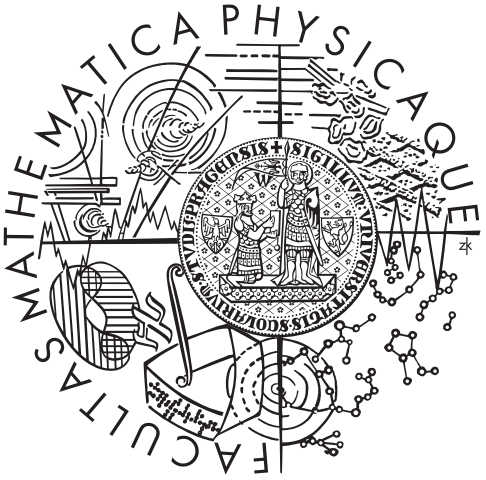
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Jakub Lokoč

Tree-based Indexing Methods for Similarity Search in Metric and Nonmetric Spaces

Department of Software Engineering
Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.

JUNE, 2010

Název: Stromové indexační metody pro podobnostní vyhledávání v metrických a nemetrických prostorech

Autor: Mgr. Jakub Lokoč
Katedra: Katedra softwarového inženýrství
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze
Školitel: Doc. RNDr. Tomáš Skopal, Ph.D.
Email autora: lokoc@ksi.mff.cuni.cz
Email školitele: skopal@ksi.mff.cuni.cz

Abstrakt: M-strom je dnes již klasická indexační metoda používaná pro efektivní podobnostní vyhledávání v metrických prostorech. Ačkoliv M-strom již nepatří mezi nejnovější metody, věříme, že stále nabízí zatím neobjevený potenciál. V této práci se proto zaměřujeme na způsoby, jak vylepšit jeho původní algoritmy a strukturu. Abychom umožnili rychlejší zpracování dotazů pomocí M-stromu, navrhli jsme několik nových metod jeho konstrukce (i paralelních), které vedou k vytváření kompaktnějších metrických hierarchií a přitom nejsou extrémně drahé. Dále jsme ukázali snadný způsob, jak rozšířit M-strom na novou indexační metodu NM-strom, která slouží k efektivnímu nemetrickému podobnostnímu vyhledávání za pomocí algoritmu TriGen. Všechna tato experimentálně ověřená vylepšení prokazují, že můžeme M-strom stále ještě považovat za důležitou dynamickou metrickou přístupovou metodu vhodnou pro správu rozsáhlých kolekcí nestrukturovaných dat. Všechna prezentovaná vylepšení mohou být navíc implementována do následníků M-stromu (např. do PM-stromu), což otevírá dveře pro další výzkum v této oblasti.

Klíčová slova: podobnostní vyhledávání, metrické přístupové metody, indexování, M-strom, TriGen

Title: Tree-based Indexing Methods for Similarity Search in Metric and Nonmetric Spaces

Author: Mgr. Jakub Lokoč
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.
Author's e-mail address: lokoc@ksi.mff.cuni.cz
Supervisor's e-mail address: skopal@ksi.mff.cuni.cz

Abstract: The M-tree is a well-known indexing method enabling efficient similarity search in metric spaces. Although the M-tree is an aging method nowadays, we believe it still offers an undiscovered potential. We present several approaches and directions that show how the original M-tree algorithms and structure can be improved. To allow more efficient query processing by the M-tree, we propose several new methods of (parallel) M-tree construction that achieve more compact M-tree hierarchies and preserve acceptable construction cost. We also demonstrate that the M-tree can be simply extended to a new indexing method – the NM-tree, which allows efficient nonmetric similarity search by use of the TriGen algorithm. All these experimentally verified improvements show that the M-tree can still be regarded as an important dynamic metric access method suitable for management of large collections of unstructured data. Moreover, all the improvements can be further adopted by M-tree descendants (e.g. the PM-tree), so that the results presented in this thesis open the door for future research in this area.

Keywords: similarity search, metric access methods, indexing, M-tree, TriGen

Contents

Preface **vii**
Searching by Similarity . . . viii
Summary of Contributions . . . ix
Structure of the Thesis . . . x
Acknowledgments . . . xii

1 The Fundamentals of Similarity Search in Metric Spaces **1**
1.1 Examples of Problem Domains . . . 1
1.2 Content-Based Similarity Search . . . 2
1.3 Similarity Queries . . . 4
1.3.1 Single-example Queries . . . 5
1.3.2 Multi-example Queries . . . 7
1.3.3 Motivation for Efficient Similarity Search . . . 8
1.4 The Metric Space Model . . . 8
1.4.1 Metric Distance Measures . . . 9
1.4.2 Pivots . . . 14
1.4.3 Partitioning Principles . . . 14
1.4.4 Principles of Filtering . . . 16
1.4.5 Indicators of Indexability . . . 19
1.5 Metric Access Methods . . . 20
1.5.1 Sequential File . . . 21
1.5.2 D-File . . . 21
1.5.3 Pivot Tables . . . 22
1.5.4 GNAT . . . 23
1.5.5 M-tree . . . 24
1.5.6 D-index . . . 25
1.5.7 M-index . . . 26
1.5.8 Other Methods . . . 28

1.5.9 M-tree versus other MAMs 29

I Revisiting M-tree Construction 31

2 M-tree 33
2.1 Similarity Queries in M-tree 34
2.2 Compactness of M-tree Hierarchy 35
2.3 Building the M-tree 36
2.3.1 Leaf Selection 36
2.3.2 Node Splitting 36
2.4 Related Work 37
2.4.1 Slim-down Algorithm 38
2.4.2 Multi-way Leaf Selection 39
2.4.3 Bulk Loading 39
2.5 PM-tree 40

3 Forced Reinsertions and Hybrid Way Leaf Selection 43
3.1 Forced Reinsertions 43
3.1.1 Full Reinsertions 45
3.1.2 Conservative Reinsertions 47
3.1.3 Construction vs. Query Efficiency 50
3.1.4 Average Leaf Node Utilization 51
3.2 Hybrid-way Leaf Selection 52
3.3 Experimental Evaluation 54
3.3.1 Databases 55
3.3.2 Experiment Settings 55
3.3.3 The Results 56
3.3.4 Summary 61
3.4 Discussion 62

4 Parallel Dynamic Batch Loading 65
4.1 Parallel Processing in MAMs 65
4.2 Parallel Opportunities in the M-tree Construction 66
4.2.1 Node Locking 67
4.2.2 Parallel Distance Matrix Evaluation 67
4.2.3 Parallel Promotion 68
4.3 Parallel Dynamic Batch Loading 68

4.3.1 Iteration Steps . . . 69
4.3.2 Scalability Notes . . . 71
4.4 Experimental Evaluation . . . 73
4.4.1 Databases . . . 74
4.4.2 M-tree Settings . . . 74
4.4.3 S Generating Heuristics . . . 74
4.4.4 Parallel Batch Loading Speedup . . . 75
4.4.5 Comparison with the Original Algorithm . . . 76
4.5 Discussion . . . 78

II Beyond the Metric Space Model 81

5 Nonmetric Similarity Search 83
5.1 Motivation for Nonmetric Search . . . 83
5.2 Similarity Search in Nonmetric Spaces . . . 87
5.3 Distance modifications . . . 88
5.3.1 Similarity-Preserving Modifiers . . . 89
5.3.2 Triangle-Generating Modifiers . . . 90
5.3.3 Triangle-Violating Modifiers . . . 93
5.4 Discussion . . . 96

6 TriGen Algorithm 97
6.1 T-error . . . 97
6.2 T-bases . . . 98
6.2.1 Indexability . . . 100
6.3 TriGen algorithm . . . 100
6.3.1 Experimental Results . . . 102
6.3.2 Discussion . . . 103

7 NM-tree 105
7.1 Indexing . . . 105
7.2 Query processing . . . 107
7.2.1 Exact search. . . . 107
7.2.2 Approximate search. . . . 107
7.3 Experimental Results . . . 109
7.3.1 Databases . . . 110
7.3.2 Querying . . . 111

7.4 Discussion 114

8 Conclusions **115**
8.1 Outlook 117

Preface

During the last two decades, devices for capturing digital data have become cheap and widely available. In connection with the practically "unlimited" available storage capacity, the popularity of such devices led to an exponential growth of the volume of produced digital data. For example, the International Data Corporation predicts over a thousand exabytes of digital data will be generated in 2010 [Gantz, 2008]. Moreover, there is no doubt the trend will continue and that more complex devices will soon infiltrate all domains of human activities. Even now, there are a lot of devices owned by nearly every person that produce diverse digital data. For example, very popular "all-in-one" devices, like smart-phones, can capture and record photos, videos, sounds, trip paths consisting of GPS coordinates, and allow to publish media immediately on the Internet. Besides devices usually used for entertainment, there emerge many machines, analyzers and meters in all spheres of industry, business, health service, science, etc., producing enormous collections of unstructured digital data[1] stored in specific data repositories. The most requested ability of such repositories is to allow efficient data management, analysis and searching. Unfortunately, digital data like photos, videos, sounds, time series, are usually large and complex objects with very poor or completely missing content structure. Here, the classical data management approaches, like relational databases or text-based retrieval systems, cannot be employed, since they are primarily designed for well-structured or textual data. Hence, some other retrieval models have to be employed for unstructured data.

One possible retrieval model is to equip unstructured data by keyword annotation, and thus create a bridge between unstructured data and text-based retrieval systems [Baeza-Yates and Ribeiro-Neto, 1999], [Aggarwal and Yu,

[1] For example, the Large Hadron Collider at CERN generates 40 terabytes of data every second [CERN, 2010].

2000]. For example, medicine doctors usually add keywords to EEG curves or x-ray pictures, to highlight some important anomalies used for diagnostics. However, the annotation approach has several weak points. It is possible to annotate just a small fraction of the captured digital data, because it is usually highly qualified work and the number of domain experts is limited. The annotation is also usually very subjective, so that the search results cannot satisfy all users. Hence, a highly qualified human annotation is restricted to domain-specific catalogs with a limited number of records. An automatic machine-based annotation of multimedia content used by Internet giants like Google or Yahoo, is a different concept of keywords assignment (determined from URL and the embedding web page). However, an automatic annotation is very imprecise in guessing the semantics from the surrounding text and thus it cannot be always used for effective multimedia search.

Searching by Similarity

The content-based retrieval [Deb, 2004], [Blanken et al., 2007], [Datta et al., 2008], where data object itself serves as a query (the query-by example paradigm), is a more suitable approach to the retrieval of raw unstructured data. It also happens that objects we are searching for are not present in the requested database, so that exact matching makes no sense. The similarity search concept is a more appropriate search paradigm for content-based retrieval systems.

As for annotation, the domain experts are essential also for the similarity search, however, they are necessary only to specify the model space (i.e., defining data descriptors and the similarity function working with them). This concept is more profitable, since a particular feature extraction and a similarity function are reusable algorithms that can be employed for an arbitrary number of processed objects. Thus, the experts can primarily focus on modeling of complex similarity functions that satisfy user expectations.

