, \ldots p_n\}$ of objects from $\mathcal{D}$, called pivots, and for each database object o, the mapping determines its characteristic (feature) vector as $\Psi(o) = (d(o,p_1), d(o,p_2), \ldots d(o,p_n))$. We designate the new metric space as $\mathcal{M}_P = (\mathbb{R}^n, L_\infty)$. At search time, we compute for a query object q the query feature vector $\Psi(q) = (d(q,p_1), d(q,p_2), \ldots d(q,p_t))$ and discard for the search radius r an object o if

$$L_\infty(\Psi(o), \Psi(q)) > r \tag{4.1}$$

In other words, the object o can be discarded if for some pivot p_i,

$$\mid d(q,p_i) - d(o,p_i) \mid > r \tag{4.2}$$

Due to the triangle inequality, the mapping Ψ is contractive, that is all discarded objects do not belong to the result set. However, some not-discarded objects may not be relevant and must be verified through the original distance function $d(\cdot)$. For more details, see Section 2.6 or [21].

4.3 Illustration of the idea

Before we specify the structure and the insertion/search algorithms of D-Index, we illustrate the ρ-split clustering and pivoting techniques by simple examples.

It is intuitively clear that many ρ-split functions can be defined, and that some of functions can be more convenient than the others, which might also depend on the data. On the extreme basis, a ρ-split function can allocate all objects in one of the subsets (partitions) either in a separable set or in the exclusion set, keeping the others empty. Of course, there is no meaning in such a split whatever subset would contain the result. Considering the fact that we can combine several first-order ρ split functions in order to obtain smaller separable partitions, we can say that a good ρ-split function should aim at maximizing the following two objective measures:

balancing – Separable sets should be of (approximately) equal size to avoid a skew in object allocation for separable sets. In principle, there are two reasons to achieve such the objective. First, since one of the separable sets can always be neglected for any kind of query, balanced partitions should guarantee good performance on the average – large sets have high probability of access, provided the distribution of query objects follows the distribution of searched data, which is usually true. Second, recursive partitioning of unbalanced sets can only increase the data allocation skew.

minimization – The exclusion set should be small, because it is the set that cannot typically be eliminated from the search point of view.

Several different types of first order ρ-split functions are proposed, analyzed, and evaluated in [32]. The ball partitioning split (bps) originally proposed in [88] under the name excluded middle partitioning, provided the smallest exclusion set. For this reason, we also apply this approach that can be characterized as follows.

The ball partitioning ρ-split function $bps^{\rho}(x, x_v)$ uses one object $x_v \in \mathcal{D}$ and the medium distance d_m to partition the data file into three subsets $BPS^{1,\rho}_{[0]}$, $BPS^{1,\rho}_{[1]}$ and $BPS^{1,\rho}_{[2]}$ – the result of the function $\langle bps^{1,\rho}(x)\rangle$ gives the index of the set to which the object x belongs, for convenience see Figure 4.1.

$$bps^{1,\rho}(x) = \begin{cases} 0 & \text{if } d(x, x_v) \leq d_m - \rho \\ 1 & \text{if } d(x, x_v) > d_m + \rho \\ - & \text{otherwise} \end{cases} \tag{4.3}$$

Note that the medium distance d_m is relative to x_v and it is defined to satisfy the balancing requirement, the cardinality of the set $BPS^{1,\rho}_{[0]}$ and the cardinality of $BPS^{1,\rho}_{[1]}$ are nearly the same. In other words, the number of objects with distances smaller than or equal to d_m is practically the same as the number of objects with distances larger than d_m. For more details, see [87].

In order to see that the bps^{ρ}-split function behaves in the desired way, it is sufficient to prove that the separable and symmetric properties hold.

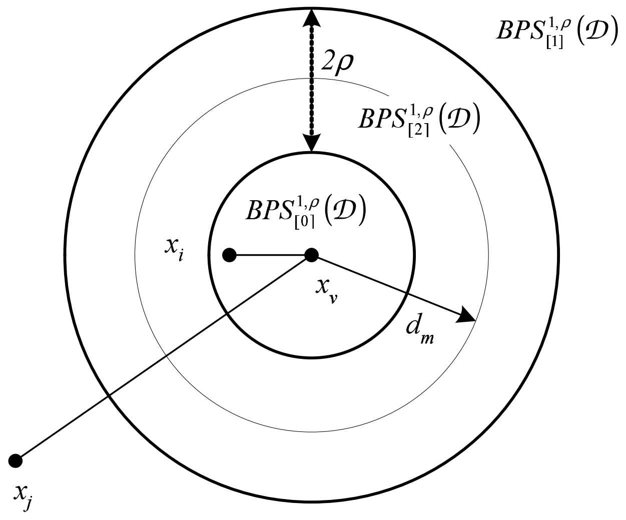

$BPS^{1,\rho}_{[1]}(\mathcal{D})$
2ρ
$BPS^{1,\rho}_{[2]}(\mathcal{D})$
$BPS^{1,\rho}_{[0]}(\mathcal{D})$
x_i
x_v
d_m
x_j

Figure 4.1: The ball partitioning based on the excluded middle partitioning

Separable property We have to prove that $\forall x_0 \in BPS^{1,\rho}_{[0]}, x_1 \in BPS^{1,\rho}_{[1]} : d(x_0, x_1) > 2\rho$.

Proof: If we consider the definition of the $bps^{1,\rho}$ split function in Equation 4.3, we can write $d(x_0, x_v) \leq d_m - \rho$ and $d(x_1, x_v) > d_m + \rho$. Since the triangle inequality among x_0, x_1, x_v holds, we get $d(x_0, x_1) + d(x_0, x_v) \geq d(x_1, x_v)$. If we combine these inequalities and simplify the expression, we get $d(x_0, x_1) > 2\rho$. ■

Symmetric property We have to prove that $\forall x_0 \in BPS^{1,\rho_2}_{[0]}, x_1 \in BPS^{1,\rho_2}_{[1]}, y \in BPS^{1,\rho_1}_{[2]} : d(x_0, y) > \rho_2 - \rho_1$ and $d(x_1, y) > \rho_2 - \rho_1$ with $\rho_2 \geq \rho_1$.

Proof: We prove the assertion only for the object x_0 because the proof for x_1 is analogous. Since $y \in BPS^{1,\rho_1}_{[2]}$ then $d(y, x_v) > d_m - \rho_1$. Moreover, since $x_0 \in BPS^{1,\rho_2}_{[0]}$ then $d_m - \rho_2 \geq d(x_0, x_v)$. By summing both the sides of the above inequalities we obtain $d(y, x_v) - d(x_0, x_v) > \rho_2 - \rho_1$. Finally, from the triangle inequality we have that

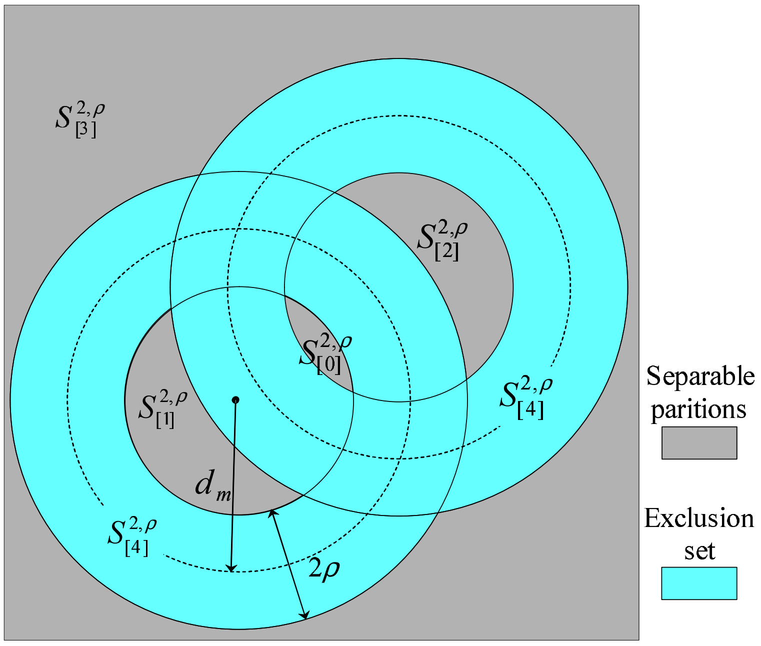


Figure 4.2: Combination of two first-order ρ split functions applied on the two-dimensional space.

$d(x_0, y) \geq d(y, x_v) - d(x_0, x_v)$ and then $d(x_0, y) > \rho_2 - \rho_1$. ■

As explained in Section 4.1, once we have defined a set of first order ρ-split functions, it is possible to combine them in order to obtain a function that generates more partitions. This idea is depicted in Figure 4.2 where two bps-split functions, on the two-dimensional space, are used. The domain $\mathcal{D}$, represented by the gray square, is divided into four regions $\{S^{2,\rho}_{[\cdot]}(\mathcal{D})\}$=$\{S^{2,\rho}_{[0]}(\mathcal{D}), S^{2,\rho}_{[1]}(\mathcal{D}), S^{2,\rho}_{[2]}(\mathcal{D}), S^{2,\rho}_{[3]}(\mathcal{D})\}$, corresponding to the separable partitions. The partition $S^{2,\rho}_{[4]}(\mathcal{D})$, the exclusion set, is represented by the brighter region and it is formed by the union of the exclusion sets resulting from the two splits.

In Figure 4.3, the basic principle of the pivoting technique is illustrated, where q is the query object, r the query radius, and p_i a pivot. Provided the distance between any object and p_i is known, the

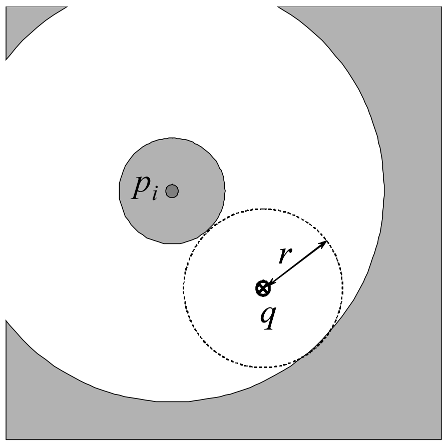


Figure 4.3: Example of pivots behavior.

gray area represents the region of objects x, that do not belong to the query result. This can easily be decided without actually computing the distance between q and x, by using the triangle inequalities $d(x, p_i) + d(x, q) \geq d(p_i, q)$ and $d(p_i, q) + d(x, q) \geq d(x, p_i)$, and respecting the query radius r. Naturally, by using more pivots, we can improve the probability of excluding an object without actually computing its distance with respect to q. More details are provided in Section 2.6.

4.4 System Structure and Algorithms

In this section we specify the structure of the Distance Searching Index, D-Index, and discuss the problems related to the definition of ρ-split functions and the choice of reference objects. Then we specify the insertion algorithm and outline the principle of searching. Finally, we define the generic similarity range and nearest neighbor algorithms.

4.4.1 Storage Architecture

The basic idea of the D-Index is to create a multilevel storage and retrieval structure that uses several ρ-split functions, one per each level, to create an array of *buckets* for storing objects. On the first level, we use a ρ-split function for separating objects of the whole data set. For any other level, objects mapped to the exclusion bucket of the previous level are the candidates for storage in separable buckets of this level. Finally, the exclusion bucket of the last level forms the exclusion bucket of the whole D-Index structure. It is worth noting that the ρ-split functions of individual levels use the same value of the parameter ρ. Moreover, split functions can have different order, typically decreasing with the level, allowing the D-Index structure to have levels with the different number of buckets. More precisely, the D-Index structure can be defined as follows.

Definition: Given a set of objects $X \subset \mathcal{D}$, the h-level distance searching index $DI^{\rho}(X, m_1, m_2, \ldots, m_h)$ with the buckets at each level separable up to 2ρ, is determined by h independent ρ-split functions $s_i^{m_i,\rho}$, where $i = 1, 2, \ldots, h$, which generate:

$$\text{Exclusion bucket } E_i = \begin{cases} S^{m_1,\rho}_{1[2^{m_1}]}(X) & \text{if } i = 1 \\ S^{m_i,\rho}_{i[2^{m_i}]}(E_{i-1}) & \text{if } i > 1 \end{cases}$$

$$\text{Separable buckets } \{B_{i,0}, \ldots, B_{i,2^{m_i}-1}\} = \begin{cases} \{S^{m_1,\rho}_{1[\cdot]}(X)\} & \text{if } i = 1 \\ \{S^{m_i,\rho}_{i[\cdot]}(E_{i-1})\} & \text{if } i > 1 \end{cases}$$

■

From the structure point of view, you can see the buckets organized as the following two-dimensional array consisting of $1 + \sum_{i=1}^{h} 2^{m_i}$ elements.

$$\begin{array}{l} B_{1,0}, B_{1,1}, \ldots, B_{1,2^{m_1}-1} \\ B_{2,0}, B_{2,1}, \ldots, B_{2,2^{m_2}-1} \\ \vdots \\ B_{h,0}, B_{h,1}, \ldots, B_{h,2^{m_h}-1}, E_h \end{array}$$

All separable buckets are included, but only the E_h exclusion bucket is present because exclusion buckets $E_{i<h}$ are recursively repartitioned on the level $i + 1$. Then, for each row (i.e. the D-Index level)

i, 2^{m_i} buckets are separable up to 2ρ thus we are sure that do not exist two buckets at the same level i both containing relevant objects for any similarity range query with radius $r \leq \rho$. Naturally, there is a tradeoff between the number of separable buckets and the amount of objects in the exclusion bucket – the more separable buckets are, the greater the number of objects in the exclusion bucket is. However, the set of objects in the exclusion bucket can recursively be repartitioned, with possibly different number of bits (m_i) and certainly different ρ-split functions. In Figure 4.4, we present an example of such a D-Index structure with varying number of separable buckets per level. The structure consists of six levels. The exclusion buckets, which are recursively repartitioned are shown as the dashed squares. Obviously, the exclusion bucket of the sixth level forms the exclusion bucket of the whole structure. Some illustrative objects are inserted into the structure to show the behavior of ρ-split functions. Notice that an object (e.g. the rectangle) can fall into the exclusion set several times until it is accommodated in a separable bucket or in the global exclusion bucket.

4.4.2 Insertion and Search Strategies

In order to complete our description of the D-Index, we present an insertion algorithm and a sketch of a simplified search algorithm. Advanced search techniques are described in the next subsection.

Insertion of object $x \in X$ in $DI^{\rho}(X, m_1, m_2, \ldots, m_h)$ proceeds according to the following algorithm.

Algorithm 4.4.1 *Insertion*

```
for $i = 1$ to $h$
    if $\langle s_i^{m_i,\rho}(x) \rangle < 2^{m_i}$
        $x \mapsto B_{i,\langle s_i^{m_i,\rho}(x) \rangle}$;
        exit;
    end if
end for
$x \mapsto E_h$;                                        ■
```

Starting from the first level, Algorithm 4.4.1 tries to accommodate x into a separable bucket. If a suitable bucket exists, the object is stored

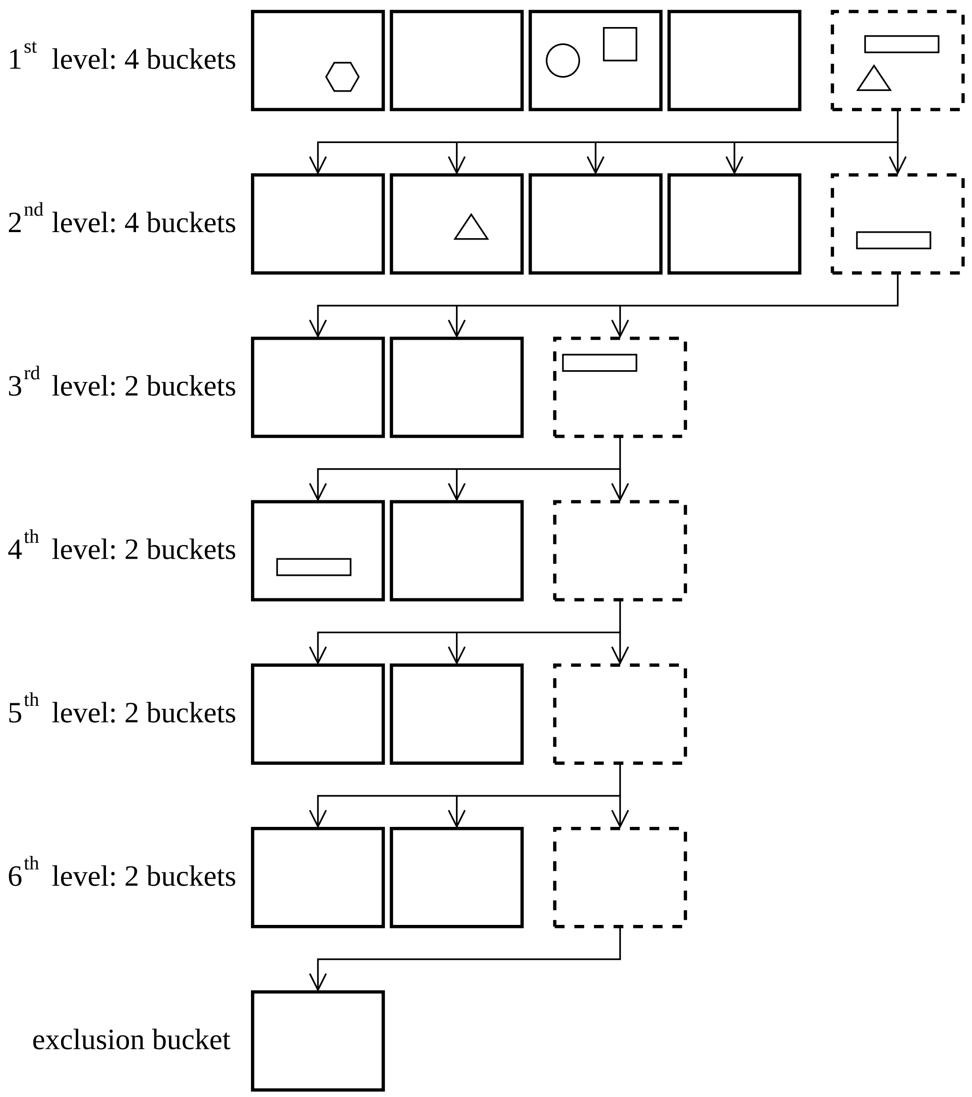


Figure 4.4: Example of D-Index structure.

in the bucket. If it fails for all levels, the object x is placed in the exclusion bucket E_h. In any case, the insertion algorithm determines exactly one bucket to store the object. As for the number of distance computations, the D-Index needs $\sum_{i=1}^{j} m_i$ distance computations to store a new object, assuming that the new object is inserted into a separable bucket of j^{th} level.

Given a query region $\mathcal{Q} = R(q, r)$ with $q \in \mathcal{D}$ and $r \leq \rho$, a simple algorithm can execute the query as follows.

Algorithm 4.4.2 *Search*

```
for $i = 1$ to $h$
    return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,0}(q)\rangle}$;
end for
return all objects $x$ such that $x \in \mathcal{Q} \cap E_h$;        ■
```

The function $\langle s_i^{m_i,0}(q)\rangle$ always gives a value smaller than 2^{m_i}, because $\rho = 0$. Consequently, one separable bucket on each level i is determined. Objects of the query response set cannot be in any other separable bucket on the level i, because r is not greater than ρ ($r \leq \rho$) and the buckets are separable up to 2ρ. However, some of them can be in the exclusion zone, Algorithm 4.4.2 assumes that exclusion buckets are always accessed. For that reason, all levels are considered, and also the exclusion bucket E_h is accessed. These straightforward search and insertion algorithms are also presented in [43], as well as, a short preliminary version of the D-Index. The execution of Algorithm 4.4.2 requires $h + 1$ bucket accesses, which forms the upper bound of a more sophisticated algorithm described in the next section.

4.4.3 Generic Search Algorithms

The search Algorithm 4.4.2 requires to access one bucket at each level of the D-Index, plus the exclusion bucket. In the following situations, however, the number of accesses can even be reduced:

- if the whole query region is contained in the exclusion partition of the level i, the query cannot have objects in the separable buckets of this level. Thus, only the next level, if it exists, must be considered;

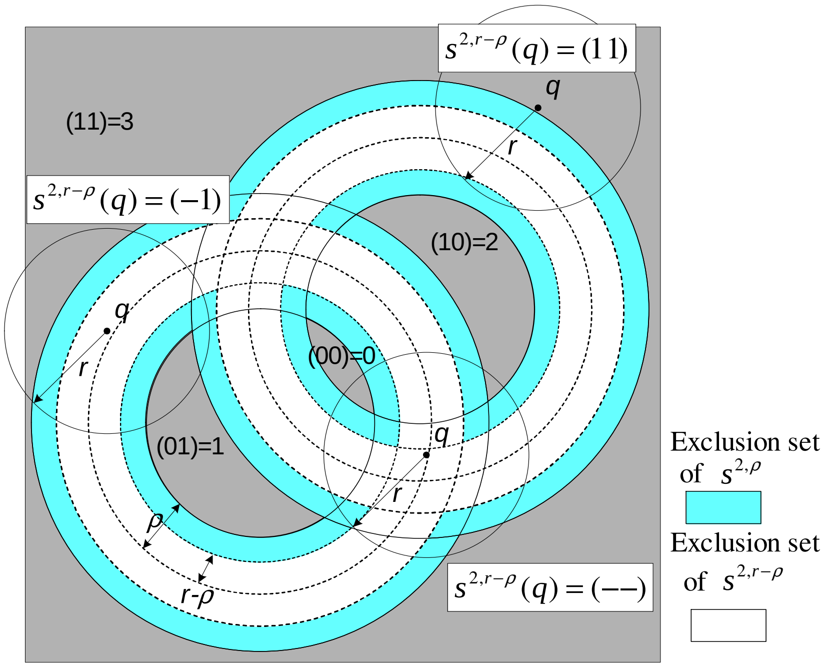


Figure 4.5: Example of use of the function G.

- if the query region is exclusively contained in a separable partition of the level i, the following levels, as well as the exclusion bucket E_h of the whole structure, need not be accessed, thus, the search terminates on the level i.

Another drawback of the simple algorithm is that it works only for search radii up to ρ. However, with additional computational effort, range queries with $r > \rho$ can also be executed. Indeed, queries with $r > \rho$ can be executed by evaluating the split function $s^{n,r-\rho}$ with the second parameter adjusted. In case $s^{n,r-\rho}$ returns a string without any '$-$', the result is contained in a single bucket (namely $B_{\langle s^{n,r-\rho}\rangle}$) plus, possibly, the exclusion bucket. Figure 4.5 provides the example of this situation where the function $s^{2,r-\rho}$ returns the string '11' for the range query $R(q,r)$.

Let us now consider that the returned string contains at least one '$-$'. We indicate this string as $(b_1,\ldots,b_n)$ with $b_i = \{0,1,-\}$. In case there is only one $b_i =$ '$-$', we must access all buckets B which indexes are obtained by replacing the '$-$' by 0 and 1 in the result of $s^{n,r-\rho}$. In the most general case we must substitute in $(b_1, \ldots, b_n)$ all symbols '$-$' with zeros and ones to generate all possible combinations. A simple example of this concept is illustrated in Figure 4.5 where the query $R(q', r)$ enters the separable buckets denoted by '01' and '11' because $s^{2,r-\rho}(q')$ returns '-1'. Similarly, the range query with the query object q'' visits all separable buckets because the function s evaluates to '$--$'.

In order to define an algorithm for this process, we need some additional terms and notations.

Definition: We define an extended *exclusive OR* bit operator, $\otimes$, which is based on the following truth-value table:

$bit1$	0	0	0	1	1	1	$-$	$-$	$-$
$bit2$	0	1	$-$	0	1	$-$	0	1	$-$
$\otimes$	0	1	0	1	0	0	0	0	0

■

Notice that the operator $\otimes$ can be used bit-wise on two strings of the same length and that it always returns a standard binary number (i.e., it does not contain any '$-$'). Consequently, $\langle s_1 \otimes s_2 \rangle < 2^n$ is always true for strings s_1 and s_2 of the length n (refer to Definition 4.1).

Definition: Given a string s of the length n, $G(s)$ denotes a subset of $\Omega_n = \{0, 1, \ldots, 2^n - 1\}$, such that all elements $e_i \in \Omega_n$ for which $\langle s \otimes e_i \rangle \neq 0$ are eliminated (interpreting e_i as a binary string of length n). ■

Observe that, $G(- - \ldots -) = \Omega_n$, and that the cardinality of the set is the second power of the number of '$-$' elements (2^n), that is generating all partition identifications in which the symbol '$-$' is alternatively substituted by zeros and ones. In fact, we can use $G(\cdot)$ to generate, from a string returned by a split function, the set of partitions that need be accessed to execute the query. As an example let us consider $n = 2$ as in Figure 4.5. If $s^{2,r-\rho} = (-1)$ then we must access buckets B_1 and B_3. In such the case $G(-1) = \{1, 3\}$. If $s^{2,r-\rho} = (--)$ then we must access all buckets, as the function $G(--) = \{0, 1, 2, 3\}$ indicates.

We only apply the function G when the search radius r is greater than the parameter ρ. We first evaluate the split function using $r - \rho$, which is, of course, greater than 0. Next, the function G is applied on the resulting string. In this way, G generates the partitions that intersect the query region.

In the following, we specify how such ideas are applied in the D-Index to implement generic range and nearest neighbor algorithms.

4.4.4 Range Query Algorithm

Given a query region $\mathcal{Q} = R(q, r)$ with $q \in \mathcal{D}$ and $r \leq d^+$. An advanced algorithm can execute the similarity range query as follows.

Algorithm 4.4.3 *Range Search*

```
01. for i=1 to h
02.    if $\langle s_i^{m_i,\rho+r}(q) \rangle < 2^{m_i}$ then
          # the query is exclusively contained in a separable bucket
03.          return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho+r}(q)\rangle}$;
             exit;
04.    end if
05.    if $r \leq \rho$ then
          # search radius up to $\rho$
06.       if $\langle s_i^{m_i,\rho-r}(q) \rangle < 2^{m_i}$ then
             # the query is not exclusively contained in the exclusion bucket of the level $i$
07.             return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho-r}(q)\rangle}$;
08.       end if
09.    else
          # search radius greater than $\rho$
10.       let $\{l_1, l_2, \ldots, l_k\} = G(s_i^{m_i,r-\rho}(q))$
11.       return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,l_1}$ or $x \in \mathcal{Q} \cap B_{i,l_2}$ or ... or $x \in \mathcal{Q} \cap B_{i,l_k}$;
12.    end if
13. end for
14. return all objects $x$ such that $x \in \mathcal{Q} \cap E_h$;     ■
```

In general, Algorithm 4.4.3 considers all D-Index levels and eventually also accesses the global exclusion bucket. However, due to the symmetry property, the test on the line 02 can discover the exclusive containment of the query region in a separable bucket and terminate the search earlier. Otherwise, the algorithm proceeds according to the size of the query radius. If it is not greater than ρ, line 05, there are two possibilities. If the test on line 06 is satisfied, one separable bucket is accessed. Otherwise no separable bucket is accessed on this level, because the query region is, from this level point of view, exclusively in the exclusion zone. Provided the search radius is greater than ρ, more separable buckets are accessed on a specific level as defined by lines 10 and 11. Unless terminated earlier, the algorithm accesses the exclusion bucket at line 14.

4.4.5 k-Nearest Neighbors Search

The task of the nearest neighbors search is to retrieve k closest elements from X to $q \in \mathcal{D}$, respecting the metric distance measure of the metric space $\mathcal{M}$. Specifically, it retrieves a set $A \subseteq X$ such that $\mid A \mid = k$ and $\forall x \in A, y \in X - A, d(q,x) \leq d(q,y)$. In case of ties, we are satisfied with any set of k elements complying with the condition. For further details, see Section 2.3. For our purposes, we designate the distance to the k^{th} nearest neighbor as d_k ($d_k \leq d^+$).

The general strategy of Algorithm 4.4.4 works as follows. At the beginning (line 01), we assume that the response set A is empty and $d_k = d^+$. The algorithm then proceeds with an optimistic strategy (lines 02 through 13) assuming that all k nearest neighbors are within the distance of ρ. If it fails, see the test on line 14, additional steps of the search are performed to find the correct result. Specifically, the first phase of the algorithm determines the buckets (maximally one separable at each level) by using the range query strategy with radius $r = min\{d_k, \rho\}$. In this way, the most promising buckets are accessed and their identifications are remembered to avoid multiple bucket accesses. If $d_k \leq \rho$ the search terminates successfully and A contains the result set. Otherwise, additional bucket accesses are performed (lines 15 through 22) using the strategy for radii greater than ρ of Algorithm 4.4.3, ignoring the buckets already accessed in the first (optimistic) phase.

Algorithm 4.4.4 *Nearest Neighbor Search*

```
01. $A = \emptyset, d_k = d^+$;
    # initialization
02. for i=1 to $h$
    # first, try the optimistic approach
03.     $r = min\{d_k, \rho\}$;
04.     if $\langle s_i^{m_i,\rho+r}(q)\rangle < 2^{m_i}$ then
        # the query is exclusively contained in a separable bucket
05.         access bucket $B_{i,\langle s_i^{m_i,\rho+r}(q)\rangle}$;
            update $A$ and $d_k$;
06.         if $d_k \leq \rho$ then
                exit;
                # the response set is determined
            end if
07.     else
08.         if $\langle s_i^{m_i,\rho-r}(q)\rangle < 2^{m_i}$ then
            # the query is not exclusively contained in the exclusion bucket of the level $i$
09.             access bucket $B_{i,\langle s_i^{m_i,0}(q)\rangle}$;
                update $A$ and $d_k$;
10.         end if
11.     end if
12. end for
13. access bucket $E_h$;
    update $A$ and $d_k$;
14. if $d_k > \rho$ then
    # the second phase if needed
15.     for i=1 to $h$
16.         if $\langle s_i^{m_i,\rho+d_k}(q)\rangle < 2^{m_i}$ then
            # the query is exclusively contained in a separable bucket
17.             access bucket $B_{i,\langle s_i^{m_i,\rho+d_k}(q)\rangle}$ if not already accessed;
                update $A$ and $d_k$;
                exit;
18.         else
```

```
19.             let$\{b_1, b_2, \ldots, b_k\} = G(s_i^{m_i, d_k - \rho}(q))$
20.             access buckets $B_{i,b_1}, B_{i,b_2}, \ldots, B_{i,b_k}$ if not accessed;
                update $A$ and $d_k$;
21.         end if
22.     end for
23. end if                                                          ■
```

The output of the algorithm is the result set A and the distance to the last (k^{th}) nearest neighbor, d_k. Whenever a bucket is accessed, the response set A and the distance to the k^{th} nearest neighbor must be updated. In particular, the new response set is determined as the result of the nearest neighbor query over the union of objects from the accessed bucket and the previous response set A. The value of d_k, which is the current search radius in the given bucket, is adjusted according to the new response set. When the cardinality of the set A is less than k the distance to the k^{th} nearest neighbor d_k is equal to d^+ so that no qualifying data object is lost during the query evaluation. Notice that, during the bucket accesses, the pivots used in ρ-split functions are employed in filtering algorithms to further reduce the number of potentially qualifying objects.

4.4.6 Choosing references

In order to apply D-Index, the ρ-split functions must be designed, which obviously requires specification of reference objects (pivots). An important feature of D-Index is that the same reference objects used in the ρ-split functions for the partitioning into buckets are also applied as pivots for the internal management of buckets.

We have already slightly discussed the problem of choosing reference objects in Section 2.7. We concluded that every search technique in metric spaces needs some pivots for search pruning and the way of selecting pivots effects the performance of search algorithms. However, some researches are not involved in this problem.

Many techniques such as Burkhard-Keller Tree, Fixed Queries Tree and its variants Fixed-Height Fixed Queries Tree and Fixed Queries Array selects pivots at random. Obviously, the random choice is the most trivial technique, nonetheless, the fastest one. Yianillos in [87] recognized and analyzed this issue. He suggests that

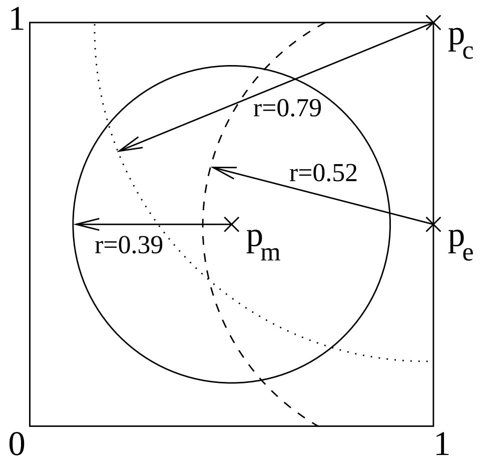


Figure 4.6: Different choices for pivot p to divide the unit square.

some elements of the space may be better pivots than the others. As an example consider Figure 4.6 which presents the unit square with uniform distribution. To partition the space using the ball partitioning we have to set the radius r. There are three natural choices for pivots: the middle point p_m, the middle point of an edge p_e, and a corner point p_c. To choose among the possibilities, notice that the probability of entering both the partitions is proportional to the length of the boundary. Thus, we aim at the minimization of the boundary length. In this respect, the most promising selection is the corner object p_c and the edge object p_e is still better than the middle point p_m. It is interesting that from the clustering point of view p_m is the center of a cluster, however, we have shown that this is the worst possible choice. This technique is applied in Vantage Point Trees.

The heuristic, which selects pivots from corners of the space, can be used in applications where we have an idea about the geometry of the space, which is not often true in metric spaces. The only information we have in metric spaces is the pairwise distances between objects. Bozkaya and Özsoyoglu in [13] give a different reason why the pivots from corners are better than the others, i.e. why they provide better partitioning. Basically, the distance distribution for a corner point is more flat than the distribution for a center point. Figure 4.7 illustrates it for the uniformly distributed points in 20-dimensional Euclidean data space. As we can easily see, the distribution for the

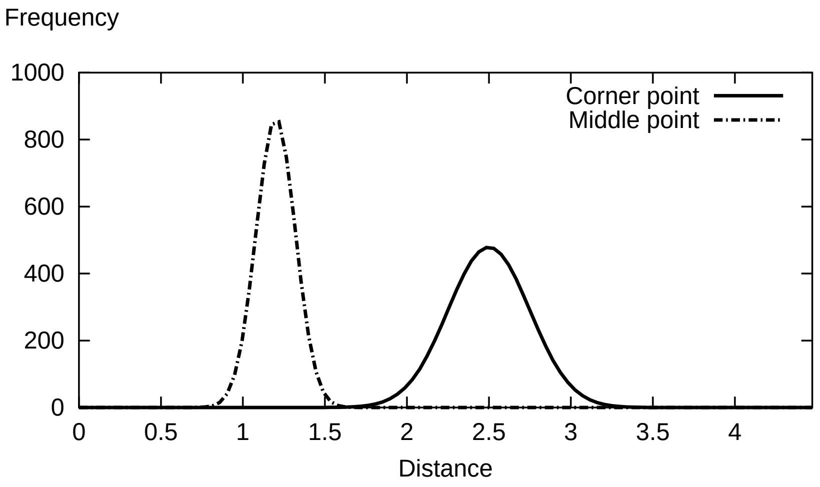


Figure 4.7: Distance distribution for two pivots, one is the center while the other is a corner object, in the unit cube of the 20-dimensional Euclidean space.

central object is sharper and thinner. Setting the radius of ball partitioning to the peak value leads to the higher concentration of objects near the boundary, as a consequence, it leads to the higher likelihood of visiting both the partitions. On contrary, it does not apply for the corner point because the distance distribution is much flatter, thus, the search would be more trimming. A simple heuristic tries to meet these requirements:

1. choose a random object
2. compute all distance from this object to the others
3. choose the farthest object as the pivot

This heuristic is used in Multi Vantage Point Trees.