In this work, we have focused on the metric space approach, since it is a model general enough to be applicable for various kinds of data, yet specific enough to be useful for efficient (fast) retrieval. In the metric model, objects can be represented by black-box feature descriptors, for which a similarity measure (actually, dissimilarity, or distance) satisfying metric axioms is defined.

The usage of the general metric space model may also lead to a perfor-

mance loss in case there is a domain-specific database solution designed for a specific application. However, there is a strong argument justifying the general metric space model – solving a problem in the metric model is more reusable for other data types. Moreover, the metric model is often the only available solution for complex and unstructured data. Hence, we rather focus on finding general algorithms in metric spaces, which can be later adopted in various domains employing content-based retrieval, classification, machine learning, pattern recognition, statistical analysis, data mining, where similarity between objects is considered.

We also address the nonmetric similarity search, which becomes very requested new approach, since metric postulates often constitute an obstacle for domain experts designing a similarity measure. On the other hand, getting rid of the metric postulates brings problems to the database experts. Especially the triangle inequality, which is the most criticized postulate, is the crucial property for efficient database indexing. If we break this strong postulate, then all metric access methods become only approximate methods. Nevertheless, approximate searching can be in some cases very efficient and effective enough. Moreover, since the amount of triangle inequality can be tuned by a domain expert, as was recently proposed in [Skopal, 2006] and [Skopal, 2007], we can employ nonmetric searching as a scheme for approximate search. Therefore, we have also focused on methods allowing tunable approximate similarity searching (both metric and nonmetric) using some well-known metric access methods.

Summary of Contributions

In this thesis we focus on M-tree, that is a hierarchical, balanced and paged indexing structure suitable for efficient similarity search in very large metric databases. We present three new construction methods in Part I and an extension of M-tree for nonmetric spaces, called NM-tree, in Part II. The contribution of Chapter 7 is a follow-up of the TriGen algorithm, a previously proposed approach to efficient nonmetric similarity search. All these contributions are summarized as follows:

- We propose a new method of M-tree construction (published in [Lokoč and Skopal, 2008] and [Skopal and Lokoč, 2009]) based on the well-known principle of forced reinsertions. This method improves the qual-

ity of an M-tree hierarchy and thus higher search efficiency is achieved. Moreover, we also use forced reinsertions as a tool guaranteeing a user-defined average leaf node utilization.

- We have also revisited the leaf selection strategies in the M-tree and proposed new hybrid-way leaf selection strategy (published in [Skopal and Lokoč, 2009]). This tunable method enables a user to choose a trade-off between indexing costs and query efficiency. We also proposed the multi-way leaf selection strategy in connection with forced reinsertions, that outperforms the other state-of-the-art M-tree construction techniques both in construction and query costs.

- As a contribution to fast M-tree construction, we propose the parallel batch loading method (published in [Lokoč, 2009]). This method employs simultaneous insertions and guarantees a significant speedup. To avoid synchronization problems, we utilize postponed reinsertions, which results in the construction of more compact M-tree hierarchies.

- In the last contribution we present a new indexing structure called NM-tree (published in [Skopal and Lokoč, 2008]) that combines the M-tree and the TriGen algorithm [Skopal, 2006]. The NM-tree natively supports both metric and nonmetric search without re-indexing. This is a very important feature, because user can decide at the query time, whether to perform approximate (faster) or exact (slower) search.

Structure of the Thesis

Apart from the first and the last chapter (introduction and conclusion), the thesis is organized into two parts. In the first part we focus on the efficient metric similarity search by M-tree, while in the second part we address the nonmetric search problems and introduce our new nonmetric access method, the NM-tree. For better transparency, description of individual chapters follows.

Chapter 1 remembers the fundamental principles of the content-based similarity search in the metric space model. We describe the motivation and the main approaches for efficient indexing and mention several state-of-the-art metric access methods.

Since our work is mainly based on the M-tree structure, we remember this metric access method in detail in Chaper 2. We also describe the related work dealing with various methods of M-tree construction, because the following chapters contribute to this topic.

In Chapter 3, we introduce the forced reinsertions and the hybrid-way leaf selection strategy as the new approaches for relatively cheap and efficient dynamic M-tree construction. In series of experiments, we present that these two techniques can improve the compactness of the M-tree hierarchy, while the indexing costs can still remain acceptable.

Since the parallel processing has become an important trend in algorithm design, we also introduce our new parallel dynamic batch loading method of M-tree construction in Chapter 4. The method provides significant speedup as shown in experiments.

We also address the nonmetric similarity search which is discussed in Chapter 5. Since we follow the approach utilizing dissimilarity modifications and reusing metric access methods for nonmetric search, we also show how to simply turn any nonmetric to a semimetric. Further, we define and prove several important formal rules necessary for adding the triangle inequality into any semimetric, making it metric.

In Chapter 6, we mention the TriGen algorithm, which is a tool for finding optimal dissimilarity modifiers. We also remember the T-error model expressing the behavior of metric access methods under a semimetric measure.

Our contributions to the M-tree are not restricted just to the metric model. In Chapter 7, we introduce a new M-tree based nonmetric access method, the NM-tree, utilizing the TriGen algorithm. The NM-tree allows both exact and approximate search in nonmetric spaces, while its performance remains similar as that for the M-tree.

Chapter 8 concludes the thesis and gives an outlook for the future research, based on the ideas and methods presented in this thesis.

Acknowledgments

I would like to thank all those who supported me in my doctoral studies and work on my thesis. In the first place I very appreciate the help and advice received from my supervisor Tomáš Skopal and I am grateful for numerous corrections and comments. Secondly, I would like to thank to my family for their enormous patience, love and support. And undoubtedly, I must also express my thanks to all the anonymous reviewers of my papers for helpful remarks and ideas.

My thanks also go to institutions that provided financial support for my research work. During my doctoral studies, my work was partially supported by the following research projects:

- Czech Science Foundation (GAČR), grant number 201/09/0683
- Grant Agency of Charles University (GAUK), grant number 18208
- National Programme of Research, Information Society Project number 1ET100300419
- Ministry of Education of the Czech Republic (grant MSM0021620838)
- Specific research program SVV-2010-261312

Jakub Lokoč

June, 2010

Chapter 1

The Fundamentals of Similarity Search in Metric Spaces

During the last two decades, similarity searching in metric spaces has become intensively investigated research area, as documented in several excellent monographs and surveys [Chávez et al., 2001], [Zezula et al., 2005], [Samet, 2006]. In the next paragraphs, we will shortly summarize basic fields of the content-based similarity searching using metric space model. For more details, see one of the referred publications.

1.1 Examples of Problem Domains

The most important goal of any approach is its applicability. Hence, we start with several examples of problem domains, where the content-based similarity search is the mostly requested search operation.

- *Multimedia* – images, videos, sounds, XML documents, are the most expanding data collections [Gantz, 2008]. Such an expansion is caused mainly by the enormous popularity of the Internet and social networks like Facebook, where it is really very easy to upload and share multimedia content. Searching this flood of multimedia data becomes harder and harder, which brings opportunity for undesirable phenomena like plagiarism or identity theft. Retrieval systems, supporting queries like "Return all publications similar to these papers." or "Return all photos with somebody's face published in the Internet.", can help to fight these problems. The popularity of the multimedia retrieval (especially the

image retrieval) is confirmed by the growth of the number of published articles in this area [Datta et al., 2008].

- *Biometric identification* – a tool for solving immediate identity problems, where unique biometric characteristics like fingerprint, face shape, iris, voice, are used for the person identification [Brunelli and Poggio, 1993], [Moghaddam and Pentland, 1999]. Here, the similarity search is a way how to identify a person, since it cannot be guaranteed that the actually scanned biometric characteristics of a person are exactly the same as the characteristics stored in the database of persons.

- *Spatial models* – the problem of 2D and 3D shape matching [Huttenlocher et al., 1993], [Bartolini et al., 2005], [Bustos et al., 2005], [Keogh et al., 2009] arises in various domains including CAD/CAM systems, virtual reality, molecular biology, GIS systems. An efficient retrieval of the most similar shapes is crucial for the performance of such systems.

- *Time series* – the similarity search in areas like seismology [Angeles-Yreta et al., 2004], stock exchange [Fu et al., 2007], medicine (EEG, ECG), voice recognition, works with large sequences of numbers and massive databases [Shieh and Keogh, 2009]. Finding similar subsequences [Faloutsos et al., 1994] or trends in time series is very important task in various predictive techniques like earthquake detection, stock behavior extrapolation, network load prediction, future sales forecasting, to name a few.

1.2 Content-Based Similarity Search

One of the key problems occurring in the content-based similarity retrieval is to understand and describe what people consider as similar in a specific domain. The psychological background of similarity is concerned in many publications like [Tversky, 1977], [Ashby and Perrin, 1988], [Ashby, 1992], [Santini and Jain, 1999]. However, it is impossible to write a universal algorithm that will excellently supply human perception of similarity. The reason is that the process of similarity evaluation, which happens in our brain is often too complex and individual. Nevertheless, it is possible to computerize this process for the price of some acceptable error rate. Similarity is usually

modeled by a similarity function σ, that accepts a pair of objects and returns a real number, where the higher number means the higher similarity.

However, it is impossible to design a similarity measure σ for very complex and unstructured objects from a database collection $\mathcal{C}$. Hence, a simplified feature space $\mathbb{U}$, consisting only from significant extracted features, is derived from the original space $\mathcal{C}$ by an extraction function $e : \mathcal{C} \mapsto \mathbb{U}$.

Definition 1. *Let $\mathbb{U}$ be a feature space derived from $\mathcal{C}$ and σ be a total function defined as $\mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}$, then σ is called the pairwise similarity function and the couple ($\mathbb{U}$, σ) is called the similarity space.*

The similarity measure is often modeled by a dissimilarity (or distance) measure δ, which adopts an inverse view of similarity, that is, a higher distance stands for a lower similarity score, and vice versa. A trivial conversion between bounded similarity and dissimilarity spaces can be realized by formula $\delta = \sigma_{max} - \sigma$.

Definition 2. *Let $\mathbb{U}$ be a feature space derived from $\mathcal{C}$ and δ be a total function defined as $\mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}$, then δ is called the pairwise distance function and the couple ($\mathbb{U}$, δ) is called the distance space.*

For example, in the content-based image retrieval, the original collection $\mathcal{C}$ of images is usually transformed to the feature space $\mathbb{U}$ consisting of color and edge histograms, shapes, etc., extracted from the original images. The *Euclidean* or the *Hausdorff* distance (for definition see Section 1.4.1) is then often applied to compare two objects from $\mathbb{U}$.