Zezula et al. [25] provide several alternatives to select pivots in M-Trees. The most convenient option is the heuristic **mM_RAD**. Whenever a node splits this technique selects a pair of pivots that has the smallest maximum of both covering radii among all possible pairs of objects in the split node. Nonetheless, such the heuristic is suitable for structures that cluster data not for ones that partition data.

LAESA access structure also selects pivots far away from each other. The procedure picks an arbitrary object and computes the distances to all remaining objects. The pivot is the most distant object. Next pivot is selected as the furthest object from all previously chosen pivots. This procedure is also used in [79].

Recently, the problem was systematically studied in [17], and several strategies for selecting pivots have been proposed and tested. The authors propose the *efficiency criterion* that compares two sets of pivots and states which of them is better. It uses the mean of distances between every pair of objects in $\mathcal{D}$, denoted by $\mu_{\mathcal{D}}$. Given two sets of pivots $T_1 = \{p_1, p_2, \ldots p_t\}$ and $T_2 = \{p'_1, p'_2, \ldots p'_t\}$ we can say that T_1 is better than T_2 if

$$\mu_{\mathcal{D}_{T_1}} > \mu_{\mathcal{D}_{T_2}}. \tag{4.4}$$

However, the problem is how to find the mean for given pivot set T. An estimate of such quantity is computed as:

- at random, choose l pairs of elements $\{(o_1, o'_1), (o_2, o'_2), \ldots (o_l, o'_l)\}$ from $X \subset \mathcal{D}$;
- for all pairs of elements, compute their distances in the feature space determined by the pivot set T, that is $L_\infty(\Psi(o_1), \Psi(o'_1)), \ldots, L_\infty(\Psi(o_l), \Psi(o'_l))$, where the function Ψ transforms the metric space into the feature space associated by the set of pivot T;
- compute $\mu_{\mathcal{D}_T}$ as the mean of these distances.

In the D-Index, we apply the *incremental selection* strategy, which was proposed in [17] as the most suitable for real world metric spaces. The strategy works as follows. In the beginning, a set $T_1 = \{p_1\}$ of one element from a sample of m database objects is chosen, such that the pivot p_1 has the maximum $\mu_{\mathcal{D}_{p_1}}$ value. Then, a second pivot p_2 is chosen from another sample of m objects of the database, creating a new set $T_2 = \{p_1, p_2\}$ for fixed p_1, maximizing $\mu_{\mathcal{D}_{T_2}}$. The third pivot p_3 is chosen in the same manner, creating another set $T_3 = \{p_1, p_2, p_3\}$ for fixed p_1, p_2, maximizing $\mu_{\mathcal{D}_{T_3}}$. The process is repeated until the desired number of pivots is determined. If all distances needed to estimate $\mu_{\mathcal{D}_T}$ are kept we must compute only

$2ml$ distances to estimate the new value of $\mu_{\mathcal{D}}$ whenever a new pivot is added. The total cost for selecting k pivots is $2lmk$ distance computations.

In order to verify the capability of the incremental selection strategy (INC), we have also tried to select pivots at random (RAN) and with a modified incremental strategy, called the *middle search* strategy (MID). The MID strategy combines the original INC approach with the following idea. The set of pivots $T_1 = \{p_1, p_2, \dots p_t\}$ is better than the set $T_2 = \{p'_1, p'_2, \dots p'_t\}$ if

$$\mid \mu_{T_1} - d_m \mid < \mid \mu_{T_2} - d_m \mid, \tag{4.5}$$

where d_m represents the global mean distance of the given data set, that is the mean of distances between all pairs of objects in $\mathcal{D}$ not in the feature space. In principle, this strategy supports choices of pivots with high $\mu_{\mathcal{D}_T}$ but also tries to keep distances between pairs of pivots close to d_m. The hypothesis is that too close or too distant pairs of pivots are not suitable for good partitioning of an arbitrary metric space. Consider a metric space with sets as objects and Jacard coefficient (see Section 2.2), the INC heuristic would select a pivot which is far away from other objects. In the worst case, the selected pivot p that is completely different from the others, that is, the distance $d(p, o) = 1$ for all $o \in \mathcal{D}$. Such the anchor is useless from the search point of view because it is not able to filter any object. The MID strategy aims at avoiding such the behavior.

4.4.7 Implementation of D-Index

Since a uniform bucket occupation is difficult to guarantee, each bucket consists of a header plus a dynamic list of fixed size blocks of capacity that is sufficient to accommodate any object from the processed file. Notice that the size of data objects can be different. The number of blocks of a bucket B_{ij} is denoted by b_{ij}. We refer to individual blocks of the bucket B_{ij} as members of an array, that is, $B_{ij}[\cdot]$. In this respect, each bucket consists of the following blocks $B_{ij}[1], B_{ij}[2], \dots, B_{ij}[b_{ij}]$. The header is characterized by a set of pivots $p_1, \dots, p_n$ which was used in split functions to create the bucket. The header also contains distances from these pivots to all objects stored in the bucket. These distances are computed during object insertions and are applied in the pivot-filtering algorithm that is described below. Furthermore, objects are sorted according to their distances to the first pivot p_1 to form a non–decreasing sequence. In order to cope with dynamic data, a block overflow is solved by splits.

Given a query object q with the search radius r a bucket evaluation proceeds in the following steps:

- ignore all blocks of the bucket that contain objects with distances to p_1 outside the interval $[d(q, p_1) - r, d(q, p_1) + r]$;
- for each not ignored block $B_{ij}[k]$, if $\forall o \in B_{ij}[k]$, $L_\infty(\Psi(o), \Psi(q)) > r$ ignore the block;
- access remaining blocks $\{B_{ij}[\cdot]\}$ and $\forall o \in B_{ij}[\cdot]$ compute $d(q, o)$ if $L_\infty(\Psi(o), \Psi(q)) \leq r$. If $d(q, o) \leq r$ the object o qualifies.

In this way, the number of accessed blocks and necessary distance computations are minimized. The mapping function Ψ uses the set of pivots $p_1, \ldots, p_n$ and transform the metric space $\mathcal{M}$ into a vector space, for further detail see Section 2.6.

4.4.8 Properties of D-Index

On a qualitative level, the most important properties of D-Index can be summarized as follows:

- An object is typically inserted at the cost of one block access because objects in buckets are sorted. Since blocks are of fixed length a block can overflow and the split of the overloaded block costs another block access. The number of distance computations between the reference (pivot) and inserted objects varies from m_1 to $\sum_{i=1}^{h} m_i$. For an object inserted on the level j, $\sum_{i=1}^{j} m_i$ distance computations are needed.
- For all response sets such that the distance to the most dissimilar object does not exceed ρ, the number of bucket accesses is maximally $h + 1$, that is, one bucket per level and the global exclusion bucket. The following rule is generally true: the smaller the search radius is the more efficient the search can be.

- For $r = 0$, i.e. the exact match, the successful search is typically solved with one block access, moreover, the unsuccessful search usually does not require any access – very skewed distance distributions may result in slightly more accesses.

- For search radii greater than ρ, the number of bucket accesses increases and is higher than $h + 1$, and furthermore for very large search radii the query evaluation can eventually access all buckets.

- The number of accessed blocks of a bucket depends on the search radius and the distance distribution with respect to the first pivot.

- The number of distance computations between the query object and pivots is determined by the highest level where a bucket was accessed. Provided that the level is j, the number of distance computations is $\sum_{i=1}^{j} m_i$, with the upper bound for any kind of query $\sum_{i=1}^{h} m_i$.

- The number of distance computations between the query object and data objects stored in accessed buckets depends on the search radii and on the number of precomputed distances to pivots and, of course, on the data distribution.

From the quantitative point of view, we investigate the performance of D-Index in the next chapter.

Chapter 5

D-Index Evaluation

This chapter presents the experimental evaluation of the proposed access structure D-Index. We have implemented a fully functional prototype of the D-Index and conducted numerous experiments to verify its properties on three data sets. We have also compared our structure with other access methods for metric spaces, namely M-Tree and VPF forest.

5.1 Data Sets

We have executed all experiments on realistic data sets with significantly different data distributions to demonstrate the wide range of applicability of the D-Index. We used the following three metric data sets:

VEC 45-dimensional vectors of image color features compared by the quadratic distance measure respecting correlations between individual colors.

URL sets of URL addresses attended by users during work sessions with the Masaryk University information system. The distance measure used was based on the similarity of sets, defined as the fraction of cardinalities of intersection and union (Jacard coefficient).

STR sentences of a Czech language corpus compared by the edit distance measure that counts the minimum number of insertions, deletions or substitutions to transform one string into the other.

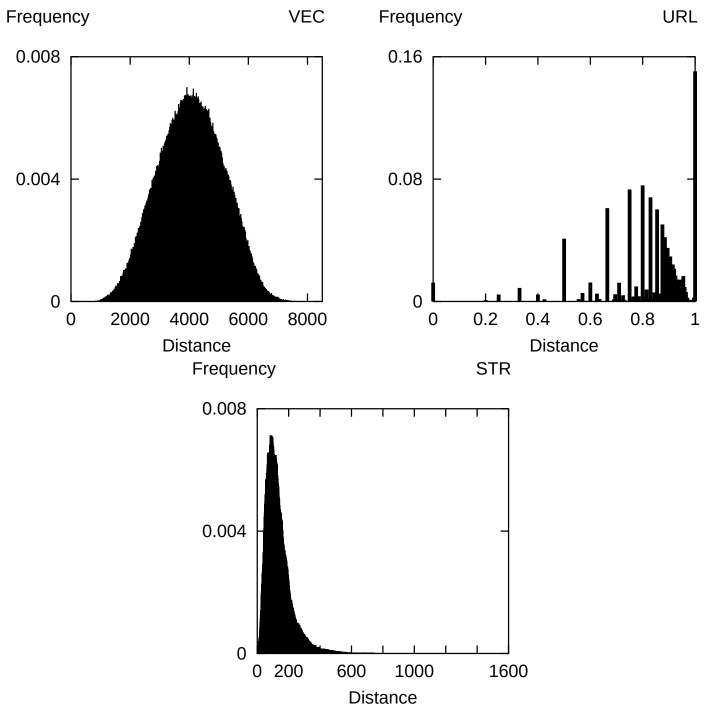


Figure 5.1: Distance densities for VEC, URL, and STR.

For illustration, see Figure 5.1 for distance densities of all our data sets. Notice the practically normal distribution of VEC, very discrete distribution of URL, and the skewed distribution of STR.

In all our experiments, the query objects are not chosen from the indexed data sets, but they follow the same distance distribution. The search costs are measured in terms of distance computations and block reads. We deliberately ignore the L_∞ distance measures used by the D-Index for the pivot-based filtering, because according to our tests, the costs to compute such distances are several orders of magnitude smaller than the costs needed to compute any of the distance functions of our experiments. All presented cost values are mean values obtained by executing queries for 50 different query objects and a constant search selectivity, that is, queries using the same search radius or the same number of the nearest neighbors.

5.2 D-Index Performance Evaluation

In order to test the basic properties of the D-Index, we have considered about 11,000 objects for each of our data sets VEC, URL, and STR. Notice that the number of objects used does not affect the significance of the results, since the cost measures (i.e. the number of block reads and the number of the distance computations) are not influenced by the cache of the system. Moreover, as shown in the following, performance scalability is linear with data set size. First, we have tested the efficiency of the incremental (INC), the random (RAN), and the middle (MID) strategies to select good reference objects, for details see Section 4.4.6. We have built D-Index organizations for each the data set and preselected reference objects. The structures of D-Index, which are specified by the number of levels and specific ρ-split functions were designed manually. In Table 5.2, we summarize characteristics of the resulting D-Index organizations for individual data sets separately.

set	h	# of buckets	# of blocks	block size	ρ
VEC	9	21	2,600	1KB	1,200
URL	9	21	228	13KB	0.225
STR	9	21	289	6KB	14

Figure 5.2: D-Index organization parameters

We measured the search efficiency for range queries by changing the search radius to retrieve maximally 20% of data, that is, about 2,000 objects. The results are presented in Figures 5.3, 5.4, 5.5 for all three data sets and for both distance computations (on the left) and block reads (on the right).

The general conclusion is that the incremental method proved to be the most suitable for choosing reference objects for the D-Index. Except for very few search cases, the method was systematically better and can certainly be recommended – for the sentences, the middle technique was nearly as good as the incremental, and for the vectors, the differences in performance of all three tested methods were the least evident. An interesting observation is that for vectors with distribution close to uniform, even the randomly selected reference

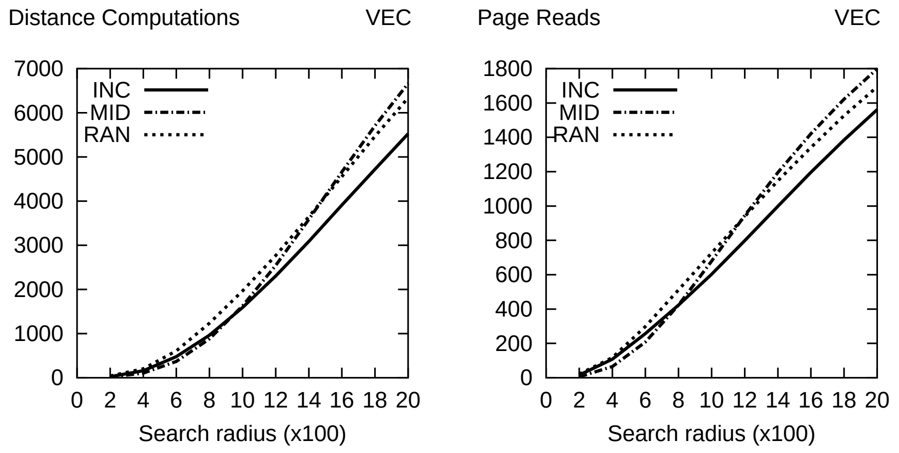


Figure 5.3: Search efficiency for VEC in the number of distance computations and in the number of read blocks.

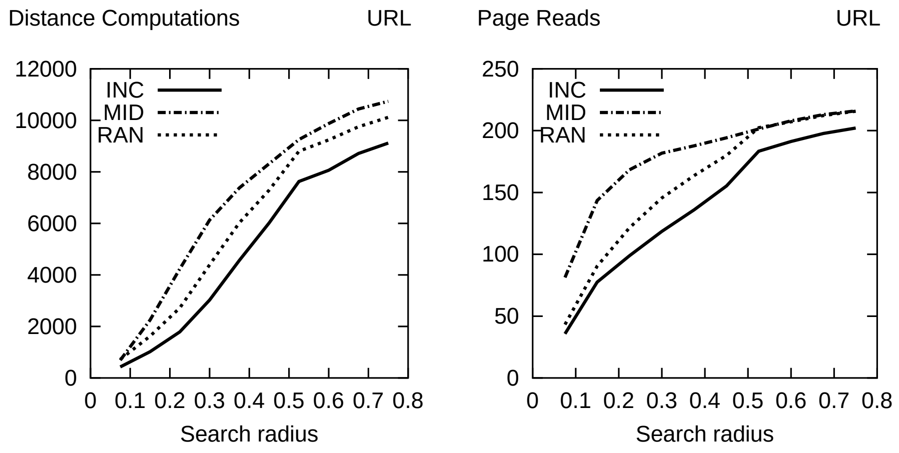


Figure 5.4: Search efficiency for URL in the number of distance computations and in the number of read blocks.

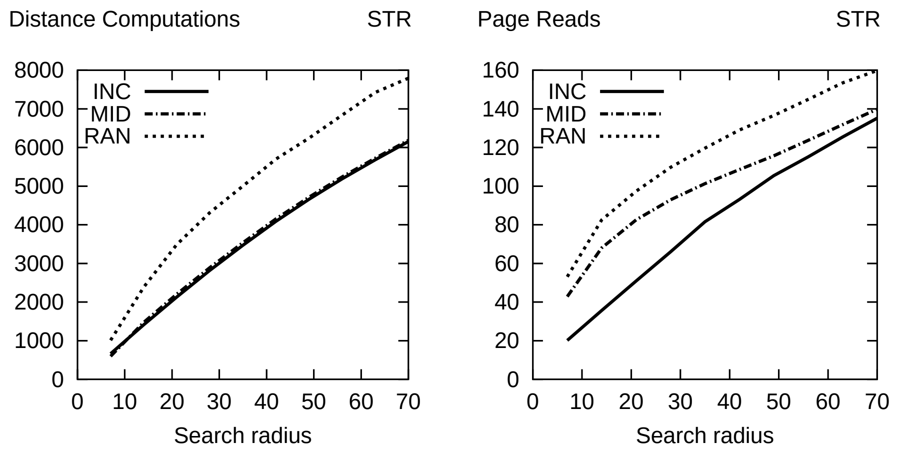


Figure 5.5: Search efficiency for TEXT in the number of distance computations and in the number of read blocks.

points worked quite well. However, when the distance distribution was significantly different from the uniform, the performance with the randomly selected reference points was poor.

Notice also some differences in performance between the execution costs quantified in block reads and distance computations. But the general trends are similar and the incremental method is the best. However, different cost quantities are not equally important for all the data sets. For example, the average time for quadratic form distances for the vectors were computed in much less than 90 μsec, while the average time spent computing a distance between two sentences was about 4.5 msec. This implies that the reduction of block reads for vectors was quite important in the total search time, while for the sentences, it was not important at all.

5.3 D-Index Structure Design

The objective of the second group of tests was to investigate the relationship between the D-Index structure and the search efficiency. We modified the value of the parameter ρ, which influences the structure of D-Index and executed several queries to verify the search performance. We have to remark that the value of the parameter ρ influences the size of the exclusion set of a ρ-split function, the greater ρ is

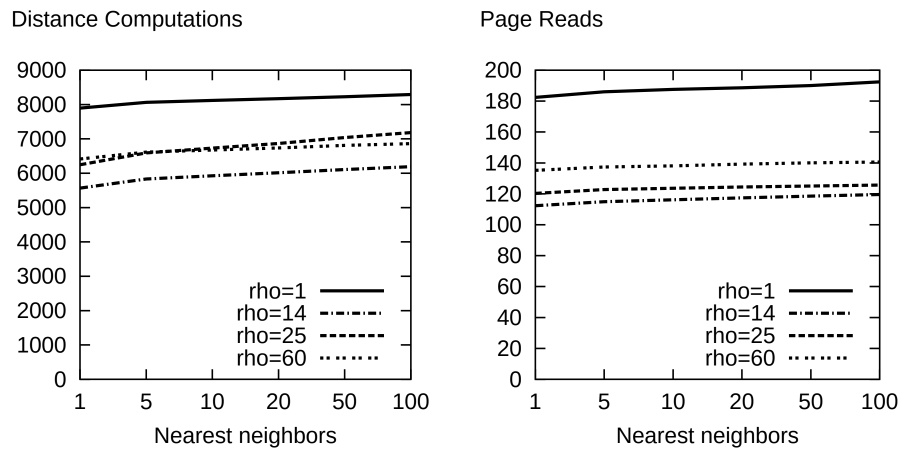


Figure 5.6: Nearest neighbors search on sentences using different D-Index structures.

the larger the exclusion set is and the smaller the separable buckets are. Assuming that we want to have separable buckets with a certain capacity (in number of objects) we must modify the order of split functions if we modify ρ. The general rule is that if we increase the value of ρ we must decrease the amount of separable buckets, that is, the order of split function. Obviously, we can build a structure with few levels h and many buckets b per level, using a relatively small ρ. But when a larger ρ is used, we are able to organize the same data in the D-Index with more levels, but less buckets per level.

We have tested this hypothesis on the data set STR, using $\rho = 1, 14, 25, 60$. The parameters of structures created for each value of ρ are summarized in the following table.

ρ	1	14	25	60
h	3	3	10	9
b	43	31	23	19

The results of experiments for the nearest neighbors search are reported in Figure 5.6 while the performance for range queries is summarized in Figure 5.7. It is quite obvious that very small values of ρ can only be suitable when a range search with small radius is placed. However, this was not typical for our data sets where the distance to the nearest neighbor was rather high – see in Figure 5.6 the relatively high costs to get the nearest neighbor. This implies that higher

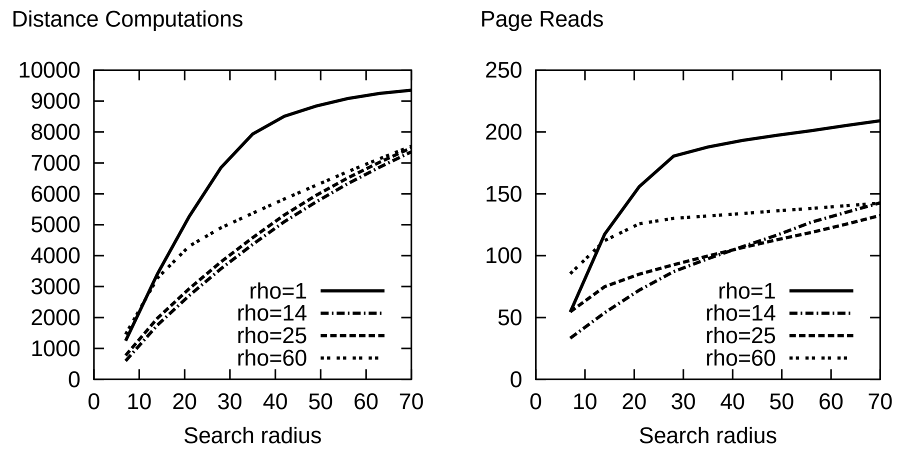


Figure 5.7: Range search on sentences using different D-Index structures

values of ρ, such as 14, are preferable. However, there are limits to increase such value, since the structure with $\rho = 25$ was only exceptionally better than the one with $\rho = 14$, and the performance with $\rho = 60$ was rather poor. The experiments demonstrate that the proper choice of structure can significantly influence the performance. The selection of optimized parameters is still an open research issue that we plan to investigate in the near future.

In the last group of tests, we analyzed the influence of the block size on system performance. By definition, the size of blocks must be able to accommodate any object of the indexed data set. However, we can have a bucket consisting from only one (large) block, which can be compared with the same bucket consisting from many small blocks. We have experimentally studied this issue on the VEC data set for both the nearest neighbor and the range search. The results are summarized in Figure 5.8.

In order to obtain an objective comparison, we express the cost as relative block reads, i.e. the percentage of blocks that was necessary to read for answering a query.

We can conclude from the experiments that the search with smaller blocks requires to read a smaller fraction of the entire data structure. Notice that the number of distance computations for different block sizes is practically constant and depends only on the

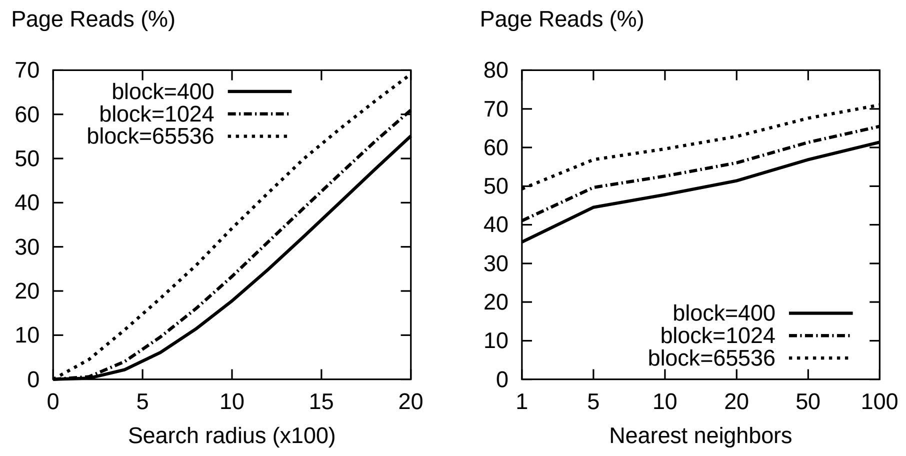


Figure 5.8: Search efficiency versus the block size

query. However, the small block size implies the large amount of blocks; this means that if the access cost to a block is significant (eg. blocks are allocated randomly) then large block sizes may become preferable. Nonetheless, when the blocks are stored continuously small blocks can be preferable because an optimized strategy for reading a set of disk pages such as one proposed in [77] can be applied. For a static file, we can easily achieve such the storage allocation of the D-Index through a simple reorganization.

5.4 D-Index Performance Comparison

We have also compared the performance of D-Index with other index structures under the same workload. In particular, we considered the M-tree[1] [25], the sequential organization (SEQ), and the Excluded Middle Vantage Point Forest (VPF) [88]. According to [22], M-Tree and SEQ are the only types of index structures for metric data that use a disk memory to store objects. We have also chosen VPF for the reason that the split strategy utilized in the D-Index is the modified approach of VPF. In order to maximize the objectivity of comparison, the actual performance is measured in terms of the number of distance computations and I/O operations. All index structures ex-

[1]The software is available at http://www-db.deis.unibo.it/research/Mtree/

cept VPF are implemented using the GIST package [49] and the I/O costs are measured in terms of the number of accessed disk blocks – the basic access unit of disk memory, which had the same size for all index structures. Because VPF is the only method of compared indexes that does not store data on disks the number of disk accesses for VPF is not provided in the results.

The main objective of these experiments was to compare the similarity search efficiency of the D-Index with the other organizations. We have also considered the space efficiency, costs to build an index, and the scalability to process growing data collections.

5.4.1 Search efficiency for different data sets

We have built the D-Index, M-tree, VPF, and SEQ for all the data sets VEC, URL, and STR. The M-tree was built by using the bulk-load package and the Min-Max splitting strategy. This approach was found in [25] as the most efficient to construct the tree. The algorithm presented by authors of [13] was used to select pivots for the Vantage Point Forest (VPF) access structure. This algorithm prefers pivots that are far away from each other, so-called *outliers*. We have measured average performance over fifty different query objects considering numerous similarity range and nearest neighbor queries. The results for the range search are shown in Figures 5.9 and 5.10, while the performance for the nearest neighbor search is presented in Figures 5.11 and 5.12.

For all tested queries, i.e. retrieving subsets up to 20% of the database, the M-tree, the VPF, and the D-Index always needed less distance computations than the sequential scan. However, this was not true for the block accesses, where even the sequential scan was typically more efficient than the M-tree. In general, the number of block reads of the M-tree was significantly higher than for the D-Index. Only for the URL sets when retrieving large subsets, the number of block reads of the D-Index exceeded the SEQ. In this situation, the M-tree accessed nearly three times more blocks than the D-Index or SEQ. On the other hand, the M-tree and the D-Index required for this search case much less distance computations than the SEQ.

Figure 5.10 demonstrates another interesting observation: to run the exact match query, i.e. range search with $r = 0$, the D-Index only

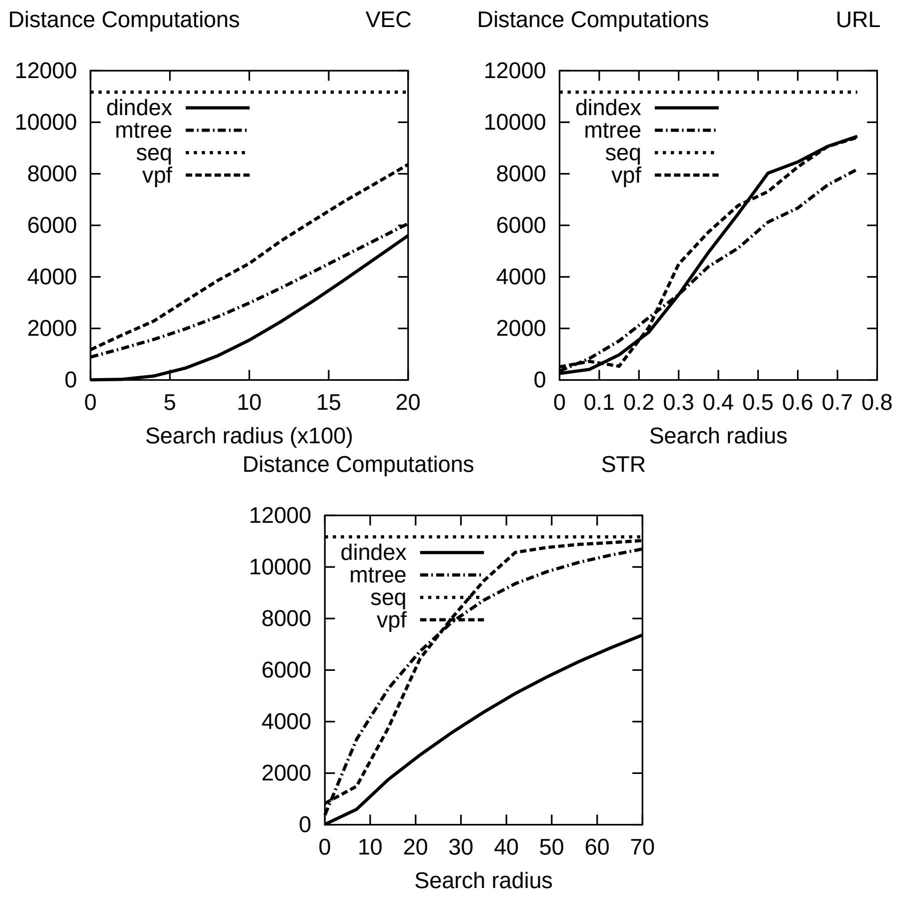


Figure 5.9: Comparison of the range search efficiency in the number of distance computations for VEC, URL, and STR.

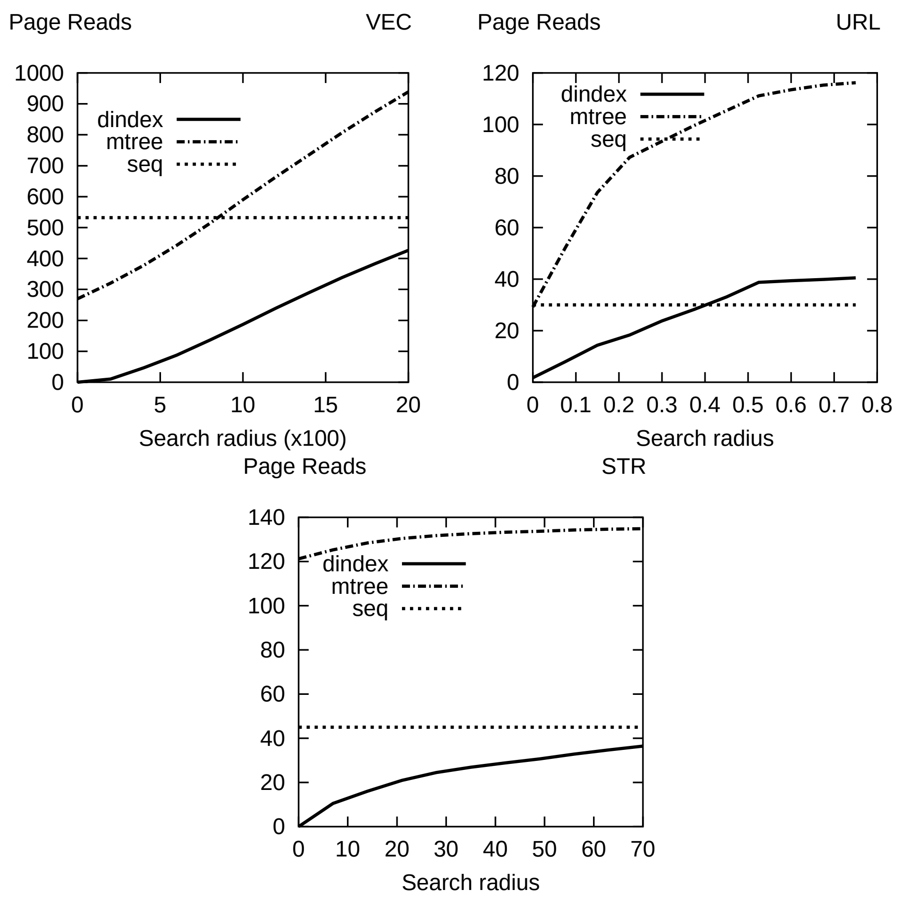


Figure 5.10: Comparison of the range search efficiency in the number of block reads for VEC, URL, and STR.

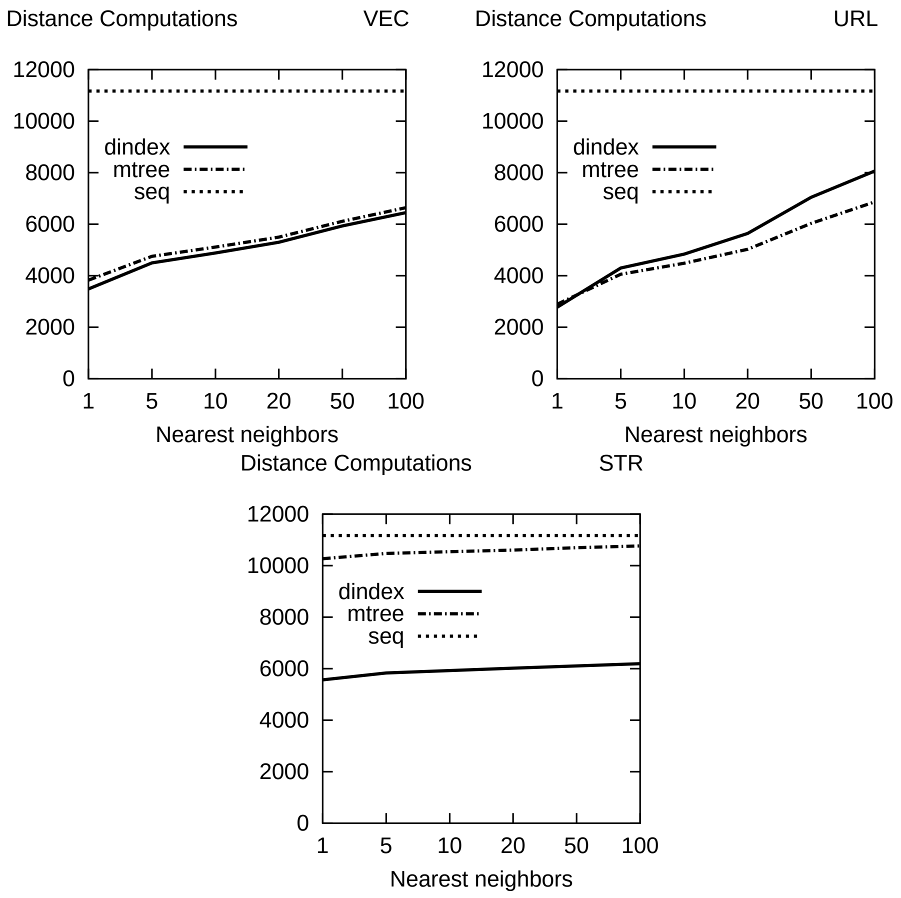


Figure 5.11: Comparison of the nearest neighbor search efficiency in the number of distance computations for VEC, URL, and STR.