Since a feature extraction should capture all important properties of the modeled objects, which may result in a nontrivial (non-vectorial) data representation, a very general mathematical model is needed, to formalize and prove basic concepts of similarity searching. Metric spaces [Kelley, 1975] proved to be a very suitable model, because it puts minimal constraints both on the data representation and the distance function. On the other side, the metric postulates still allow efficient indexing and query processing, which is the essential condition for a successful management of very large databases.

Preferences of Similarity Search

A suitability of the employed retrieval model is usually measured by its *effectiveness* and *efficiency*, where the effectiveness means the quality of the query results and the efficiency means how quick the query processing is. The

effectiveness in the content-based similarity search is tightly bound to the employed similarity measure. Usually, the more effective search is required, the more complex and expensive similarity measure has to be employed. We distinguish two orthogonal preferences of the similarity search as follows:

- *Topological preferences.* The complex similarity measures usually relax some important topological properties and are modeled by nonmetrics. On the other hand, topological properties like metric postulates are necessary for indexing and efficient retrieval. Hence, we have to choose whether to prefer metric or nonmetric measures.

- *Precision preferences.* Although in some applications the precision is the essential property of the employed similarity measure (e.g., biometric identification), in most areas the similarity measuring and retrieval is inherently imprecise, subjective and changing over time (e.g., image retrieval). Hence, a user may prefer faster but approximate search, where an acceptable number of false hits and false dismissals may appear.

In this thesis, we focus both on the exact and approximate similarity search, both metric and nonmetric. In Part I, we address methods allowing more efficient exact similarity search, while in Part II, we propose a new access method allowing tunable approximate similarity search.

1.3 Similarity Queries

The task of similarity search in an unstructured database $\mathbb{S} \subseteq \mathbb{U}$ is usually accomplished by "single-example" queries, where only one query object $q \in \mathbb{U}$ and a proximity criterion is used to describe the expected result set $\mathbb{X}$. The expressive power of these "single-example" queries may fail in cases, when the user's delicate query intent is not available as a single example. Thus, there also emerged several new query types, so-called "multi-example" queries, considering more objects forming a query. In the next two Sections, we will recall some basic "single-example" and "multi-example" query types used in the content-based retrieval.

1.3.1 Single-example Queries

The portfolio of available similarity query types consists of mostly single-example queries, where only one query object is considered. As a result, such objects are returned, which satisfy a proximity condition specified by the query object and the query type.

Range Query

The similarity range query $R(q, r_q)$ is the simplest type of similarity queries. The query parameters are a query object q (need not exist in the database) and a user-defined query radius r_q constituting a query ball (see Figure 1.1a).

Definition 3. *Let* $q \in \mathbb{U}$, $\mathbb{S} \subseteq \mathbb{U}$, $r_q \in \mathbb{R}_0^+$ *and* $(\mathbb{U}, \delta)$ *be a distance space, then* $R(q, r_q) = \{\mathbb{X} \subseteq \mathbb{S}, \forall x \in \mathbb{X} : \delta(q, x) \leqslant r_q\}$

Objects in the result are usually ranked according to their similarity to the query object. The special case of the range query, where the radius is set to zero $R(q, 0)$, is called the point query or the exact match query.

k Nearest Neighbor Query

The most noticeable problem of the range query is the necessity of a query radius. A setting of too low or too large radius may result in an undesired result set size. In most cases, a user requests just the k most similar objects, thus here the k nearest neighbor queries $kNN(q)$ are more suitable (see Figure 1.1b).

Definition 4. *Let* $q \in \mathbb{U}$, $\mathbb{S} \subseteq \mathbb{U}$, $k \in \mathbb{N}$, $k > 0$ *and* $(\mathbb{U}, \delta)$ *be a distance space, then* $kNN(q) = \{\mathbb{X} \subseteq \mathbb{S}, |\mathbb{X}| = k \wedge \forall x \in \mathbb{X}, \forall y \in \mathbb{S} - \mathbb{X} : \delta(q, x) \leqslant \delta(q, y)\}$

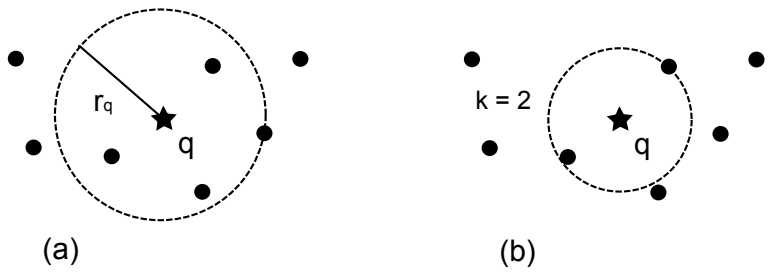


Figure 1.1: (a) Range query ball with radius r_q (b) 2NN query

The shape of the kNN query is (again) the range query ball with radius equal to the distance to the k-th nearest neighbor of the query object. However, the radius is initially unknown and thus heuristics for its fast estimation are employed when searching. The radius is usually set to ∞ and decreased using the nearest neighbors candidates array and the priority queue.

k Farthest Neighbor Query

The k farthest neighbor query is the opposite to the k nearest neighbor query, that is, the query returns the k most distant objects to the query object.

Definition 5. *Let* $q \in \mathbb{U}$, $\mathbb{S} \subseteq \mathbb{U}$, $k \in \mathbb{N}$, $k > 0$ *and* $(\mathbb{U}, \delta)$ *be a distance space, then* $kFN(q) = \{\mathbb{X} \subseteq \mathbb{S}, |\mathbb{X}| = k \wedge \forall x \in \mathbb{X}, \forall y \in \mathbb{S} - \mathbb{X} : \delta(q, x) \geqslant \delta(q, y)\}$

k Reverse Nearest Neighbor Query

The k reverse nearest neighbor query $kRNN(q)$ [Korn and Muthukrishnan, 2000] detects how a query object is perceived by other objects in the database. In other words, it selects objects that view the query object q as their near neighbor (see Figure 1.2a).

Definition 6. *Let* $q \in \mathbb{U}$, $\mathbb{S} \subseteq \mathbb{U}$, $k \in \mathbb{N}$, $k \geqslant 0$ *and* $(\mathbb{U}, \delta)$ *be a distance space, then* $kRNN(q) = \{\mathbb{X} \subseteq \mathbb{S}, \forall x \in \mathbb{X}, q \in kNN(x) \wedge \forall y \in \mathbb{S} - \mathbb{X} : q \notin kNN(y)\}$

k Distinct Nearest Neighbor Query

The k distinct nearest neighbor query kDNN [Skopal et al., 2009], which excludes all objects that are too similar to any of the already reported ob-

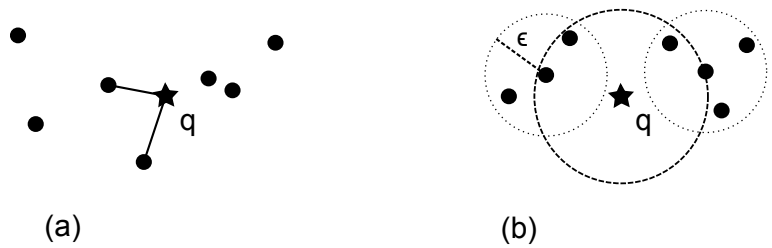


Figure 1.2: (a) 1RNN query (b) 2DNN query (returned objects are the centers of two smaller balls)

jects, have become a new query type for very large databases, filtering near-duplicates from the result set (see Figure 1.2b).

Definition 7. *Let* $q \in \mathbb{U}$*,* δ *be a distance function,* $\epsilon \in \mathbb{R}_0^+$*,* $k \in \mathbb{N}$*,* $k \geqslant 0$ *and* $(\mathbb{U}, \delta)$ *be a distance space, then* $kDNN(q, \epsilon) = \{\mathbb{X} \subseteq \mathbb{S}, |\mathbb{X}| = k \wedge \forall x, z \in \mathbb{X}, y \in \mathbb{S} - \mathbb{X} : \delta(x, z) > \epsilon \wedge \delta(q, x) \leqslant \delta(q, y) \vee \exists w \in \mathbb{X} : \delta(y, w) < \epsilon\}$

1.3.2 Multi-example Queries

Although the single-example queries are frequently used nowadays, their expressive power may become unsatisfactory in the future due to increasing complexity and quantity of available data. The acquirement of an example query object is the user's "ad-hoc" responsibility. However, when just a single query example should represent the user's delicate intent on the subject of retrieval, finding an appropriate example could be a hard task. Such a scenario is likely to occur when a large data collection is available, and so the query specification has to be fine-grained. Hence, instead of querying by a single example, an easier way for the user could be a specification of several query examples which jointly describe the query intent. Such a multi-example approach allows the user to set the number of query examples and to weigh the contribution of individual examples. Moreover, obtaining multiple examples, where each example corresponds to a partial query intent, is much easier task than finding a single "holy-grail" example.

The practical class of retrieval techniques, where multi-example queries have been successfully implanted, is the content-based image retrieval. There are several studies considering query effectiveness using multiple query images, for more details see [Tahaghoghi et al., 2001] and [Tahaghoghi et al., 2002]. Other techniques, using the relevance feedback to improve the quality of the result (using positive and negative examples), can be found in [Porkaew et al., 1999]. Very recently, a novel probabilistic framework to process multiple sample queries in the content-based image retrieval has been introduced in [Arevalillo-Herráez et al., 2010].

As for existing solutions to multi-example query types, there are three main directions. First, there exist many model-specific techniques based on an aggregation or unification of the multiple examples, e.g., querying by a centroid in case of vectors, or by a union, intersection or other composition of features of the query examples [Tang and Acton, 2003].

Second, a popular approach to multi-query example is issuing multiple

single-example queries, while the resulting multiple ranked lists are aggregated by means of a top-k operator [Fagin, 1999]. The advantage of this approach is the employment of an arbitrary aggregation function which provides an important add-on to the expressive power of querying. Similarity joins [Jacox and Samet, 2008], joining pairs of objects (from one or more databases) based on their proximity, can be also included to this group, although they can be regarded as an operator consisting of series of single-example queries, rather than a regular multi-example query type.

Third, there are also complex approaches, where all the query objects are necessary during each step of query processing. For example, the algorithms for efficient processing of the skyline operator in vector [Papadias et al., 2003] and metric spaces fall into this category [Chen and Lian, 2009], [Skopal and Lokoč, 2010].

1.3.3 Motivation for Efficient Similarity Search

All the presented similarity queries are generally formulated, and can be employed in an arbitrary distance space. The examples of such spaces could be – 2D points (representing coordinates of gas stations, hotels, etc.) measured by the Euclidean metric, collection of biometric descriptors measured by the Hausdorff distance, collection of strings measured by the Edit distance or collection of image color histograms measured by the Quadratic form distance[1]. The last three examples show that the employed dissimilarity measure could be computationally expensive ($\geqslant O(n^2)$, n being the descriptor size), hence, the sequential processing is often unacceptable for larger collections. Therefore, it is necessary to utilize a model providing rules for safe discarding of irrelevant objects (or sets of objects). In the following section, we will remember the metric space model, that is often employed for the similarity search over unstructured raw data.