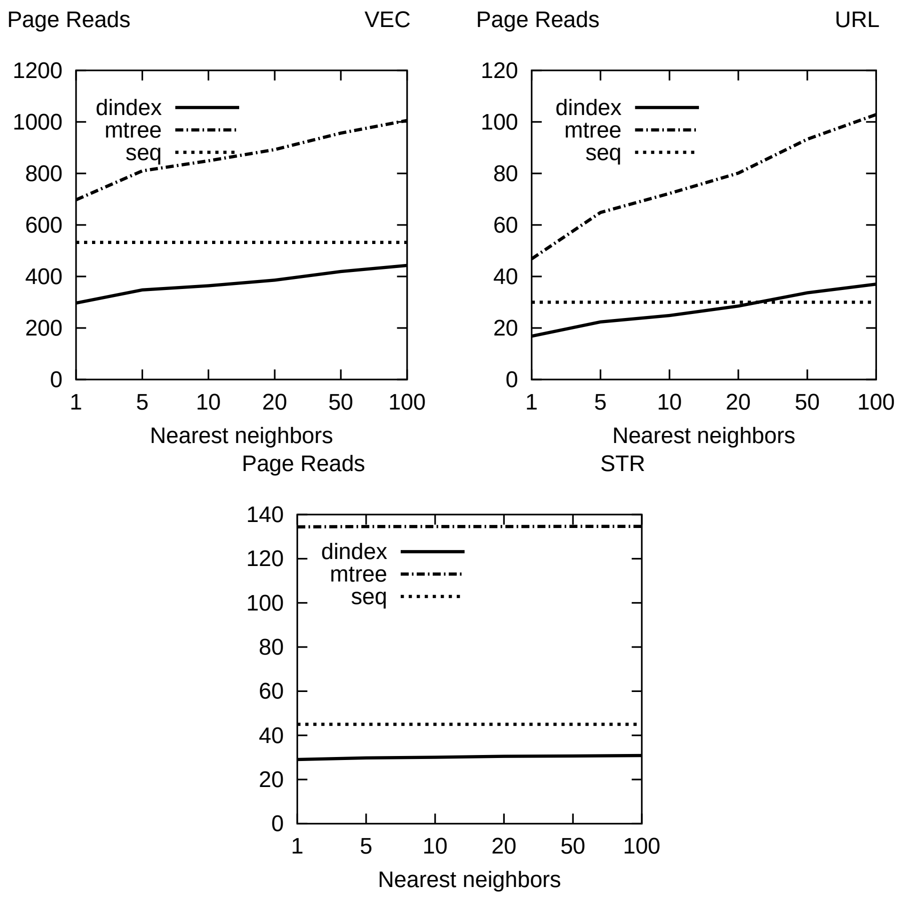


Figure 5.12: Comparison of the nearest neighbor search efficiency in the number of block reads for VEC, URL, and STR.

needs to access one block. As a comparison, see (in the same figure) the number of block reads for the M-tree: they are one half of the SEQ for vectors, equal to the SEQ for the URL sets, and even three times more than the SEQ for the sentences. Notice that the exact match search is important when a specific object is to be eliminated – the location of the deleted object forms the main cost. In this respect, the D-Index is able to manage deletions more efficiently than the M-tree. We did not verify this fact by explicit experiments because the available version of the M-tree does not support deletions. The outcome of the same test with $r = 0$ in terms of distance computations is summarized in the following table. The conclusion is that the D-Index is very efficient in insertions or deletions of objects in comparison to other techniques.

	# of distance computations		
$r_q = 0$	STR	URL	VEC
D-Index	12	254	5
M-tree	375	349	270
VPF	829	512	1176
SEQ	11,169	11,169	11,169

Concerning the block reads, the D-Index looks to be significantly better than the M-tree, while comparison on the level of distance computations is not so clear. The D-Index certainly performed much better for the sentences and also for the range search over vectors. However, the nearest neighbor search on vectors needed practically the same number of distance computations. For the URL sets, the M-tree was on average only slightly better than the D-Index. On VEC data set, the VPF performed slowly than the M-tree, and the D-Index as well. Nonetheless, the VPF was faster than the M-tree for smaller radii on sentences. The most obscure situation is again for URL data set where the VPF performed even better that the D-Index in several isolated cases. In general, we conducted the experiments with the VPF only for the range queries because the results in Figure 5.9 revealed that the M-tree outperforms the VPF.

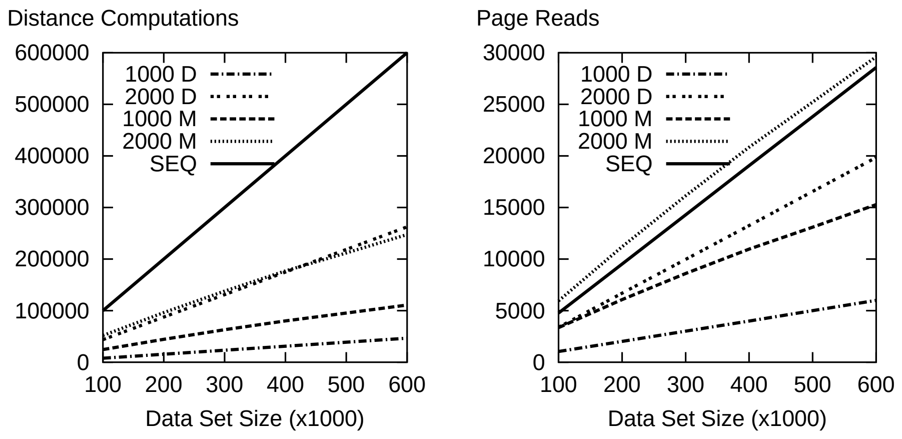


Figure 5.13: Range search scalability.

5.4.2 Search, space, and insertion scalability

To measure the scalability of the D-Index, M-tree, and SEQ, we have used the 45-dimensional color feature histograms. Particularly, we have considered collections of vectors ranging from 100,000 to 600,000 elements. For these experiments, the structure of D-Index was defined using thirty-seven reference objects and seventy-four buckets, where only the number of blocks in buckets was increasing to deal with growing files. The results for several typical nearest neighbor queries are reported in Figure 5.14. The execution costs for range queries are reported in Figure 5.13. Except for the SEQ organization, the individual curves are labeled with a number, representing either the number of nearest neighbors or the search radius, and a letter, where D stands for the D-Index and M for the M-tree. The size of query is not provided for the results of SEQ because the sequential organization has the same costs regardless whether the query is the range or the nearest neighbor.

We can observe that on the level of distance computations the D-Index is usually slightly better than the M-tree, but the differences are not significant – the D-Index and M-tree can save considerable number of distance computations compared to the SEQ. To solve a query, the M-tree needs significantly more block reads than the D-Index and for some queries, see 2000M in Figure 5.13, this number

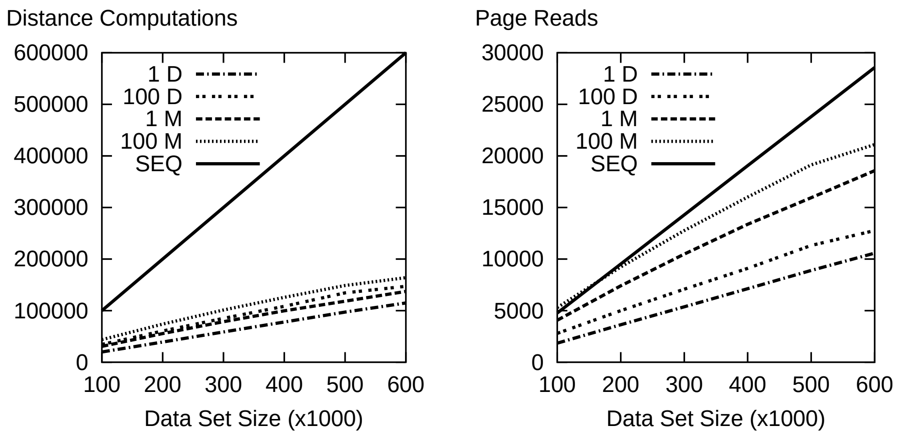


Figure 5.14: Nearest neighbor search scalability.

is even higher than for the SEQ. We have also investigated the performance of range queries with zero radius, i.e. the exact match. Independently of the data set size, the D-Index required one block access and eighteen distance comparisons. This was in a sharp contrast with the M-tree, which needed about 6,000 block reads and 20,000 distance computations to find the exact match in the set of 600,000 vectors. Moreover, the D-Index have constant costs to insert one object regardless the data size, which is in a sharp contrast with linear behavior of the M-tree. The details are summarized in the following table. The number of accessed disk pages for the D-Index varies from zero to one depending on whether the exact match search was successful or unsuccessful, i.e. whether the object was found or not.

Exact Match Search Costs				
	Distance Computations		**Accessed Blocks**	
Data Size	D-Index	M-tree	D-Index	M-tree
100k	18	5890	0-1	1324
200k	18	10792	0-1	2326
300k	18	15290	0-1	3255
400k	18	19451	0-1	4138
500k	18	23236	0-1	4910
600k	18	27054	0-1	5699

The search scale up of the D-Index was strictly linear. In this respect, the M-tree was even slightly better, because the execution costs for processing a two times larger file were not two times higher. This sublinear behavior should be attributed to the fact that the M-tree is incrementally reorganizing its structure by splitting blocks and, in this way, improving the clustering of data. On the other hand, the D-Index was using a constant bucket structure, where only the number of blocks was changing. Such observation is posing another research challenge, specifically, to build a D-Index structure with a dynamic number of buckets.

The disk space to store data was also increasing in all our organizations linearly. The D-Index needed about 20% more space than the SEQ. In contrast, the M-tree occupied two times more space than the SEQ. In order to insert one object, the D-Index evaluated on average the distance function eighteen times. That means that distances to (practically) one half of the thirty-seven reference objects were computed. For this operation, the D-Index performed one block read and, due to the dynamic bucket implementation using splits, a bit more than one block writes. The insertion into the M-tree was more expensive and required on average sixty distance computations, twelve block reads, and two block writes.

In summary, the D-Index is better index structure than the M-tree and SEQ because it saves many block accesses and distance computations when executing queries. An interesting property is the very fast response time to execute the exact match queries. D-Index is economical in space with only 20% overhead compared to the sequential approach and the two times less space requirements than necessary for the M-tree.

Chapter 6

eD-Index

The development of Internet services often requires an integration of heterogeneous sources of data. Such sources are typically unstructured whereas the intended services often require structured data. Once again, the main challenge is to provide consistent and error-free data, which implies the data cleaning, typically implemented by a sort of similarity join. In order to perform such tasks, similarity rules are specified to decide whether specific pieces of data may actually be the same things or not. A similar approach can also be applied to the copy detection. However, when the database is large, the data cleaning can take a long time, so the processing time (or the performance) is the most critical factor that can only be reduced by means of convenient similarity search indexes.

For example, consider a document collection of books and a collection of compact disk documents. A possible search request can require to find all pairs of books and compact disks which have similar titles. But the similarity joins are not only useful for text. Given a collection of time series of stocks, a relevant query can be report all pairs of stocks that are within distance μ from each other. Though the similarity join has always been considered as the basic similarity search operation, there are only few indexing techniques, most of them concentrating on vector spaces. In this paper, we consider the problem from much broader perspective and assume distance measures as metric functions. Such the view extends the range of possible data types to the multimedia dimension, which is typical for modern information retrieval systems.

The problem of approximate string processing has recently been studied in [42] in the context of data cleaning, that is, removing inconsistencies and errors from large data sets such as those occurring in data warehouses. A technique for building approximate string join capabilities on top of commercial databases has been proposed in [44]. The core idea of these approaches is to transform the difficult problem of approximate string matching into other search problems for which some more efficient solutions exist.

In this chapter, we systematically study the difficult problem of similarity join implementation. We define three categories of implementation strategies and propose two algorithms based on the D-Index.

6.1 Algorithm Categories

The problem of similarity joins is precisely defined in Section 2.3. In this section, we categorize solutions for similarity joins [34]. First, we define a measure s that is used for comparison of different algorithms. Next, we introduce filter-based, partition-based, and range query based algorithms.

6.1.1 Complexity

To compare algorithms for similarity joins, we measure the computational complexity through the number of distance evaluations. The similarity join can be evaluated by a simple algorithm, which computes $|X| \cdot |Y|$ distances between all pairs of objects. We call this approach the *Nested Loops* (NL). For simplicity and without loss of generality, we assume $N = |X| = |Y|$. Therefore, the complexity of the NL algorithm is N^2. For the similarity self join ($X \equiv Y$), we have to ignore the evaluation of an object with itself so that the number of computed distances is reduced by N. Moreover, due to the symmetric property of metric functions, it is not necessary to evaluate the pair (x_j, x_i) if we have already examined the pair (x_i, x_j). That means the number of distance evaluations is $N(N-1)/2$ for the similarity self join.

As a preliminary analysis of the expected performance, we specify three general approaches for solving the problem of the similarity join in the next section. The performance index used to compare these approaches is the *speedup* (s) with respect to the trivial NL algorithm. Generally, if an algorithm requires n distance evaluations to compute the similarity join, the speedup is defined as follows:

$$s = \frac{N^2}{n}.$$

In case of similarity self join, the speedup is given by

$$s = \frac{N(N-1)/2}{n} = \frac{N(N-1)}{2n}.$$

In the following, we present three categories of algorithms that are able to reduce the computational complexity:

1. The *Filter Based algorithms* (FB), which try to reduce the number of expensive distance computations by applying simplified distance functions,

2. the *Partition Based algorithms* (PB), which partition the data set into smaller subsets where the similarity join can be computed with a smaller cost,

3. and the *Range Query algorithms* (RQ), which for each object of one data set execute a range query in the other data set, of course, an algorithm for range queries can apply some filtering or partitioning techniques to speedup retrieval.

6.1.2 Filter Based Algorithms

The Filter Based algorithms eliminate all pairs (x, y) that do not satisfy the similarity join condition of Equation 2.1 by using the lower bound and the upper bound distances, $d_l(x, y)$ and $d_u(x, y)$, having the following property:

$$d_l(x, y) \leq d(x, y) \leq d_u(x, y).$$

The assumption is that the computation of d_l and d_u is much less expensive than the computation of d. Given a generic pair of objects (x_i, y_j), the algorithm first evaluates the distance $d_l(x_i, y_j)$. If $d_l(x_i, y_j) > \mu$ then $d(x_i, y_j) > \mu$ and the pair can be discarded. On the contrary, if $d_u(x_i, y_j) \leq \mu$ then the pair qualifies. If $d_l(x_i, y_j) \leq \mu$ and $d_u(x_i, y_j) > \mu$, it is necessary to compute $d(x_i, y_j)$. In the following, we present a sketch of the algorithm.

Algorithm 6.1.1 *Generic Filter Based Algorithm*

```
for $i = 1$ to $N$
    for $j = 1$ to $M$
        if $d_l(x_i, y_j) \leq \mu$ then
            if $d_u(x_i, y_j) \leq \mu$ then
                add $(x_i, y_j)$ to the result
            else if $d(x_i, y_j) \leq \mu$ then
                add $(x_i, y_j)$ to the result
            end if
        end if
    end for
end for
```

Let us suppose that p_u is the probability that $d_u(x_i, y_i) \leq \mu$ and p_l is the probability that $d_l(x_i, y_i) \leq \mu$. Moreover, let T_u, T_l, and T be the average computation times of d_u, d_l and d, respectively. The total computation time T_{tot} of the FB algorithm is

$$T_{tot} = T_l N^2 + p_l(T_u N^2 + (1 - p_u) T N^2),$$

and therefore

$$s = \frac{TN^2}{T_{tot}} = \frac{1}{\alpha_l + p_l(\alpha_u + (1 - p_u))}, \tag{6.1}$$

where $\alpha_u = \frac{T_u}{T}$ and $\alpha_l = \frac{T_l}{T}$. Notice that we have implicitly assumed that the events associated with p_u and p_l are independent, which is not too far from the reality. From this equation, it is possible to see that changing the order of evaluation of d_u and d_l, we can get different performance.

6.1.3 Partition Based Algorithms

This technique divides each of the data sets X and Y into h subsets with non-null intersections. Usually, the subsets are generated by using a set of regions of the whole domain $\mathcal{D}$. Let $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_h$ be the regions of partitioning, we define the subsets as $X_i = \{x \in X | x \in \mathcal{D}_i\}$ and $Y_i = \{y \in Y | y \in \mathcal{D}_i\}$, where $X = \bigcup_{i=1}^{h} X_i$ and $Y = \bigcup_{i=1}^{h} Y_i$. The idea is that neighboring regions overlap, so that by computing the similarity join in each local region no pair of objects is lost. Therefore, the algorithm proceeds by evaluating the similarity join for each pair of subsets X_i, Y_i for $1 \leq i \leq h$. The algorithm can be designed in a manner that it is able to recognize whether a pair has already been evaluated in another partition or not, therefore, the number of distance evaluations in a local region can be smaller than $|X_i| \cdot |Y_i|$. Let $n_i = |X_i|$ and $m_i = |Y_i|$; in the worst case the speedup is given by

$$s = \frac{N^2}{\sum_{i=1}^{h} n_i m_i}.$$

In case of similarity self join, the speedup is

$$s = \frac{N^2}{\sum_{i=1}^{h} n_i^2}.$$

The ideal case where objects are uniformly distributed among partitions is expressed by

$$s = \frac{N^2}{\sum_{i=1}^{h} (N/h)^2} = \frac{N^2}{h(N/h)^2} = h. \tag{6.2}$$

6.1.4 Range Query Algorithms

Algorithms based on RQ strategy use an access structure supporting the similarity range search in order to retrieve qualified pairs of objects. The idea is to perform N range queries for all objects in the database with the radius $r = \mu$. Given the average performance of a range query, for a specific access structure, it is easy to estimate the performance of the RQ algorithm. Therefore, if we have on average n_r distance computations for a range query, the speedup is given by

$$s = \frac{N^2}{N n_r} = \frac{N}{n_r}, \tag{6.3}$$

and

$$s = \frac{N(N-1)/2}{(N n_r/2)} = \frac{N-1}{n_r}, \tag{6.4}$$

for the similarity self join.

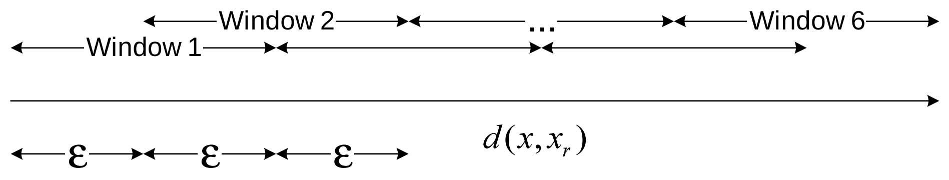


Figure 6.1: Illustration of the Sliding Algorithm.

6.2 Implementation and Performance Evaluation

In this section, we provide examples of implementations of the techniques described in the previous section, specifically, the Sliding Window algorithm for the partition based approach (PB) and the Pivot Based Filtering for the filtering based approach (FB). The implementation of the range query approach (RQ) is provided in Section 6.3. We also present a simplified performance evaluation.