1.4 The Metric Space Model

In this section, we will describe some fundamentals of metric spaces – the definition of metric space and the way it can be utilized for indexing and efficient searching. We will also present several frequently used metric distance functions.

[1] All the mentioned distances are described later in the Section 1.4.1.

Definition 8. *The metric space is a pair* $\mathbb{M} = (\mathbb{U}, \delta)$*, where* $(\mathbb{U}, \delta)$ *is a distance space and* δ *satisfies the following conditions for all objects* $x, y, z \in \mathbb{U}$*:*

$$\begin{array}{rcll} \delta(x,y) & = & 0 & \Leftrightarrow x = y \quad \textit{identity} \\ \delta(x,y) & \geqslant & 0 & \textit{non-negativity} \\ \delta(x,y) & = & \delta(y,x) & \textit{symmetry} \\ \delta(x,y) + \delta(y,z) & \geqslant & \delta(x,z) & \textit{triangle inequality} \end{array}$$

The distance function δ *is also referred to as the metric distance function.*

By removing or restricting some of the postulates, we get particular nonmetric spaces (for more details see Part II). Several examples of nonmetric spaces are summarized as follows:

- The *pseudometric space* – by restricting the identity property to the weaker reflexivity property ($\forall x : \delta(x, x) = 0$).
- The *quasimetric space* – by removing the symmetry property.
- The *semimetric space* – by removing the triangle inequality property.

For more variants and properties of metric spaces see [Watson, 1999] or [Zezula et al., 2005].

1.4.1 Metric Distance Measures

Distance functions are used to quantify the closeness of objects in various domains. It is up to domain experts to choose (and eventually parameterize) a suitable distance function that will mostly correspond to user expectations. In the following, we will recall several popular metric distances used for various kind of data represented in vector or metric spaces. The examples of several popular metric distances are depicted in Figure 1.3a-g. An example of a metric defined by the combination of two metrics is depicted in Figure 1.3h. The border region balls show the objects in a given fixed distance from q.

Minkowski Metrics

The Minkowski metrics (or L_p metrics) are the most popular dissimilarity measures used in various applications. However, the metrics are restricted just to vector spaces, where a distance between two vectors (points) is computed.

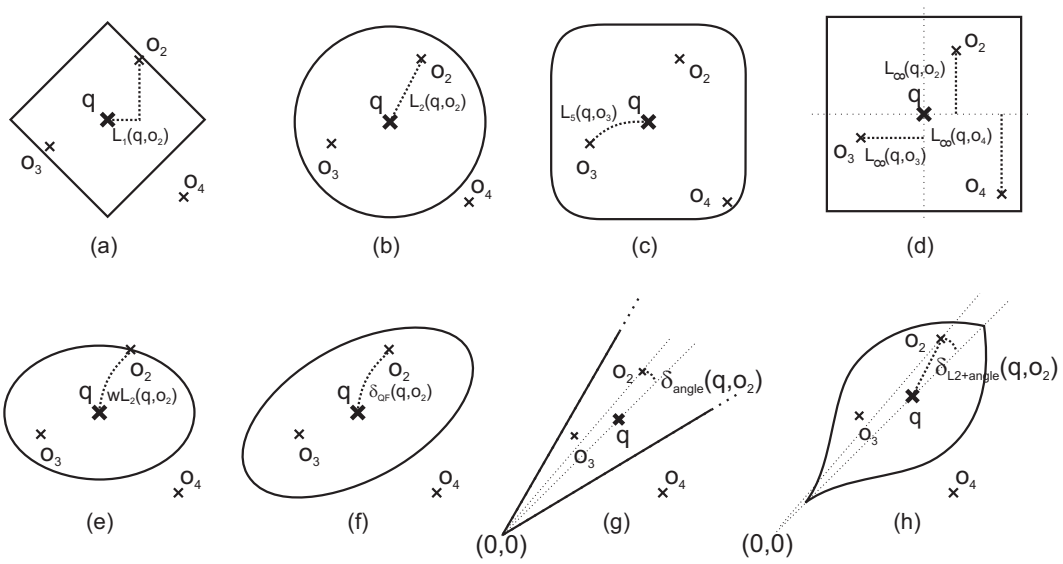


Figure 1.3: Region balls of various metric distances

Definition 9. *Let $\mathbb{V}$ be an n-dimensional vector space and $x, y \in \mathbb{V}$, then an L_p metric is defined as:*

$$L_p(x, y) = (\sum_{i=1}^{n} |x[i] - y[i]|^p)^{\frac{1}{p}} \qquad (p \geqslant 1)$$

In the Figure 1.3a-d, there are several examples of L_p metrics depicted, among them the L_1 distance (see Figure 1.3a) known as the *Manhattan distance*, the L_2 metric (see Figure 1.3b) well-known as the *Euclidean distance* and the L_∞ (see Figure 1.3d) called the *Chessboard distance*. The condition $p \geqslant 1$ is important, otherwise the distance would not be a metric – for $p < 1$ it does not satisfy the triangle inequality (see Figure 1.4). The time complexity of the distance evaluation is $O(n)$, hence L_p metrics are considered as cheap dissimilarity measures. The L_p metrics are suitable to model a dissimilarity in vector spaces with independent dimensions.

There exist also cases, where it is profitable to prefer more significant coordinates and to suppress the less significant ones. For such cases, the weighted L_p metric can be used as a generalized variant of the original L_p metric (see Figure 1.3e).

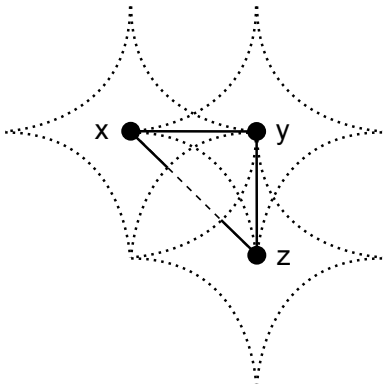


Figure 1.4: Nonmetric $L_{0.5}$ space, where $\delta(x, y) + \delta(y, z) < \delta(x, z)$

Quadratic Form Distance

The quadratic form distance [Seidl and Kriegel, 1997], also called the Mahalanobis distance, is a generalization of the weighted Euclidean metric. Quadratic form distance uses a geometric transformation matrix M, which expresses a correlation between vector coordinates, and thus the distance could better fit user expectations (see Figure 1.3f). Its typical application is histogram similarity, etc.

Definition 10. *Let $\mathbb{V}$ be an n dimensional vector space, $x, y \in \mathbb{V}$ and M be an $n \times n$ square positive semi-definite matrix, then the Quadratic form distance is defined as:*

$$L_{QF}(x, y) = \sqrt{(x - y)M(x - y)^T}$$

Angle Distance

The angle between two vectors (see Figure 1.3g) is also a metric that can be viewed as an L_2-distance along the surface of origin-centered unitary L_2-ball. The angle distance, in a form of non-metric cosine measure (see section 5.1), is widely used in Information Retrieval [Baeza-Yates and Ribeiro-Neto, 1999].

Levenshtein Metric

Levenshtein metric [Levenshtein, 1965], also called the edit distance, measures the proximity of two strings x, y, which can be expressed as the minimal

number of three basic edit operations (insert, delete and replace a character) used to transform string x into string y. An efficient evaluation of the edit distance is accomplished by the dynamic programming techniques. However, for two strings of length n and m, the time complexity of the distance evaluation is still high – $O(nm)$. Thus, for two long strings the edit distance becomes an expensive operation.

Tree Edit Distance

The similarity of two linear structures can be computed by the edit distance. For tree structures, there was a more sophisticated (and also more expensive) method developed – the tree edit distance. The tree edit distance is used for tree pattern matching [Zhang and Shasha, 1997] and is defined as the minimal number of tree edit operations like insertion or deletion of a node. The time complexity of the tree edit distance has a worst-case time complexity of $O(n^4)$, where n is the number of tree nodes.

Jaccard's Distance

In some applications, a data object is formed by a set. For example, a user behavior in the Internet can be represented as the set of visited web pages. To compare the behavior of two users, sets containing visited URLs are compared. A proximity of the two sets is usually measured by the Jaccard's distance, that measures a normed overlap distance between two sets.

Definition 11. *Assuming two sets X and Y, the Jaccard's distance is defined as:*

$$\delta_{JD}(X, Y) = 1 - \frac{|X \bigcap Y|}{|X \bigcup Y|}$$

Hausdorff Distance

The Hausdorff distance [Huttenlocher et al., 1993] is another way how to measure the proximity of two sets. The Hausdorff distance does not use just match/mismatch between elements of two sets, it employs an arbitrary metric subdistance δ' to compute nearest neighbors between all elements of the sets. This approach is usually applied for matching two sets consisting of geometric shapes, e.g., to compare two fingerprints represented by 2D points. The computation of the Hausdorff distance has a time complexity

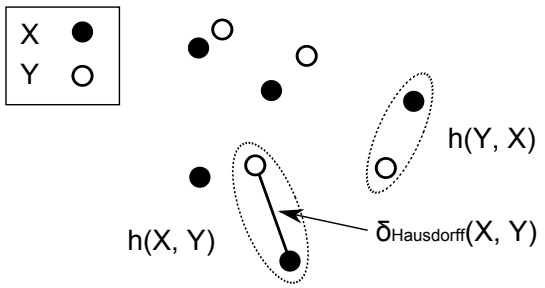


Figure 1.5: The Hausdorff distance between two sets of 2D points.

$O(nm)O(\delta')$ for sets of sizes n and m. An example of the Hausdorff distance, used to measure two sets of 2D points by L_2 metric, is depicted in the Figure 1.5.

Definition 12. *Assuming two sets X, Y, and a metric function δ', the Hausdorff distance is defined as:*

$$\delta_{Hausdorff}(X,Y) = \max\{h(X,Y), h(Y,X)\}$$

$$h(X,Y) = \max_{i=1...|X|}\{\min_{j=1...|Y|}\{\delta'(x_i, y_j)\}\}$$

where x_i means the i-th element of the set X.

Earth Mover's Distance

The Earth mover's distance (EMD) [Rubner et al., 1998] is used in content-based image retrieval as a metric between two distributions. The distance is based on the minimal cost, that must be paid to transform one distribution into the other, that is, to solve the *transportation problem*. A theoretical analysis of the time complexity of the transportation problem shows that the problem depends exponentially on the input size (in worst case). However, in practice, a good initial solution is available which drastically decrease the number of iterations needed, thus for smaller instances of the problem it is possible to evaluate the distance efficiently. Moreover, if both feature vectors have the same number of dimensions, the EMD can be computed in $O(n^3 \log n)$ time [Rubner and Tomasi, 2001].

1.4.2 Pivots

As mentioned in Section 1.3.3, sequential search of a database is often unacceptable and thus it is necessary to construct an index. Since the only "exploitable" information in a metric space are the values produced by the distance function δ, the employed indexing methods can utilize only distances between objects. Basically, all the metric indexing methods utilize a subset $\mathbb{P} \subset \mathbb{S}$ of specially selected objects, so-called pivots p_i (also referred to as vantage points or reference points). A relationship (e.g., in the form of precomputed distances) between pivots and the remaining database objects is later used for pruning and filtering during the search.

For example, one of the most fundamental rule employed in the metric search is the lower/upper bound distance estimation. Let $p, q, x \in \mathbb{S}$ and $d_1 = \delta(p, x)$, $d_2 = \delta(p, q)$, then from the triangle inequality we can estimate the lower/upper bound of distance $\delta(x, q) \in \langle |d_1 - d_2|, d_1 + d_2 \rangle$. These estimated distances can be employed during the search for safe filtering of irrelevant objects (described in the following Sections).

From the previous example, it follows that we need a set of pivots which will provide a good estimation of unknown distances. If all the database objects were used as pivots, we could obtain excellent distance estimations. However, the number of pivots should be as low as possible to reduce the index size and initial query costs[2]. Hence, to propose good and "cheap" distance estimations, a proper minimal set of pivots has to be selected. During the last decade, there arose several pivot selection techniques, considering the quality of the selected set of pivots. For more details, see [Bustos et al., 2003] or [Bustos et al., 2008].

1.4.3 Partitioning Principles

Partitioning is one of the most fundamental principles of any data access method. Objects from the database $\mathbb{S} \subseteq \mathbb{U}$ are divided into sub-groups according to their proximity, so that for a particular query, only several sub-groups have to be searched. Partitioning in metric spaces is more complicated, because there does not exist geometrical borders as in (Euclidean) vector spaces. The specially selected pivots p_i are used to break the database $\mathbb{S}$ into subsets containing objects similar by close pivots. Later, during the

[2]The distances between the query object and all the pivots are usually evaluated.

search, the pivots are used as the representatives of classes containing similar objects, and may be utilized in filtering rules. We distinguish two basic partitioning principles and several extensions combining them.

Ball partitioning

Ball partitioning [Uhlmann, 1991] uses only one pivot p to break the space $\mathbb{S}$ into two subgroups S_1 and S_2. Let r be a user-defined radius of a ball centered in p, then S_1 and S_2 are defined as:

- $S_1 = \{o_i | d(o_i, p) \leqslant r\}$
- $S_2 = \{o_i | d(o_i, p) \geqslant r\}$

To obtain a balanced partitioning, we can set r to distance d_m equal to the median from all the distances $d(o_i, p)$. If there are more objects with the median distance from p, the objects are distributed arbitrarily between S_1 and S_2, preserving the balanced fashion. An example of 2D vector space, divided into two parts by the ball partitioning, is depicted in the Figure 1.6a.

Generalized hyperplane partitioning

Generalized hyperplane partitioning also breaks the space $\mathbb{S}$ into two subgroups S_1 and S_2. In contrast to the ball partitioning, it uses two pivots p_1 and p_2 to divide objects into two subgroups according to their distance from the pivots. The sets S_1 and S_2 are defined as follows:

- $S_1 = \{o_i | d(p_1, o_i) \leqslant d(p_2, o_i)\}$
- $S_2 = \{o_i | d(p_1, o_i) \geqslant d(p_2, o_i)\}$

The generalized hyperplane partitioning does not guarantee a balanced split, hence a nontrivial effort should be spent to properly select the two pivots. An example of 2D vector space, divided into two parts by the generalized hyperplane partitioning, is depicted in the Figure 1.6b.

Extensions

In the previous two paragraphs, we have mentioned two basic partitioning principles used in metric spaces to organize data into groups of similar objects. These two principles can be further extended for particular applications

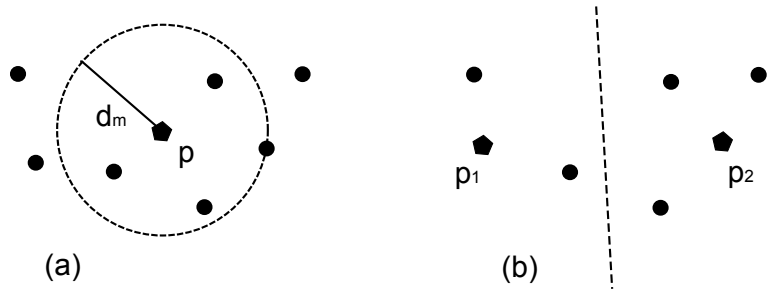


Figure 1.6: (a) The Ball and (b) the Hyperplane partitioning.

to gain more suitable partitions. For example, the excluded middle partitioning [Yianilos, 1999], equipping the ball partitioning with the excluded middle ring around median distance d_m, divides the space into three subsets as follows:

- $S_1 = \{o_i | d(o_i, p) \leqslant d_m - \rho\}$
- $S_2 = \{o_i | d(o_i, p) > d_m + \rho\}$
- $S_3 = \{o_i | d(o_i, p) > d_m - \rho \wedge d(o_i, p) \leqslant d_m + \rho\}$

The excluded middle partitioning has been motivated by similarity queries represented by small ball regions. For such queries and a sufficiently large ρ, the search always prunes at least one of the subsets.

Other possible extensions are summarized as follows:

- Utilizing more pivots or more threshold values to create more partitions in one step.
- Recursive splitting of previously created groups and forming a hierarchy of partitions.

1.4.4 Principles of Filtering

In the previous section, we have shown basic methods of metric space partitioning into equivalence classes of similar objects. All the objects from an equivalence class are bounded by its metric region defined by one or several pivots. The metric regions are then used during the search to avoid exhaustive sequential processing. Hence, correct and effective filtering mechanisms

for metric regions are necessary. Based on the metric postulates, several low-level filtering principles can be defined for each type of metric regions and the most common query shape – the ball region.

Filtering of ball-shaped regions

Having a query ball (q, r_q) and a ball-shaped data region (p, r_p), the data region can be excluded (filtered) from the search if the two balls do not overlap, that is, in case that predicate

$$\delta(q, p) > r_q + r_p$$

is true (see Figure 1.7a). Note that this simple predicate applies also on filtering database objects themselves (rather than regions), considering just database object p, i.e., $r_p = 0$.

Some metric indexes combine two balls to form a ring, which is a pair of two concentric balls, where the smaller one is regarded as a hole in the bigger. In order to determine an overlap with query ball, the previous predicate alone cannot be used to determine that a query ball is entirely inside the hole. Hence, we use another predicate

$$\delta(q, p) < r_p - r_q$$

to determine whether the query ball is entirely inside the hole (see Figure 1.7b). A query ball is not overlapped by a ring region in case the first predicate is true for the bigger ball or the latter predicate is true for the smaller ball (hole).

Filtering of half-space regions

Several metric indexes partition the metric space by use of a border composed of "hyperplanes"[3]. Given m pivot objects, the border is formed by all such points of the universe, which are equally distant to the two closest pivots out of all the pivot objects. A region assigned to pivot object p does not overlap a query region (q, r_q) if the following predicate is true

$$\forall p_i : \delta(q, p) - r_q > \delta(q, p_i) + r_q$$

where $\forall p_i$ are the remaining pivot objects (see Figure 1.7c).

[3]This is just an intuition taken from the L_2 space, because borders do not exist in metric spaces.

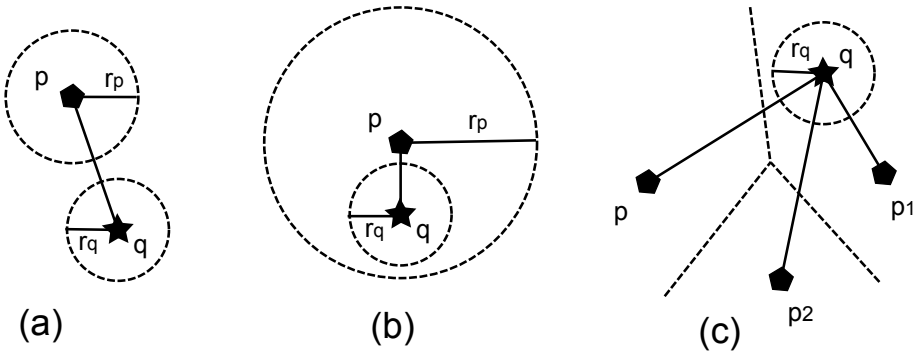


Figure 1.7: Filtering of (a), (b) ball-shaped and (c) half-space regions.

Distance estimation

All the mentioned filtering rules determine regions that can be safely discarded during the search. However, all the filtering rules assume evaluation of distance δ between the query object and the pivots. Although the filtering rules save some cost, the search can be still too expensive due to the need of explicit δ evaluation. Hence, the filtering rules are usually split into two steps. First, the lower or upper bound of the distance δ is estimated and used in a filtering rule. If the rule is fulfilled, the region can be filtered without any distance evaluation. Otherwise, as the second step, the real distance δ is evaluated and used in the filtering rule second time.

A distance $\delta(q, x)$ between the query object q and the database object x can be estimated using a pivot p, to which distances $\delta(q, p)$ and $\delta(x, p)$ have already been evaluated (and stored). Generally, there exist three cases, where such precomputed distances useful for bound determination are available, summarized as follows:

- Several objects are stated as global pivots and the distance matrix between all database objects and pivots is computed. Before a query processing the distances between pivots and the query object must be evaluated as well.

- In a metric index, there are usually stored precomputed distances between database objects and some local pivots, while the distance between the query object and a local pivot has been evaluated during the previous steps of a search algorithm.

- The distance cache is used – for more details about distance caching in metric spaces see [Skopal and Bustos, 2009].

1.4.5 Indicators of Indexability

The topological properties of employed distance measure are necessary but not sufficient for design of a successful access method. Hence, other information based on some kind of statistical analysis over a particular database is needed. Let us mention two important indicators of indexability.

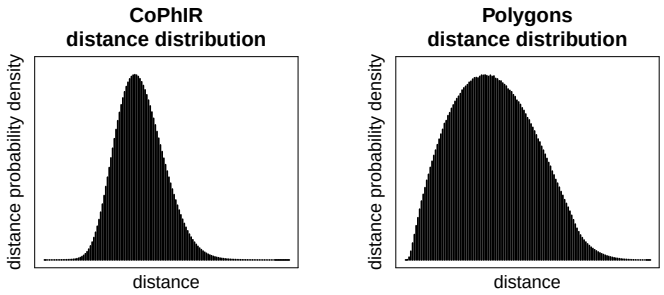


Figure 1.8: Histograms with (a) high and (b) low intrinsic dimensionality

Intrinsic Dimensionality

The distance distribution inside the database $\mathbb{S}$ can reveal whether there appear clusters of objects and how tight they may be. The *intrinsic dimensionality* [Chávez et al., 2001; Chávez and Navarro, 2001] can indicate efficiency limits of any access method and is defined as follows:

$$\rho(\mathbb{S}, \delta) = \frac{\mu^2}{2\sigma^2}$$

Basically, the intrinsic dimensionality of the data space is a global characteristic related to the mean μ and variance σ computed on the set of pairwise distances within the data space. A high intrinsic dimensionality of the data leads to poor partitioning/indexing by any access method (resulting in slower searching), and vice versa. The examples of distance distribution histograms with high and low intrinsic dimensionality are depicted in Figure 1.8. The problem of high intrinsic dimensionality can be considered as a generalization of the well-known *curse of dimensionality* [Böhm et al., 2001] into metric spaces.