6.2.1 Sliding Window Algorithm

The Sliding Window algorithm belongs to the Partition Based category. In the following, we describe the variant of this algorithm specialized to execute the similarity self join. The idea is straightforward (see Figure 6.1). We order the objects of the data set X with respect to an arbitrary reference object $x_r \in \mathcal{D}$, and we partition the objects in n_w overlapping "windows" each one having the width of 2μ. Notice that μ is the threshold of a similarity join query. The algorithm starts from the first window (partition) with extremes $[0, 2\mu]$ and centered in μ collecting all the pairs of objects (x_i, x_j) such that $d(x_i, x_j) \leq \mu$. The NL algorithm is used for each window. The algorithm proceeds with the next window, with extremes $[\mu, 3\mu]$ and centered in 2μ, however, the distance evaluation of a pair of objects which are both in the overlap with the previous window is avoided. The algorithm continues until the last window is reached. The detailed description of the algorithm requires the following definitions:

- Reference object (pivot): an object x_r from $\mathcal{D}$
- Maximum distance with respect to x_r: $d_r^+ = max_{i \in X}[d(x_r, x_i)]$

- The number of windows used: $n_w = \left\lfloor \frac{d_r^+}{\mu} \right\rfloor$
- The set of objects contained in the window m:

$$W_m = \{x_i \in X \mid (m-1)\mu < d(x_r, x_i) \leq (m+1)\mu\}$$

- The number of objects in W_m: $w_m = |W_m|$

Without loss of generality we study the algorithm for the case of similarity self join only.

Algorithm 6.2.1 *Sliding Window*

```
for $i = 1$ to $n_w$
    for $j = 1$ to $w_i$
        for $k = j + 1$ to $w_i$
            if $i > 1$
                if $d(x_j, x_r) > i\mu$ or $d(x_k, x_r) > i\mu$
                    if $d(x_j, x_k) \leq \mu$
                        add $(x_j, x_k)$ to the result set
                    end if
                end if
            else
                if $d(x_j, x_k) \leq \mu$
                    add $(x_j, x_k)$ to the result set
                end if
            end if
        end for
    end for
end for
```

Notice that the computation of distances $d(x_j, x_r)$ and $d(x_k, x_r)$ is done during the phase of ordering of objects with respect to x_r; they do not need to be computed again during the execution of the algorithm.

6.2.2 Pivot Based Filtering

Pivoting algorithms can be seen as a particular case of the FB category. In Figure 6.2, the basic principle of the pivoting technique is illustrated, where x is one object of a pair and p_i is a preselected object,

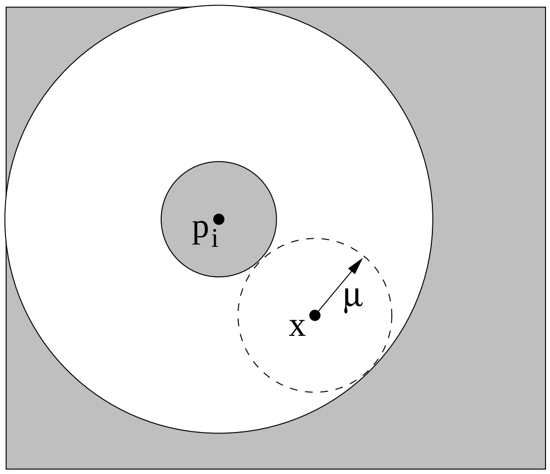


Figure 6.2: Example of pivots behavior.

called a pivot. Provided that the distance between any object and p_i is known, the gray area represents the region of objects y, that do not form a qualifying pair with x. This can easily be decided without actually computing the distance between x and y, by using the triangle inequalities $d(p_i, y) + d(x, y) \geq d(p_i, x)$ and $d(p_i, x) + d(p_i, y) \geq d(x, y)$ we have that $d(p_i, x) - d(p_i, y) \leq d(x, y) \leq d(p_i, y) + d(p_i, x)$, where $d(p_i, x)$ and $d(p_i, y)$ are precomputed. It is obvious that, by using more pivots we can improve the probability of excluding an object y without actually computing its distance to x. For further details, we refer to Section 2.6.

Dynamic Pivots

Since the evaluation of distances between pivots and all objects of the sets X and Y is very expensive, it is possible to generate the precomputed distances during the join elaboration. The idea is that each object of the data set X is promoted to a pivot during the elaboration of the join algorithm. For instance, when for all pairs (x_1, y_j) $(1 \leq j \leq M)$ distances $d(x_1, y_j)$ are computed, the object $x_1 \in X$ becomes a pivot.

Given K pivots $\{p_1, p_2, \ldots, p_K\}$, we can define a new lower bound distance d_l as $d_l(x, y) = \max_{k=1..K}(d_l^k(x, y)) = \max_{k=1..K}(|d(p_k, y) -$

$d(p_k, x)|)$, which is always less than or equal to $d(x, y)$. Naturally, this particular lower bound distance can be defined for every type of metric space; what we need is just a set of pivots with associated precomputed distances to objects of the sets X and Y. If we define p_l as the probability that $\forall k = 1..K, d_l^k(x, y) \leq \mu$, and we neglect the evaluation of d_l^k (which involves only a subtraction operation), we have

$$s = \frac{1}{p_l}.$$

6.2.3 Comparison

In order to compare the proposed approaches, we evaluate their speedups by using two simple data sets composed of vectors; the distances are measured using the Euclidean distance. The first set contains uniformly distributed vectors in a 10-dimensional hypercube with coordinates limited to $[0, 1]$, while the vectors in the second data set are clustered and the dimension of each vector is twenty.

The experiments were conducted for different values of μ ranging from 0.20 up to 1.00. Figures 6.3 and 6.4 show the results of experiments. The former presents values of speedup for the Pivot Based Filtering while the latter shows results for the Sliding Window algorithm.

The problem with all the algorithms for the similarity join is that their complexity is $O(N^2)$ or, in other words, that their speedups are constant with respect to N. Unfortunately, even if the speedup is high, when the number of objects grows the number of distance computations grows quadratically. For the FB algorithm (see Equation 6.1), the speedup depends on the costs of distances d_l and d_u, i.e. α_l and α_u, which do not depend on N. Moreover, the probabilities p_l and p_u depend on adopted distance functions and on the distribution of objects in a metric space. We can conclude that FB has the constant speedup. As for the PB algorithm, from Equation 6.2 it is clear that the speedup cannot be greater than h. The only promising algorithm could be the RQ. In fact, as Equation 6.3 shows, in principle the speedup grows linearly if n_r is constant. Unfortunately, in the access structures in the literature, n_r increases linearly with N and therefore the speedup is constant also in the RQ case.

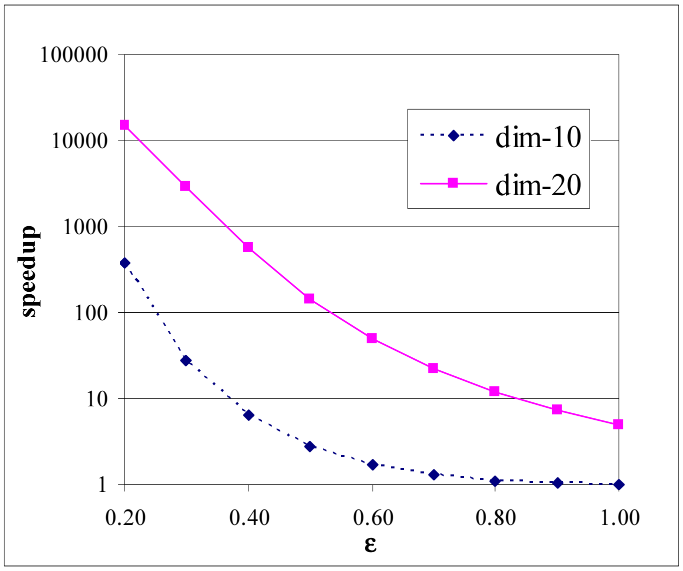


Figure 6.3: Performance of the Pivot Based Filtering algorithm on the 10 and 20 dimensional data sets.

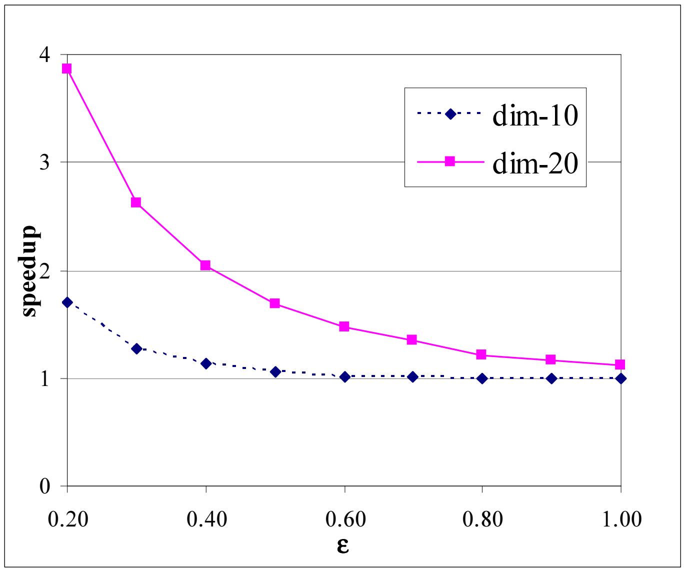


Figure 6.4: Performance of the Sliding Window algorithm on the 10 and 20 dimensional data sets.

In the next two sections, we propose and describe two specialized algorithms for similarity self join queries, the range query join algorithm and the overloading join algorithm. The former one is implemented using the D-Index structure [33] while the latter employs an extended version of the D-Index, called eD-Index [37, 36].

6.3 Range Query Join Algorithm

The D-Index access structure provides every efficient algorithm for executing similarity range queries, especially, when applied on small query radii that are often used for example in the data cleaning area. Thus, we define a new algorithm for similarity self joins based on range queries [35]. The algorithm is every straightforward it executes a series of range queries with the fixed radius equal to μ, that is, the threshold of a join query. Only the query object in range queries differs, i.e. every object stored in the D-Index structure is picked and the range query is processed with. In the following, we provide the code of the algorithm in a pseudo language.

Algorithm 6.3.1 *Range Query Join Algorithm*

```
# for every level
for $i = 1$ to $h$
    # for every bucket $B_{i,j}$ on the level $i$
    for $j = 1$ to $2^{m_i} - 1$
        # for every block of the bucket $B_{i,j}$
        for $k = 1$ to $b_{i,j}$
            # for every object $q$ in the block $B_{i,j}[k]$
            forall $q$ in $B_{i,j}[k]$
                execute $S = R(q, \mu)$
                $\forall o \in S$ : add the pair $(q, o)$ to the response set
            end forall
        end for
    end for
end for
# access the exclusion bucket
# for every block of the exclusion bucket $E_h$
for $k = 1$ to $e_h$
```

```
        # for every object $q$ in the block $E_h[k]$
        **forall** $q$ **in** $E_h[k]$
            *execute* $S = R(q, \mu)$
            $\forall o \in S$ : *add the pair* $(q, o)$ *to the response set*
        **end forall**
    **end for**
```

This algorithm proceeds from the first level to the last level h. On each level i, it accesses all buckets $B_{i,1} \ldots B_{i,2^{m_i}-1}$. The bucket $B_{i,j}$ consists of $b_{i,j}$ blocks where objects are stored. The algorithm continues by reading all blocks $B_{i,j}[1] \ldots B_{i,j}[b_{i,j}]$ of the bucket $B_{i,j}$ and every object q stored in the block $B_{i,j}[k]$ is used in the range query $R(q, \mu)$, which is then executed on the same D-Index structure. The result set S of the range query contains all objects $o \in X$ such that $d(q, o) \leq \mu$. We pair up all objects of S with the query object q and add the pairs to the response set of the similarity self join query $SJ(\mu)$. Finally, we process objects stored in the global exclusion bucket. This way the algorithm processes a join query.

However, the range query join algorithm has the drawback that the reported response set contains duplicate pairs of objects. For example, assume that we execute a range query for the object q that resides in the bucket $B_{1,1}$. The query returns the set $o_1, \ldots, o_i, \ldots, o_n$. Providing that the object o_i is stored in the bucket $B_{3,2}$, i.e. on the third level, when we process objects in the bucket $B_{3,2}$ we execute a range query for o_ii, which indeed returns the object q in its response set. As a result, we first report the pair (q, o_i) and then return the pair (o_i, q). In general, each pair (o_i, o_j) satisfying the join query $SJ(\mu)$ is present twice in the response set returned by this algorithm once in the form (o_i, o_j) and next as (o_j, o_i). Thus, the response set needs additional cleaning. Assuming that each object has an unique identification, which is a linear ordering (objects can be sorted) we can insert a pair (o_i, o_j) into the response in the ordered form, i.e. $ID(o_i) <= ID(o_j)$. In this respect, the post-processing cleaning phase would only need sorting pairs by a kind of sort algorithm that requires $\mathcal{O}(n \log n)$. As a result, the duplicated pairs are at even positions in the sorted list.

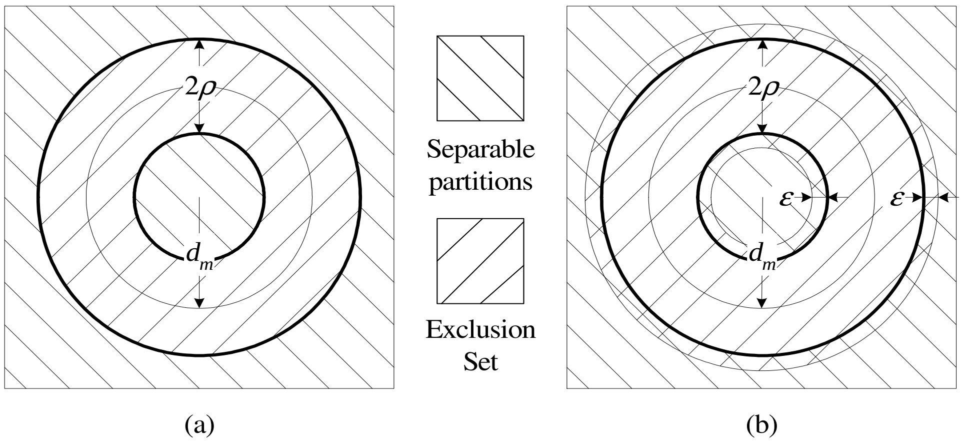


Figure 6.5: The modified bps split function: (a) original ρ-split function; (b) modified ρ-split function.

6.4 eD-Index Access Structure

In this section, we propose a new access structure, which supports not only range and k-nearest neighbor queries but even similarity joins, specifically, similarity self joins. The access structure extends the D-Index and is called eD-Index [37].

The idea behind the eD-Index is to modify the ρ-split function so that the exclusion set and separable sets overlap of distance ϵ, $0 \leq \epsilon \leq 2\rho$. Figure 6.5 depicts the modified ρ-split function. The objects that belong to both the separable and the exclusion sets are replicated. This principle, called the *exclusion set overloading*, ensures that there always exists a bucket for every qualifying pair $(x, y)|d(x, y) \leq \mu \leq \epsilon$ where both the objects x, y occur. As explained later, a special algorithm is used to efficiently find these buckets and avoid access to duplicates. In this way, the eD-Index speeds up the evaluation of similarity self joins.

Formally, the eD-Index access structure uses the same ρ-split functions and the overloading principle is implemented by manipulations with the parameter ρ of split functions. To fulfill all the requirements of replication, we modify the insertion algorithm. The procedure of storing a new object $x \in X$ in the eD-Index structure $eDI^{\rho,\epsilon}(X, m_1, m_2, \ldots, m_h)$ proceeds according to the following algorithm.

Algorithm 6.4.1 *Insertion*

```
for $i = 1$ to $h$
    if $\langle s_i^{m_i,\rho}(x)\rangle < 2^{m_i}$
        $x \mapsto B_{i,\langle s_i^{m_i,\rho}(x)\rangle}$;
        if $\langle s_i^{m_i,\rho+\epsilon}(x)\rangle < 2^{m_i}$
            exit;
        end if
        set_copy_flag(x);
    end if
end for
$x \mapsto E_h$;
```
■

Algorithm 6.4.1 starts at the first level and tries to find a separable bucket to accommodate x. If a suitable bucket exists, the object is stored in the bucket. If the object is outside the overloading area, i.e. $\langle s_i^{m_i,\rho+\epsilon}(x)\rangle < 2^{m_i}$, the algorithm stops. When the expression $\langle s_i^{m_i,\rho+\epsilon}(x)\rangle$ returns 2^{m_i} the object x must be replicated. The copy flag of x is set on and object is reinserted on a next level. Finally, if x[1] is not inserted on any level it is placed in the exclusion bucket E_h. Notice that the object x can be duplicated more than once depending on the occurrences in overloading areas. In the worst case, the object is placed in a bucket on the first level and is replicated in all successive level and the exclusion bucket as well. To insert one object, the algorithm needs to access at least one bucket, however, due to the overloading principle the cost can be higher. Nonetheless, the upper bound on the number of accessed buckets is $h+1$, that is, one bucket per level plus the exclusion bucket. As for the number of distance computations, the eD-Index needs $\sum_{i=1}^{j} m_i$ distance computations to store a new object, assuming that the new object or any of its duplicates is inserted into a separable bucket of j^{th} level. The maximum number of distance evaluations is limited by $\sum_{i=1}^{h} m_i$. One can object to the fact that the insertion algorithm calls a split function twice per level, however, during the first call the evaluated distances are stored thus the second call of the function can benefit from the saved distanced that need not be computed again.

[1]The original object x or any of its copies.

Because some objects are replicated in the eD-Index we also have to modify search algorithms. In particular, algorithms without any modifications would report also duplicates and such the behavior is not desirable.

6.4.1 Range Query Algorithm

In this section, we provide the modified algorithm for range queries. Given a query region $\mathcal{Q} = R(q, r)$ with $q \in \mathcal{D}$ and the query radius $r \leq d^+$. A range search algorithm executes the similarity range query as follows.