Ball-Overlap Factor

The *ball-overlap factor* (BOF) [Skopal, 2007] captures an information about real relationships between data clusters described by particular regions in the distance space. The BOF is defined as:

$$\mathrm{BOF}_k(\mathbb{S},\delta) = \frac{2}{|\mathbb{S}^*| * (|\mathbb{S}^*|-1)} \sum_{\forall o_i,o_j \in \mathbb{S}^*, i>j} sgn(|(o_i,\delta(o_i,kNN(o_i)))\bar{\cap} \bar{\cap}(o_j,\delta(o_j,kNN(o_j)))|)$$

where $\delta(o_i,kNN(o_i))$ is the distance to o_i's k-th nearest neighbor in $\mathbb{S}^* \subset \mathbb{S}$ and $(o_i,\delta(o_i,kNN(o_i)))$ is thus the ball in metric space centered in o_i of radius $\delta(o_i,kNN(o_i))$. The statement $sgn((\cdot,\cdot)\bar{\cap}(\cdot,\cdot))$ returns 1 if the two balls overlap [4] and 0 if they do not. The ball overlap condition is defined as $\delta(o_i,kNN(o_i)) + \delta(o_j,kNN(o_j)) \geqslant \delta(o_i,o_j)$. The BOF can serve us as an appropriate efficiency indicator for access methods based on the ball partitioning.

1.5 Metric Access Methods

Among general techniques to efficient similarity search, the *metric access methods* (MAMs) are suitable in situations where the similarity measure δ is a metric distance. The metric postulates allow us to organize the database $\mathbb{S}$ within equivalence classes, embedded in a data structure which is stored in an index file.

The index is later used to quickly answer typical similarity queries - either a k nearest neighbors (kNN) query like "return the 5 most similar images to my image of a dog", or a range query like "return all voices more similar than 90% to the voice coming from a treetop". In particular, when issued a similarity query, the MAMs exclude many non-relevant equivalence classes from the search (based on metric properties of δ), so only several candidate classes of objects have to be exhaustively (sequentially) searched, see Figure 1.9. In consequence, searching a small number of candidate classes turns out in reduced computation cost of the query.

The efficiency of similarity queries in metric indexes depends on minimization of the computation cost and I/O cost. A semantically complex

[4] In geometric-based, not data-based, meaning.

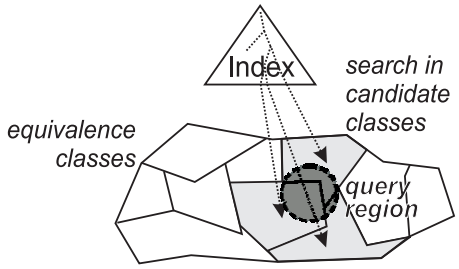


Figure 1.9: Metric access methods.

and sophisticated similarity function is often computed by an expensive algorithm (see the Earth Mover's distance in Section 1.4.1) This is the reason why MAMs focus mainly on the reduction of distance computations and why even the traditional database measure like I/O cost may become negligible. However, when using cheap metric distances, like L_2, we must take multiple types of cost into account.

In the next few Sections, we will mention several orthogonal methods of the metric space organization and pruning. For more details and for other structures, see the monographs [Zezula et al., 2005] and [Samet, 2006].

1.5.1 Sequential File

The sequential file is simply the original database, where any query involves a sequential scan over all the database objects. For a query object q and every database object o_i a distance $\delta(q, o_i)$ must be computed (regardless of query selectivity). Although this kind of "MAM" is not very smart, it does not require any index (i.e., no indexing), which can be useful in many situation. Moreover, it can be simply extended to the D-file structure [Skopal and Bustos, 2009], which has proved to be a competitive index-free metric access method.

1.5.2 D-File

The D-file [Skopal and Bustos, 2009; Skopal et al., 2010] is based on sequential scan and a main-memory structure – the D-cache. The D-cache stores the distances evaluated during previously processed queries. Given a stream of queries $q_1, .., q_n$, the earlier queries can serve as dynamic pivots for the later ones – their distances to database objects are stored in the D-cache. Hence,

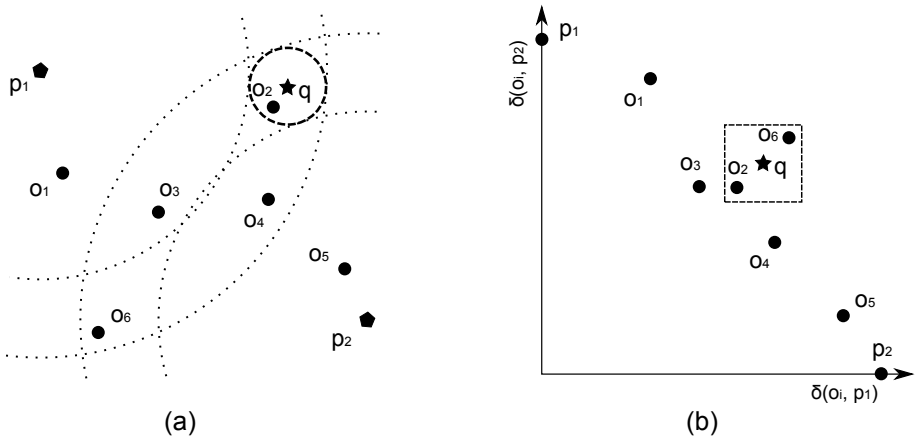


Figure 1.10: (a) The metric space and (b) the "pivot space".

if we evaluate distances between the actually issued query q and the previous queries, we can estimate lower/upper bound distances between q and the database objects.

1.5.3 Pivot Tables

A simple but efficient solution to similarity search under expensive δ represent methods called pivot tables (or distance matrix methods). In general, a set of k pivots is selected from the database, while for every database object a k-dimensional vector of distances to the pivots is created. The vectors belonging to the database objects then form a distance matrix - the pivot table. An example of a metric space and the corresponding "pivot space" is depicted in the Figure 1.10a,b.

When performing a range query (q, r), a distance vector for the query object q is determined the same way as for a database object. From the query vector and the query radius r a k-dimensional hyper-cube is created, centered in the query vector (query point, actually) and with edges of length $2r$. Then, the range query is processed on the pivot table such that database object vectors which do not fall into the query cube are filtered out from further processing. The database objects the vectors of which were not filtered by the query cube have to be subsequently checked by the usual sequential search.

There have been many MAMs developed based on pivot tables. The AESA [Vidal, 1994] treats all the database objects as pivots, so the result-

ing distance matrix has quadratic size with respect to the database size. Also, the search algorithms of AESA is different, otherwise the determination of the distance vector of the query would turn out in a sequential scan of the entire database. The advantage of AESA is empirical average constant complexity of nearest neighbor search, the drawback is the quadratic space complexity and also quadratic time complexity of indexing (creating the matrix) and of the external CPU cost (loading the matrix when querying). The LAESA [Micó, 1992] is a linear variant of AESA, where the number of pivots is assumed far less than the size of the database (so that query vector determination is not a large overhead). The concept of LAESA was implemented many times under different conditions, we name, e.g., TLAESA [Micó et al., 1996] (pivot table indexed by GH-tree-like structure), Spaghettis [Chávez et al., 1999] (pivot table indexed by multiple sorted arrays), OMNI family [Traina et al., 2007] (pivot table indexed by R-tree) and PM-tree [Skopal, 2004] (hybrid approach combining M-tree and pivot tables) described further in the next chapter.

1.5.4 GNAT

The Geometric Near-Neighbor Access Tree (GNAT) [Brin, 1995] is a metric access method that extends the Generalized-Hyperplane Tree [Uhlmann, 1991]. The main idea behind GNAT is to partition the space into subgroups, so-called zones, that contain close objects.

The root node of the tree contains m objects selected from the space, the so-called split points (pivots). The rest of the objects is assigned to their closest split point. The construction algorithm selects with a greedy algorithm the split points, such that they are far away from each other.

Each zone defined by the selected split points is partitioned recursively in the same way (possibly using a different value for m), thus forming a search hierarchy. At each node of the tree, an $O(m^2)$ table stores the range (minimum and maximum distance) from each split point to each zone defined by the other split points. That is,

$$range[i,j]_{o \in zone(sp_j)} = [\min \delta(sp_i, o), \max \delta(sp_i, o)]$$

where sp_i denotes the i^{th} split point and $zone(sp_j)$ denotes the zone defined by split point sp_j.

The original GNAT structure has been further extended to EGNAT [Uribe et al., 2006] by storing the distances from each split point to the split point

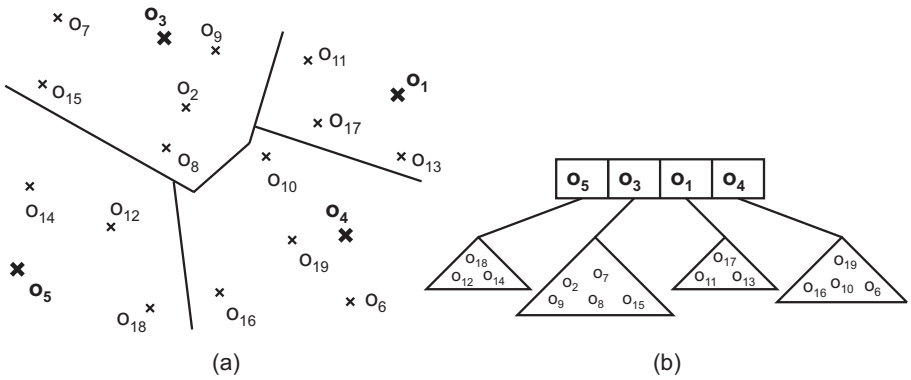


Figure 1.11: GNAT node with $m = 4$ split points (o_1, o_3, o_4, o_5).

of the parent node/zone, similarly as in M-tree. This straightforward improvement can speedup the query performance several times. Besides other enhancements, we name the deletion algorithm utilizing so-called ghost hyperplanes, or special leaf node type "bucket" that stores only the distances to parent which results in smaller index size.