Algorithm 6.4.2 *Range Search*

```
01. for i=1 to h
02.     if $\langle s_i^{m_i,\rho+r}(q)\rangle < 2^{m_i}$ then
            # the query is exclusively contained in a separable bucket
03a.        $R = \{x | x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho+r}(q)\rangle}\}$;
03b.        return all objects $x$ such that $x \in R$ and $x$ has not the copy_flag switched on;
            exit;
04.     end if
05.     if $r \leq \rho$ then
            # search radius up to $\rho$
06.         if $\langle s_i^{m_i,\rho-r}(q)\rangle < 2^{m_i}$ then
                # the query is not exclusively contained in the exclusion bucket of the level $i$
07a.            $R = \{x | x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho-r}(q)\rangle}\}$;
07b.            return all objects $x$ such that $x \in R$ and $x$ has not the copy_flag switched on;
08.         end if
09.     else
            # search radius greater than $\rho$
10.         let$\{l_1, l_2, \ldots, l_k\} = G(s_i^{m_i,r-\rho}(q))$
11a.        $R = \{x | x \in \mathcal{Q} \cap B_{i,l_1}$ or $x \in \mathcal{Q} \cap B_{i,l_2}$ or $\ldots$ or $x \in \mathcal{Q} \cap B_{i,l_k}\}$;
```

```
11b.        return all objects $x$ such that $x \in R$ and $x$ has not the
              copy_flag switched on;
12.     end if
13. end for
14a. $R = \{x | x \in \mathcal{Q} \cap E_h\}$;
14b. return all objects $x$ such that $x \in R$ and $x$ has not the
       copy_flag switched on;
```

■

The detail description of Algorithm 6.4.2 is provided in Section 4.4.4 where Algorithm 4.4.3 is presented. This algorithm is a slight modification of Algorithm 4.4.3. Changes are made only in steps where qualifying objects are reported. Specifically, the original step 03 is expanded into two lines $03a$ and $03b$. On the line $03a$ we collect all objects that satisfy the query (this is the same as in the original algorithm) and we then discard the objects that are duplicates, i.e. they have the *copy_flag* set. The same modifications are made on the lines $07a, 07b$, $11a, 11b$, and $14a, 14b$.

6.4.2 Nearest Neighbors Search

In section 4.4.5, Algorithm 4.4.4 for k-nearest neighbors queries is specified. However, the eD-Index structure also needs slightly modified version of the algorithm. In the new version, we only change the fashion in which bucket are accessed. When a bucket is visited and all objects that satisfy the constraint by a range query are reported we filter out every object that is marked as a copy. For further details about accessing buckets see the description of the original algorithm in Section 4.4.5.

6.4.3 Similarity Self Join Algorithm

The outline of the similarity self join algorithm is following: execute the join query independently on every separable bucket of every level of the eD-Index and additionally on the exclusion bucket of the whole structure. This behavior is correct due to the exclusion set overloading principle – every object of a separable set that can make a qualifying pair with an object of the exclusion set of a level is copied to the exclusion set. The partial results are then concatenated and form the final answer.

Given a similarity self join query $SJ(\mu)$, the similarity self join algorithm which processes sub-queries in individual buckets is based on the sliding window algorithm. The idea of this algorithm is straightforward, see Figure 6.6. All objects of a bucket are or-

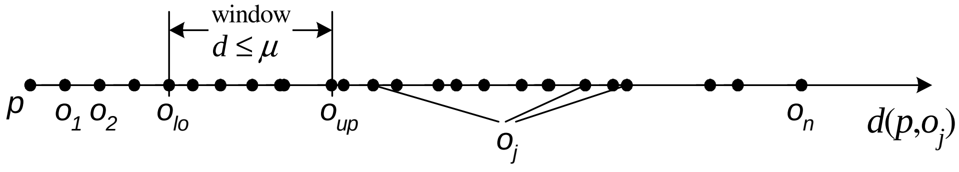


Figure 6.6: The Sliding Window algorithm.

dered with respect to a pivot p, which is the reference object of a ρ-split function used by the eD-Index. We then define a sliding window of objects $[o_{lo}, o_{up}]$ that always satisfies the constraint: $d(p, o_{up}) - d(p, o_{lo}) \leq \mu$, i.e. the window's width is $\leq \mu$. Algorithm 6.4.3 starts with the window $[o_1, o_2]$ and successively moves the upper bound of the window up by one object while the lower bound is increased to preserve the window's width $\leq \mu$. The algorithm terminates when the last object o_n is reached. All pairs (o_j, o_{up}), such that $lo \leq j < up$, are collected in each window $[o_{lo}, o_{up}]$ and, at the same time, the applied pivot-based strategy significantly reduces the number of pairs which must be checked. Finally, all qualifying pairs are reported.

Algorithm 6.4.3 *Sliding Window*

```
$lo = 1$
**for** $up = 2$ **to** $n$
    # move the lower boundary up to preserve window's
    width $\leq \mu$
    **increment** $lo$ **while** $d(o_{up}, p) - d(o_{lo}, p) > \mu$

    **for** $j = lo$ **to** $up - 1$
        # for all objects in the window
        **if** *PivotCheck()* = *FALSE* **then**
```

```
            # apply the pivot-based strategy
            compute $d(o_j, o_{up})$
            if $d(o_j, o_{up}) \leq \mu$ then
                add pair $(o_j, o_{up})$ to result
            end if
        end if
    end for
end for
```

The eD-Index structure also stores distances between stored objects and reference objects of ρ-split functions. Remind that these distances are computed when objects are inserted into the structure and they are utilized by the pivot-based strategy, see Section 2.6 for details. The function *PivotCheck()* in the algorithm uses not only the pivots of ρ-split functions but it also utilizes the dynamic pivot principle described in Section 6.2.2. In particular, the algorithm promotes the upper boundary of the current sliding window (i.e. the object o_{up}) to the dynamic pivot. Once the current dynamic pivot goes out of scope, that is, out of the sliding window, the upper boundary is selected as the new dynamic pivot again.

The application of the exclusion set overloading principle implies two important issues. The former one concerns the problem of duplicate pairs in the result of a join query. This fact is caused by the copies of objects that are reinserted into the exclusion set. The described algorithm evades this behavior by coloring object's duplicates. Precisely, each level of the eD-Index has its unique color and every duplicate of an object has colors of all the preceding levels where the replicated object is stored. Figure 6.7 provides with an example of an eD-Index structure with 4 objects emphasized (a circle, a square, a triangle, and a hexagon). For example, the circle is replicated and stored at levels 1, 3, and 6. The circle at level 1 has no color because it is not a duplicate while the circle at level 3 has color of level 1 because it has already been stored at level 1. Similarly, the circle at level 6 is colored with red and blue, since it is stored at the preceding levels 1 and 3. By analogy, other objects are marked. Notice that the exclusion bucket has not specific color assigned, however, objects stored in the exclusion bucket can have colors if they are copies (e.g. see the square). The triangle stored at the second level is not colored

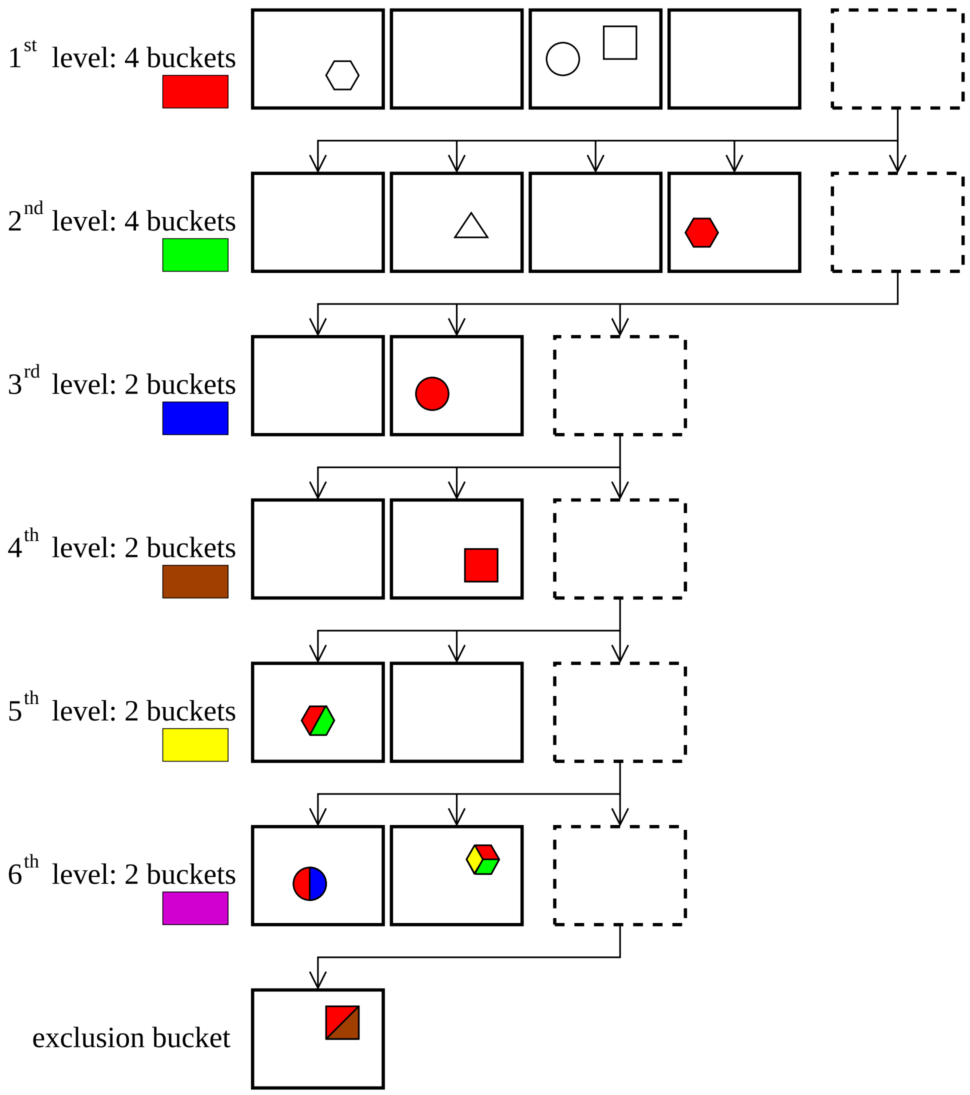


Figure 6.7: Coloring technique applied in the eD-Index.

because it is not accommodated at any preceding levels. Before the algorithm examines a pair it decides whether the objects of the pair share any color. If they have at least one color in common the pair is eliminated. The concept of sharing a color by two objects means that these objects are stored at the same level thus they are checked in a bucket of that level. For convenience, see the circle and the hexagon at the level 6, which share the red color. This is the consequence that both these objects are also stored at the first level.

The latter issue limits the value of parameter ρ, that is, $2\rho \geq \epsilon$. If $\epsilon > 2\rho$ some qualifying pairs are not examined by the algorithm. In detail, a pair is missed if one object is from one separable set while the other object of the pair is from another separable set. Such the pairs cannot be found with this algorithm because the exclusion set overloading principle does not duplicate objects among separable sets. Consequently, the separable sets are not contrasted enough to avoid missing some qualifying pairs.

Chapter 7

eD-Index Evaluation

7.1 Performance Evaluation

In order to demonstrate suitability of the eD-Index to the problem of similarity self join, we have compared several different approaches to join operation. The naive algorithm strictly follows the definition of similarity join and computes the Cartesian product between two sets to decide the pairs of objects that must be checked on the threshold μ. Considering the similarity self join, this algorithm has the time complexity $O(N^2)$, where $N = |X|$. A more efficient implementation, called the nested loops, uses the symmetric property of metric distance functions for pruning some pairs. The time complexity is $O(\frac{N \cdot (N-1)}{2})$. More sophisticated methods use prefiltering strategies to discard dissimilar pairs without actually computing distances between them. A representative of these algorithms is the range query join algorithm applied on the eD-Index. Finally, the last compared method is the overloading join algorithm, which is described in Section 6.4.3.

We have conducted experiments on two real application environments. The first data set consisted of sentences of Czech language corpus compared by the edit distance measure, so-called Levenshtein distance [68]. The most frequent distance was around 100 and the longest distance was 500, equal to the length of the longest sentence. The second data set was composed of 45-dimensional vectors of color features extracted from images. Vectors were compared by the quadratic form distance measure. The distance distribution of this data set was practically normal distribution with the most frequent distance equal to 4,100 and the maximum distance equal to 8,100. Figure 5.1 shows distance distributions in graphs.

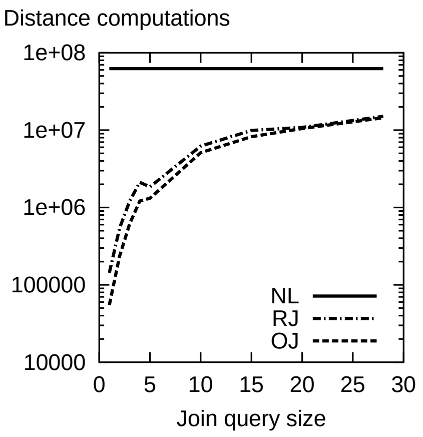


Figure 7.1: Join queries on the text data set.

In all experiments, we have compared three different techniques for the problem of the similarity self join, the nested loops (NL) algorithm, the range query join (RJ) algorithm applied on the eD-Index, and the overloading join (OJ) algorithm, again applied on the eD-Index.

7.1.1 Join-cost ratio

The objective of this group of tests was to study the relationship between the query size (threshold, radius, or selectivity) and the search costs measured in terms of distance computations. The experiments were conducted on both the data sets each consisting of 11,169 objects. The eD-Index structure used on the text data set consisted of nine levels and thirty-nine buckets. The structure for the vector data collection contained eleven levels and twenty-one buckets. Both the structures were fixed for all experiments.

We have tested several query radii up to $\mu = 28$ for the text set and up to $\mu = 1,800$ for the vector data. The similarity self join operation retrieved about 900,000 text pairs for $\mu = 28$ and 1,000,000 pairs of vectors for $\mu = 1,800$, which is much more than being interesting. Figure 7.1 shows results of experiments for the text data set. As expected, the number of distance computations performed by RJ and OJ increases quite fast with growing μ. However, RJ and OJ al-

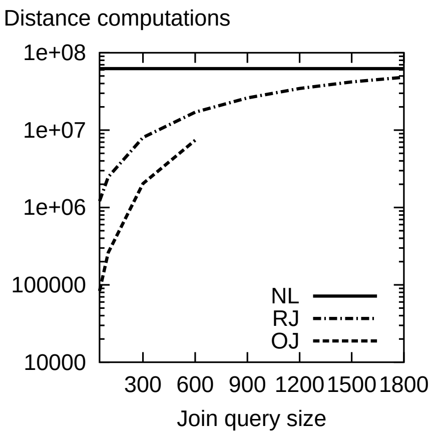


Figure 7.2: Join queries on vectors.

gorithms are still more than four times faster then NL algorithm for $\mu = 28$. OJ algorithm has nearly exceeded the performance of RJ algorithm for large $\mu > 20$. Nevertheless, OJ is more than twice faster than RJ for small values of $\mu \leq 4$, which are used in data cleaning area. Figure 7.2 demonstrates results for the vector data collection. The number of distance computations executed by RJ and OJ has the similar trend as for the text data set – it grows quite fast and it nearly exceeds the performance of NL algorithm. However, the OJ algorithm performed even better than RJ comparing to the results for the text data. Especially, OJ is fifteen times and nine times more efficient than RJ for $\mu = 50$ and $\mu = 100$, respectively. Tables 7.1, 7.2 presents the exact values of speedup measured for both text and vector datasets.

The results for OJ are presented only for radii $\mu \leq 600$. This limitation is caused by the distance distribution of the vector data set. Specifically, we have to choose values of ϵ and ρ at least equal to $1,800$ and 900, respectively, for being able to run join queries with $\mu = 1,800$. This implies that more than 80% of the whole data set is duplicated at each level of the eD-Index structure, this means that the exclusion bucket of the whole structure contains practically all objects of the data set, thus, the data set is indivisible in this respect. However, this behavior does not apply to small values of μ and ϵ where only small portions of data sets were duplicated.

Radius	Speedup	
	RJ	OJ
1	435.2	1136.0
2	115.5	266.1
3	51.5	102.3
4	29.5	52.7
5	33.7	49.2
10	10.0	13.8
15	6.3	9.2
20	5.8	7.3
28	4.1	5.8

Table 7.1: Join Speedup for text

Radius	Speedup	
	RJ	OJ
50	51.6	748.7
100	25.2	239.1
300	7.8	30.6
600	3.7	8.4
900	2.4	
1200	1.8	
1500	1.5	
1800	1.3	

Table 7.2: Join Speedup for vectors

	Speedup	
Radius	RJ	OJ
1	274.3	448.8
2	100.0	195.5
3	48.2	89.1
4	28.3	48.1
5	31.0	42.2
10	9.8	11.8
15	6.2	7.4
20	5.7	5.9
28	4.1	4.3

Table 7.3: Join Speedup for text with building costs included

This group of experiments was aimed at situations where similarity join operations are performed repetitively on the same structure. The data stored in the structure can, of course, vary, i.e. insertions and deletions can be performed. In the following tables 7.3, 7.4 we present the same results, however, the numbers of distance computations include the costs needed to create the structure. In particular, the presented values are numbers of distance function evaluations made during the join query execution and during building the structure. To ensure objective comparison, the tables quote values of speedup. The results are mainly influenced by building costs for small queries while the speedup is more stable for larger values of the query threshold. In general, the eD-Index is applicable in both the situations when a similarity join query is performed only once or the query is repeated.

7.1.2 Scalability

The scalability is probably the most important issue to investigate considering the web-based dimension of data. In the elementary case, it is necessary to study what happens with the performance of algorithms when the size of a data set grows. We have experimentally investigated the behavior of the eD-Index on the text data set with sizes from 50,000 to 250,000 objects (sentences).

We have mainly concentrated on small queries, which are typical

	Speedup	
Radius	RJ	OJ
50	50.0	521.0
100	24.8	209.8
300	7.7	30.1
600	3.7	8.4
900	2.4	
1200	1.8	
1500	1.5	
1800	1.3	

Table 7.4: Join Speedup for vectors with building costs included

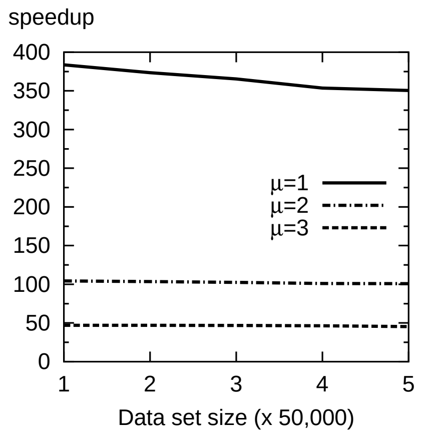


Figure 7.3: RJ algorithm scalability.

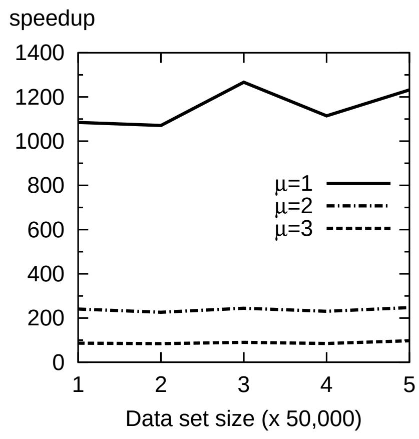


Figure 7.4: OJ algorithm scalability.

for data cleaning area. Figure 7.3 and Figure 7.4 report the speedup (s) of RJ and OJ algorithms, respectively. The results indicate that both RJ and OJ have practically constant speedup when the size of data set is significantly increasing. The exceptions are the values for $\mu = 1$ where RJ slightly deteriorates while OJ improves its performance. Nevertheless, OJ performs at least twice faster than RJ algorithm.

In summary, the figures demonstrate that the speedup is very high and constant for different values of μ with respect to the data set size. This implies that the similarity self join with the eD-Index, specifically the overloading join algorithm, is also suitable for large and growing data sets.

Chapter 8

Conclusion

Contrary to the traditional database approach, the Information Retrieval community has always considered search results as a ranked list of objects. Given a query, some objects are more relevant to the query specification than the others and users are typically interested in the most relevant objects, that is the objects with the highest ranks. This search paradigm has recently been generalized into a model in which a set of objects can only be pair-wise compared through a distance measure satisfying the metric space properties. Metric spaces have recently become an important paradigm for similarity search and many index structures supporting execution of similarity queries have been proposed. However, most of the existing structures are limited to operate on main memory only, so they do not scale up to high volumes of data. In this dissertation, we have concentrated on the case where indexed data are stored on disk memories.

8.1 Summary

In the first stage, we provide the necessities for understanding the metric spaces. We also give examples of distance measures to illustrate the wide applicability of structures based on metric spaces. Next, we define several similarity queries – the simplest is range search, the nearest neighbors search, its reverse variant and similarity join queries. Combinations of these queries are also concerned. The most important part of this chapter presents fundamental concepts of using metric postulates to increase the search efficiency. We formulate several lemmata supplemented with mathematical proofs. All indexing techniques in metric spaces utilize at least one of partitioning and filtering algorithms. We provide the reader with explanations of ideas of these methods. Finally, we mention the problem of selecting reference elements because all indexes need directly or indirectly some fixed points for partitioning or filtering.

In the next chapter, the survey of existing solutions for metric spaces is rendered. We start with the ball partitioning principle such as Fixed Queries Trees or Vantage Point Trees. Next, we concentrate on methods based on the generalized hyperplane partitioning. A separate section is dedicated to probably the most successful structure called M-Tree. To conclude this part, we focus on matrix-based approaches and we also shortly discuss the problem of transforming metric spaces to vector spaces.

The remaining parts present the contributions of this dissertation. In Chapter 4, we propose the novel access structure D-Index, which is able to deal with large archives because it supports disks. As a result, the D-Index reduces both I/O and CPU costs. The organization stores data objects in buckets with direct access to avoid hierarchical bucket dependencies. Such the organization also results in a very efficient insertion and deletion of objects. Though similarity constraints of queries can be defined arbitrarily, the structure is extremely efficient for queries searching for very close objects. The next chapter contains results of exhaustive experiments that have been conducted to verify D-Index properties. We have also compared the D-Index with other approaches, such as the M-Tree, the Excluded Middle Vantage Point Forest and a sequential scan.

In the next chapter, we concentrated on the problem of similarity joins, which are applicable in the areas of data cleaning or copy detection. First, we start with the theoretical overview of individual approaches and categorize them. We then study and compare possible implementations. Next, we propose a modification of original D-Index called eD-Index. This structure supports not only range and k-nearest neighbor queries but also similarity joins. The idea incorporated is that we duplicate some objects to allow to perform the join operation independently in buckets. This work concludes with the evaluation of eD-Index.

8.2 Contribution

This thesis contributes two novel access structures D-Index and eD-Index to the IR and DB community. The D-Index access structure synergistically combines more principles to a single system in order to minimize both I/O and CPU costs, that is, it reduces the number of accessed data and the number of distance evaluations needed to retrieve them. This new technique recursively divides the metric space into separable partitions (buckets) of data blocks to create a multi-level structure. At the same time, the applied pivot-based strategies (filtering algorithms) drastically decrease I/O costs. We sum up the most important properties of D-Index in the following:

- A new object is typically inserted with one block access. Since blocks are of fixed size the accessed block can overflow and splitting of the block leads to another block access.

- The number of distance computations between pivots and inserted objects is upper-bounded by $\sum_{i=1}^{h} m_i$.

- For all queries such that the most dissimilar object is within the distance of ρ from the query object, the number of accessed buckets is $h+1$ at maximum.

- For range queries with radius $r = 0$, i.e. the exact match, the successful search is typically solved with one block access and the unsuccessful search usually does not require any access. This is the very desired property because the exact match is used for locating an object to delete.

- For search radii greater than ρ, the number of visited buckets is higher than $h+1$, and for very large search radii, the query evaluation can eventually access all buckets of the structure.

- The number of distance computations between the query object and pivots is determined by the highest level of an accessed bucket. We can upper-bound the number of distance computations by the expression $\sum_{i=1}^{h} m_i$.

- There is also a possibility of *approximate similarity search*. If we limit the search radius r to the ρ regardless the fact that its actual value is greater than ρ we employ the property that the D-Index visits $h+1$ buckets at maximum. In this case, the costs are upper-bounded, however, we receive only partial results.

- The multilevel design of D-Index inherently offers parallel and distributed implementations. The simplest option is to allocate each level on a separate site.

- The D-Index access structure is not limited only to vectors or strings. It is applicable on any data, however, the distance function must satisfy the metric postulates.

Contrary to other index structures, such as the M-tree, the D-Index stores and deletes any object with one block access cost, so it is particularly suitable for dynamic data environments. Compared to the M-tree, it typically needs less distance computations and much less disk reads to execute a query. The D-Index is also economical in space requirements. It needs slightly more space than a sequential organization, but at least two times less disk space compared to the M-tree. As experiments confirm, it scales up well to large data volumes.

We have also concerned the problem of similarity joins. This issue did not attract much attention of research community. Nonetheless, a few studies on indexing of similarity joins have been recently published. We have developed new algorithms for similarity joins, especially, for similarity self joins. The new structure eD-Index is based on the D-Index and it extends its domain of supported queries with similarity self joins. The idea behind the eD-Index is to modify the partitioning so that the exclusion bucket overlaps with separable buckets and some objects are replicated. This principle ensures that there exists a bucket for every pair qualifying the query $SJ(\mu)$ where the pair occurs. Consequently, we can execute similarity joins in each bucket separately. A special algorithm is used to efficiently find buckets, which must be accessed, and avoid access to replicated pairs. In this way, the eD-Index speeds up the evaluation of similarity joins. To conclude, we summarize properties of the eD-Index:

- preserves all advantages of D-Index structure,

- extremely efficient for small radii of similarity joins where practically on-line response times are guaranteed,
- never worse in comparison with specialized techniques [42, 44].

8.3 Research Directions

Many interesting research directions are ahead.

- Recently, many researchers recognized the urgent need to deal with large data archives and they focus on the design of distributed search engines that would support lower response times and would scale up well to the future needs. This, of course, relates to the fast development of computer networks and the availability of broadband connections that allow such the systems. The D-Index structure is designed as a multilevel structure that offers parallel and distributed implementations. There are several ways of distributing the structure on several disks or computers. We will exploit the ideas in the future.
- The D-Index is able to cope with dynamic data but its structure is rather static and does not change. In case of difficulties the structure can be redesigned and data reinserted. Our future work will concern this issue and we plan to develop the D-Index with dynamic structure to avoid expensive reorganizations.
- Recent works have highlighted the importance of reverse nearest neighbor queries (see Section 2.3) in many application areas, such as mobile networks, decision support systems, and document repositories. This type of similarity query is very challenging and we will focus on it in the near future.
- To support usage of metric space indexes in the computer industry it is necessary to provide implementations for common database systems, e.g. PostgreSQL.
- The next goal is to verify properties of proposed structures in real and functional applications. We would like to develop

a system for image retrieval that would support similarity queries.

Bibliography

[1] Helmut Alt, Bernd Behrends, and Johannes Blömer. Approximate matching of polygonal shapes (extended abstract). In *Proceedings of the seventh annual symposium on Computational geometry*, pages 186–193. ACM Press, 1991.

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

[3] R. A. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, Inc., 1997.

[4] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In M. Crochemore and D. Gusfield, editors, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, Asilomar, CA, 1994. Springer-Verlag, Berlin.

[5] Ricardo A. Baeza-Yates and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *String Processing and Information Retrieval*, pages 14–22, 1998.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] Jon Louis Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.

[8] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[9] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[10] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.

[11] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[12] Tolga Bozkaya and Z. Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 357–368. ACM Press, 1997.

[13] Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM TODS*, 24(3):361–404, 1999.

[14] Sergey Brin. Near neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 574–584. Morgan Kaufmann, 1995.

[15] E. Bugnion, S. Fhei, T. Roos, P. Widmayer, and F. Widmer. A spatial index for approximate multiple string matching. In R. A. Baeza-Yates and N. Ziviani, editors, *1st South American Workshop on String Processing (WSP'93)*, pages 43–53, 1993.

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

[17] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. In *SCCC 2001, Proceedings of the XXI Conference of the Chilean Computer Science Society*, pages 33–40. IEEE CS Press, 2001.

[18] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid Tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 440–447. IEEE Computer Society, 1999.

[19] Edgar Chávez, José L. Marroquín, and Ricardo A. Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *Proceedings of String Processing and Information Retrieval Symposium & International Workshop on Groupware (SPIRE/CRIWG 1999)*, pages 38–46, 1999.

[20] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, pages 57–64, 1999.

[21] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed Queries Array: A fast and economical data structure for proximity searching. *ACM Multimedia Tools and Applications*, 14(2):113–135, 2001.

[22] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys (CSUR)*, 33(3):273–321, 2001.

[23] Tzi-cker Chiueh. Content-based image indexing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 582–593. Morgan Kaufmann, 1994.

[24] Paolo Ciaccia and Marco Patella. Bulk loading the M-tree. In *Proceedings of th 9th Australasian Database Conference (ADC'98), Perth, Australia*, pages 15–26, 1998.

[25] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[26] Paolo Ciaccia, Marco Patella, and Pavel Zezula. A cost model for similarity queries in metric spaces. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 59–68. ACM Press, 1998.

[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. Processing complex similarity queries with distance-based access methods. In Hans-Jörg Schek, Fèlix Saltor, Isidro Ramos, and Gustavo Alonso, editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 1998.

[28] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *IEEE International Conference on Data Engineering (ICDE'02), San Jose, California, USA*, pages 41–52, 2002.

[29] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[30] Trevor F. Cox and Micheal A. Cox. *Multidimensional Scaling*. Chapman and Hall, 1994.

[31] Frank K. H. A. Dehne and Hartmut Noltemeier. Voronoi trees and clustering problems. *Information Systems (IS)*, 12(2):171–175, 1987.

[32] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Separable splits in metric data sets. In Augusto Celentano, Letizia Tanca, and Paolo Tiberio, editors, *Proceedings*

of 9-th Italian Symposium on Advanced Database Systems, SEBD 2001, Venezia, Italy, June 27-29, 2001, pages 45–62. LCM Selecta Group - Milano, 2001.

[33] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-Index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.

[34] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity join in metric spaces. In Fabrizio Sebastiani, editor, *Proceedings of 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003*, volume 2633 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2003.

[35] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. A metric index for approximate text management. In *Proceedings of the IASTED International Conference Information Systems and Databases, ISDB 2002, Tokyo, Japan, September 25-27, 2002*, pages 37–42. ACTA press, September 2002.

[36] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Access structures for advanced similarity search in metric spaces. In Sergio Flesca, Sergio Greco, Domenico Saccà, and Ester Zumpano, editors, *Proceedings of the Eleventh Italian Symposium on Advanced Database Systems, SEBD 2003, Cetraro (CS), Italy, June 24-27, 2003*, pages 483–494. Rubettino Editore, 2003.

[37] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity join in metric spaces using ed-index. In Vladimír Mařík, Werner Retschitzegger, and Olga Štěpánková, editors, *Proceedings of Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003*, pages 484–493. Springer – Lecture Notes in Computer Science, 2003.

[38] Christos Faloutsos, Ron Barber, Myron Flickner, Jim Hafner, Wayne Niblack, Dragutin Petkovic, and William Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems (JIIS)*, 3(3/4):231–262, 1994.

[39] Christos Faloutsos and King-Ip Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 163–174. ACM Press, 1995.

[40] William Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

[41] Volker Gaede and Oliver Gnther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[42] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 371–380. Morgan Kaufmann, 2001.

[43] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity search in metric databases through hashing. In *Proceedings of the 2001 ACM workshops on Multimedia*, pages 1–5. ACM Press, 2001.

[44] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 491–500. Morgan Kaufmann, 2001.

[45] Alfred Gray. *Modern Differential Geometry of Curves and Surfaces with Mathematica*. CRC Press; 2 edition, December 29, 1997. For citations of definition of METRIC.

[46] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 287–298. ACM, 2002.

[47] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts*, pages 47–57. ACM Press, 1984.

[48] James L. Hafner, Harpreet S. Sawhney, William Equitz, Myron Flickner, and Wayne Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 17(7):729–736, July 1995.

[49] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[50] Andreas Henrich. The LSD$^{\mathrm{h}}$-Tree: An access structure for feature vectors. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 362–369. IEEE Computer Society, 1998.

[51] Gísli R. Hjaltason and Hanan Samet. Incremental similarity search in multimedia databases.

[52] Daniel P. Huttenlocher, Gregory A. Klanderman, and William Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI'93)*, 15(9):850–863, 1993.

[53] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High performance metric trees

minimizing overlap between nodes. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2000.

[54] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering (TSE'83)*, 9(5):631–634, 1983.

[55] Kothuri Venkata Ravi Kanth, Divyakant Agrawal, and Ambuj K. Singh. Dimensionality reduction for similarity searching in dynamic databases. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 166–176. ACM Press, 1998.

[56] J. L. Kelly. *General Topology*. D. Van Nostrand, New York, 1955.

[57] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatio-Temporal Database Management*, pages 119–134, 1999.

[58] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 201–212, 2000.

[59] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, 1956.

[60] Karen Kukich. Technique for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[61] Dongwon Lee. *Query Relaxation for XML Model*. PhD thesis, University of California, Los Angeles, 2002. One chapter about metrics on XML documents (XML data trees).

[62] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[63] Luisa Micó, Jose Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17(7):731–739, 1996.

[64] María Luisa Micó, José Oncina, and Enrique Vidal. An algorithm for finding nearest neighbors in constant average time with a linear space complexity. In *Proceedings of the 11th International Conference on Pattern Recognition (ICPR'92)*, volume vol. II, pages 557–560, 1992.

[65] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, 1994.

[66] Francisco Moreno-Seco, Luisa Micó, and Jose Oncina. A modification of the LAESA algorithm for approximated k-NN classification. *Pattern Recognition Letters*, 24(1-3):47–53, 2003.

[67] Gonzalo Navarro. Searching in metric spaces by spatial approximation. In *Proceedings of String Processing and Information Retrieval (SPIRE'99), Cancun, Mexico*, pages 141–148, 1999.

[68] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, 2001.

[69] Gonzalo Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, 2002.

[70] Gonzalo Navarro and Nora Reyes. Fully dynamic spatial approximation trees. In Alberto H. F. Laender and Arlindo L. Oliveira, editors, *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, volume 2476 of *Lecture Notes in Computer Science*, pages 254–270. Springer, 2002.

[71] Hartmut Noltemeier. Voronoi trees and applications. In *International Workshop on Discrete Algorithms and Complexity*, pages 69–74, 1989.

[72] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. A data structure for representing and efficient querying large scenes of geometric objects: MB^* Trees. In *Geometric Modelling*, pages 211–226, 1992.

[73] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. Monotonous Bisector* Trees - a tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms*, pages 186–203, 1992.

[74] Juan Ramón Rico-Juan and Luisa Micó. Comparison of AESA and LAESA search algorithms using string and tree-edit-distances. *Pattern Recognition Letters*, 24(9-10):1417–1426, 2003.

[75] Enrique Vidal Ruiz. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–157, 1986.

[76] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[77] Bernhard Seeger, Per-Åke Larson, and Ron McFayden. Reading a set of disk pages. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *Procceding of 19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland*, pages 592–603. Morgan Kaufmann, 1993.

[78] Thomas Seidl and Hans-Peter Kriegel. Efficient user-adaptable similarity search in large multimedia databases. In *The VLDB Journal*, pages 506–515, 1997.

[79] Marvin Shapiro. The choice of reference points in best-match file searching. *Communications of the ACM*, 20(5):339–343, 1977.

[80] Tomáš Skopal, Jaroslav Pokorný, Michal Krátký, and Václav Snášel. Revisiting M-Tree building principles. In Leonid A. Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe

Wloka, editors, *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, volume 2798 of *Lecture Notes in Computer Science*. Springer, 2003.

[81] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[82] Ioana Stanoi, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Discovery of influence sets in frequently updated databases. In *The VLDB Journal*, pages 99–108, 2001.

[83] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[84] Enrique Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters*, 15(1):1–7, 1994.

[85] Juan Miguel Vilar. Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm. *Information Processing Letters*, 56(5):265–271, 1995.

[86] Congjun Yang and King-Ip Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, pages 485–492, 2001.

[87] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. ACM Press, 1993.

[88] Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *6th DIMACS Implementation Challenge, ALENEX'99, Baltimore, MD*, 1999.