The GNAT performs recursively a range query (q, r) as follows. Starting at the root node, the search algorithm selects one of the split points sp_i from the node. It computes $\delta(sp_i, q)$, and if this distance is equal or smaller than r, it adds sp_i to the result. For all the other split points $sp_j, i \neq j$, the search algorithm computes the intersection between $[\delta(sp, q) - r, \delta(sp, q) + r]$ and range $[i, j]$. If the intersection is empty, the zone of sp_j can be safely discarded as it cannot contain any relevant object (this can be proved using the triangle inequality property of δ). The search algorithm repeats the process with all the split points. Finally, the algorithm searches recursively on all zones that could not be discarded.

1.5.5 M-tree

The M-tree [Ciaccia et al., 1997] is a dynamic index structure that provides good performance in secondary memory (i.e., in database environments). The M-tree index is a hierarchical structure, where some of the data objects are selected as centers (local pivots) of ball-shaped regions, and the remaining objects are partitioned among the regions in order to build up a balanced and compact hierarchy of data regions. Each region (subtree) is indexed recursively in a B-tree-like (bottom-up) way of construction. For more details,

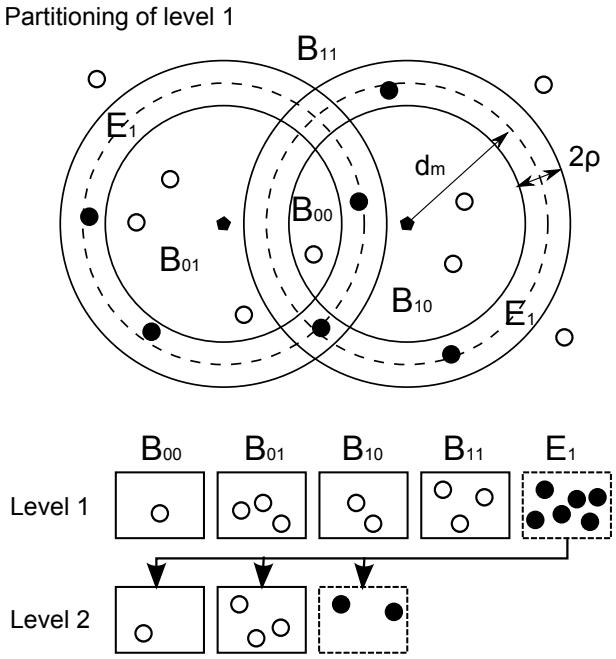


Figure 1.12: Two level D-index structure.

see the following chapter.

1.5.6 D-index

The D-index [Dohnal et al., 2003] is based on the excluded middle ball partitioning (as in case of VP-forest [Yianilos, 1999]) and the idea of similarity hashing [Gennaro et al., 2001]. The D-index uses the partitioning to define an external metric hash function, employing ball partitioning ρ-split functions $bps^{1,\rho,j}(o_i)$ returning:

- 0 if $\delta(o_i, p_j) \leqslant \delta_m - \rho$
- 1 if $\delta(o_i, p_j) > \delta_m + \rho$
- 2 otherwise

where p_j is the pivot assigned to the function $bps^{1,\rho,j}$, d_m is a median distance and ρ determines the thickness of the middle partition.

For k pivots, k such functions are used to form a complex hashing function, that assign a sequence of k digits (from the set $\{0, 1, 2\}$) to each object from the database. Each sequence determines one partition (bucket). If a sequence contains digit 2, the object is inserted to the exclusion set. The hash function partitions the database into up to 2^k buckets and one exclusion set. Since the exclusion set may contain many objects, a new set of pivots can be selected and used to organize objects from the exclusion set into new level of buckets, employing new ρ-split functions. This process may repeat in several iterations until the size of the exclusion set in the last level is sufficiently small. As a result, the multi-level hash table is created, each level using its own set of ball-partitioning ρ-split functions and pivots. The number of pivots may vary for each level. An example of the first-level partitioning (the top part) and the two-level D-index structure (the bottom part) is depicted in Figure 1.12.

When issuing a range query, the distances between the query object and all the pivots used in the first level has to be evaluated, to determine all buckets intersecting the query region. The intersecting buckets have to be sequentially searched, while the remaining buckets can be filtered out. If the query intersects the exclusion set, the next level of the index has to be entered and processed in the same way as the previous level. The authors report very good performance of the D-index in I/O cost and for queries having $r_q \leqslant \rho$.

1.5.7 M-index

A recently introduced MAM, the M-index [Novak and Batko, 2009], employs practically all known principles of metric space pruning and filtering. Inspired by iDistance [Jagadish et al., 2005] (designed for high-dimensional vector spaces), objects from the universe $\mathbb{U}$ are mapped into the real domain, which can be effectively managed by B^+-tree or distributed to more nodes, as shown in the M-chord index [Novak and Zezula, 2006]. Assuming a normalized metric distance δ, the mapping function uses the set of global pivots $p_0, ..., p_{n-1}$ and the Voronoi partitioning. More specifically, each object is assigned a real number key, consisting of the object's distance to the closest pivot (the fractional part of the key) and the index of an assigned Voronoi partition (the integer part of the key). An example of such mapping is depicted in Figure 1.13a. To obtain more partitions using the same number of pivots, repetitive Voronoi partitioning can be used, resulting in multi-level

M-index structure (see Figure 1.13b). The key is computed as:

$$key_l(o) = \delta(p_{(0)_o}, o) + \sum_{i=1}^{l-1} (i)_o n^{l-1-i}$$

where $(.)_o : \{0, ..., n-1\} \mapsto \{0, ..., n-1\}$ is a permutation of indexes such that $\delta(p_{(0)_o}, o) \leqslant \delta(p_{(1)_o}, o) \leqslant ... \leqslant \delta(p_{(n-1)_o}, o)$, l determines the l-prefix of the pivot permutation, $1 \leqslant l \leqslant n$ (the size of the key domain is n^l).

The authors also proposed the dynamic variant of the mapping where the tree of repetitively generated partitions is not balanced – see Figure 1.13, where only the cluster C_2 assigned to pivot p_2 is further expanded to two clusters C_{20} and C_{21}. The dynamic variant better fits the distribution of objects in a database. The modified key function for dynamic M-index is:

$$key_l(o) = \delta(p_{(0)_o}, o) + \sum_{i=1}^{l-1} (i)_o n^{l_{max}-1-i}$$

where the size of the l-prefix of the pivot permutation is bounded by l_{max} value.

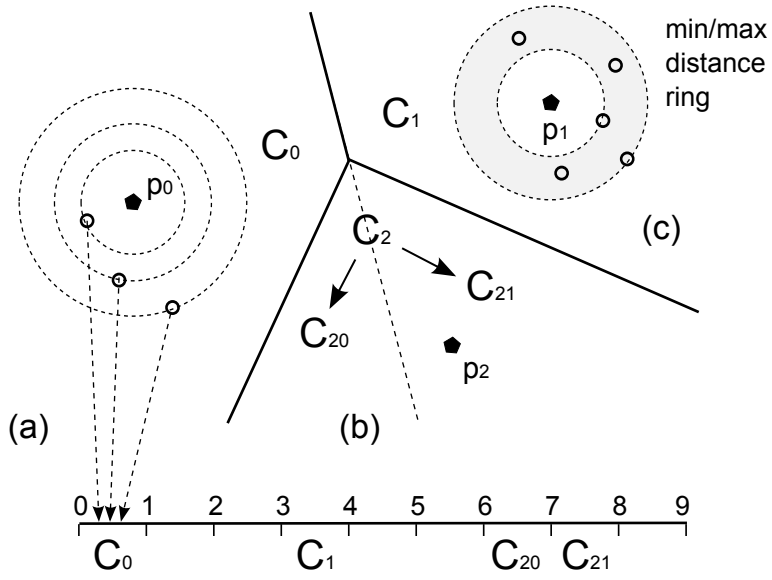


Figure 1.13: (a) Mapping to the key domain (b) repetitive Voronoi partitioning (c) minimal and maximal distance to the closest pivot.

A query processing in the M-index starts with mapping of the query object to the pivot space. Due to the repetitive Voronoi partitioning, filtering of half-space regions can be applied up to l-times. Since the ring defined by distances $\min_{\forall o \in C_i}\{\delta(p,o)\}$ and $\max_{\forall o \in C_i}\{\delta(p,o)\}$ is stored in the structure for each leaf cluster C_i (see Figure 1.13c), filtering of ball-shaped regions can be also employed. If none of the previously mentioned rules filters out a processed cluster, an interval of the searched key domain has to be determined and inspected. Finally, because of storing distances from each object to all global pivots (determined during indexing), the efficient pivot space filtering can be employed. The authors have also proposed efficient and effective approximate search algorithm, that outperforms the state-of-the-art approximate search methods.

1.5.8 Other Methods

Among MAMs, there have appeared also methods that are not based on basic pivot filtering principles (in contrast to previous methods). In the following, we will shortly recall two interesting methods.

Permutation Based Algorithm

The permutation-based algorithm [Chávez et al., 2005] is based on the fact, that a selected set of pivots forms neighborhood for objects in the database. A neighborhood of an object o can be represented as a permutation of pivot indexes $(.)_o : \{0, ..., n-1\} \mapsto \{0, ..., n-1\}$ where $\delta(p_{(0)_o}, o) \leqslant \delta(p_{(1)_o}, o) \leqslant ... \leqslant \delta(p_{(n-1)_o}, o)$. For theoretical results of how many distinct distance permutations may occur in various metric spaces, see [Skala, 2008]. Since similar objects have similar view of their neighborhood (encoded by permutations), we can use permutations as representatives of the original objects. In other words, objects from the database are mapped to the "permutation space", where usually Spearman Rho S_ρ metric is used to evaluate the similarity of two permutations and thus estimate the relative proximity of the original objects. The estimated value is then used in preprocessing heuristics before query processing.

The permutation based algorithm selects a set of pivots and creates index storing permutation $(.)_{o_i}$ for each database object o_i. When issuing a range query (q, r), permutation $(.)_q$ for query object q is computed and $S_\rho((.)_q, (.)_{o_i})$ is evaluated for each stored permutation $(.)_{o_i}$. The original objects are then

sorted in the ascending order (using their estimated proximity to the query object) and processed sequentially evaluating $\delta(q, o_i)$, until some threshold value is reached and the search is stopped. Although the algorithm performs well in cases, where the order induced by S_ρ is close to the order induced by δ, it is just approximative method, because $S_\rho((.)_q, (.)_{o_i}) < S_\rho((.)_q, (.)_{o_j})$ does not guarantee $\delta(q, o_i) < \delta(q, o_j)$. The algorithm has been later extended by indexing of the permutations by arbitrary MAM, resulting in more efficient query processing [Figueroa and Frediksson, 2009], because not all $S_\rho((.)_q, (.)_{o_i})$ have to be evaluated.

Spatial Approximation Tree

The spatial approximation tree (SAT) [Navarro, 1999, 2002] tries to approximate the structure of the *Delaunay graph* (that is a representation of relations in the Voronoi diagram). Having such structure, "spatial" approximation heuristics can be applied during query processing. The graph is constructed in the following manner: an object o is selected as the root and connected to such objects $o_i \in \mathbb{S}$, that are closer to o more than to any other object connected to o. More formally, let $N(o)$ be the set of objects connected to o, then $o_i \in N(o) \Leftrightarrow \forall o_j \in N(o) - \{o_i\} : \delta(o, o_i) < \delta(o_i, o_j)$. The fully dynamic variant of the structure has been proposed in [Navarro and Reyes, 2002].

The range query (q, r) initially selects an arbitrary root object o and tries to subsequently mount to the closest objects using a "spatial" heuristic, which follows a neighbor $o_i \in N(o)$ that is the closest to the query object q. For more details see the referred literature.

1.5.9 M-tree versus other MAMs

Although the M-tree becomes an aging method, which has been outperformed in several aspects (e.g., in distance computations by pivot tables), it still offers an unrevealed potential. Let us mention several arguments why we have revisited the M-tree, summarized as follows:

- The M-tree performs well in the secondary memory. This property is necessary in cases where the employed distance is cheap (e.g., L_p metrics) and the database is large.
- We can create more compact M-tree hierarchies by use of more sophisticated methods of M-tree construction (see Part I), resulting in more

efficient query processing.

- All the M-tree improvements can be utilized by other members of the M-tree family, especially by the PM-tree.
- The M-tree can be employed not only for searching, but also for other purposes – for example, as a hierarchical clustering method for general metric spaces.

To illustrate the impact of more sophisticated M-tree construction, we show our very recent comparison of several MAMs in Figure 1.14. We have compared four different MAMs – D-file using limited distance cache size (20MB of main memory), pivot table using 10 pivots (denoted as PT_10), GNAT (node degree was set to 10) and M-tree (the node degree was 25 in leaf nodes and 24 in inner nodes). We have used two M-tree variants described in Chapter 3. We may observe, that an M-tree constructed by sophisticated methods may become very efficient.

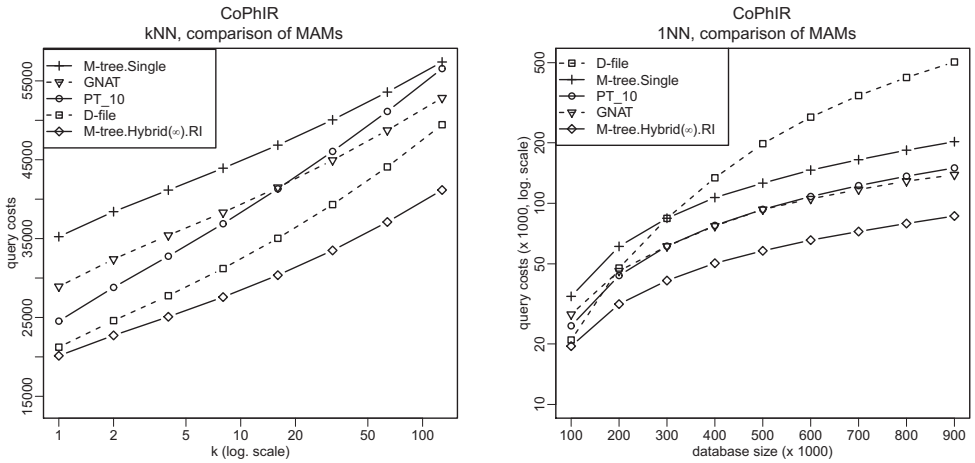


Figure 1.14: kNN queries using MAMs (a) growing k and (b) growing database size

Part I

Revisiting M-tree Construction

Chapter 2

M-tree

Based on properties well-tried in B$^+$-tree and R*-tree [Beckmann et al., 1990], the M-tree [Ciaccia et al., 1997] is a dynamic metric access method suitable for indexing large metric databases. The structure of M-tree represents a hierarchy of nested ball regions, where data is stored in leaves, see Figure 2.1a. Every node has a capacity of m entries and a minimal occupation m_{min}; only the root node is allowed to be underflowed below m_{min}. The inner nodes consist of routing entries $rout(y)$:

$$rout(y) = [y, ptr(T(y)), r_y, \delta(y, Par(y))],$$

where $y \in \mathbb{U}$ is a routing object, $ptr(T(y))$ is a pointer to the subtree $T(y)$, r_y is a covering radius, and the last component is a distance to the parent routing object $Par(y)$ (so-called *to-parent distance*[1]) denoted as $\delta(y, Par(y))$. In order to correctly bound the data in $T(y)$'s leaves, the routing entry must satisfy the *nesting condition*: $\forall o_i \in T(y), r_y \geqslant \delta(y, o_i)$. The routing entry can be viewed as a ball region in the metric space, having its center in the routing object y and radius r_y. A leaf (ground) entry has a format:

$$grnd(z) = [z, oid(z), \delta(z, Par(z))],$$

where $z \in \mathbb{S}$ and $\delta(z, Par(z))$ are similar as in the routing entry, and $oid(z)$ is an external identifier of the original object (z is just an object descriptor).

[1]The to-parent distance is not defined for entries in the root.

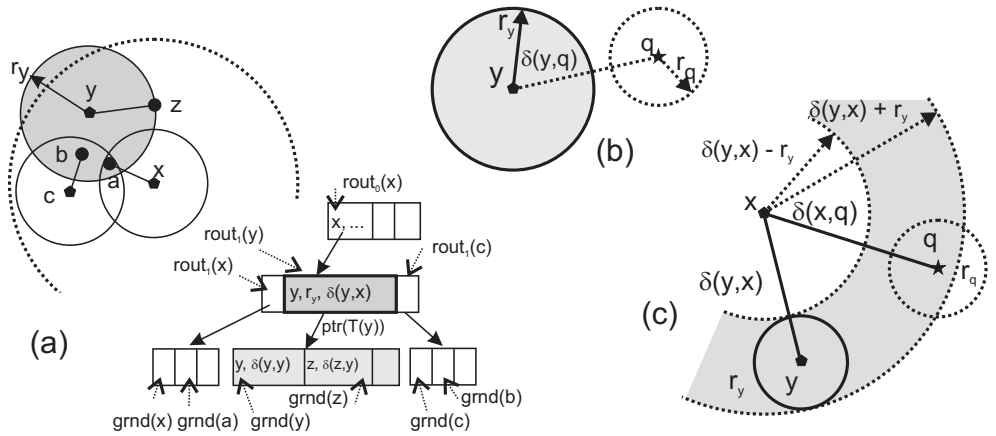


Figure 2.1: (a) An M-tree hierarchy (b) Basic filtering (c) Parent filtering

2.1 Similarity Queries in M-tree

Similarly like the data regions described by routing entries, also the two most common similarity queries (range and kNN query, defined in Section 1.3.1) are described by ball-shaped regions.

The queries are implemented by traversing the tree, starting from the root[2]. Those nodes are accessed, the parent regions of which are overlapped by the query ball. The check for region-and-query overlap requires an explicit distance computation $\delta(y, q)$ (called *basic filtering*), see Figure 2.1b. In particular, if $\delta(y, q) \leqslant r_q + r_y$, the data ball (y, r_y) overlaps the query ball (q, r_q), thus the child node has to be accessed. If not, the respective subtree is filtered from further processing. Moreover, each node in the tree contains the distances from the routing/ground entries to the center of its parent routing entry (the to-parent distances). Hence, some of the non-relevant M-tree branches can be filtered without the need of a distance computation (called *parent filtering*, see Figure 2.1c), thus avoiding the "more expensive" basic overlap check. In particular, if $|\delta(x, q) - \delta(x, y)| > r_q + r_y$, the data ball y cannot overlap the query ball, thus the child node has not to be re-checked by basic filtering. Note $\delta(x, q)$ was computed in the previous (unsuccessful) parent's basic filtering.

To improve the query search efficiency (for the price of precision), tech-

[2]We outline just the principles, for details see the original M-tree algorithms [Ciaccia et al., 1997; Skopal et al., 2003].

niques for the approximate kNN search by M-tree have been also introduced [Zezula et al., 1998]. The techniques utilize three main ideas how to give up the query result precision, summarized as:

- Utilization of a user-defined relative distance error $\epsilon \geqslant 0$, saying the distance between q and an approximation of k-th nearest neighbor must be no more than $(1 + \epsilon)$ times further than the real k-th nearest neighbor.

- A usage of the distance distribution for estimation of the best approximations of kNN, and for early termination stop conditions.

- Stopping the kNN algorithm processing as soon as the intermediate results change only slowly.

2.2 Compactness of M-tree Hierarchy

Since the ball metric regions described by routing entries are restricted just by the nesting condition, the M-tree hierarchy is very loosely defined, while for a single database we can obtain many correct M-tree hierarchies. However, not every M-tree hierarchy built on a database is *compact* enough. In more detail, when the ball regions are either too large and/or highly overlap "sibling" regions, the query processing is not efficient because routing entries of many nodes overlap the query ball. In consequence, large portion of the M-tree hierarchy must be traversed, and so many query-to-object distances have to be computed (resulting in high query costs).

Even if we use always the same construction method, the resulting M-tree hierarchy will still heavily depend on the order in which data objects are inserted (in case of dynamic insertions). A maximally compact M-tree hierarchy(ies) surely exist(s), however, such a construction would require static indexing, and, above all, an exponential construction time. Hence, we would rather prefer an efficient sub-optimal dynamic construction, yet producing sufficiently compact hierarchies.

During the last decade, many methods have been developed to challenge the problem of compact M-tree hierarchies. Besides the original M-tree construction (see Section 2.3), we overview some recent M-tree enhancements in Section 2.4 and present our contributing methods in Sections 3.1 and 3.2.

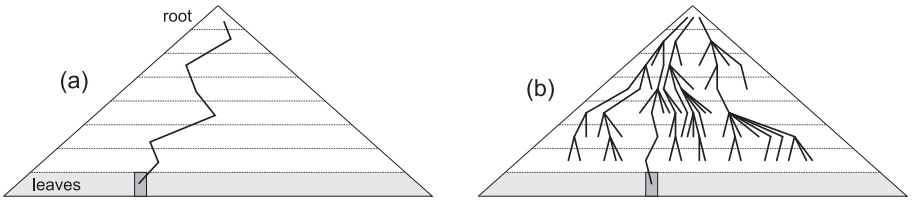


Figure 2.2: (a) Single-way leaf selection (b) Multi-way leaf selection

2.3 Building the M-tree

An M-tree is built in the bottom-up fashion (like B-tree or R-tree), so the data objects are inserted into the leaf nodes. When a leaf overflows, a split is performed – a new leaf is created and some objects are moved from the original leaf into the new one. Two new routing entries are created, one for the original updated leaf and one for the new leaf, and inserted into the parent node (entry for the original leaf is just replaced). All to-parent distances to the new routing objects are computed and replaced in the new leaves' entries. Because of inserting new routing entries, the parent node might overflow as well. In such case a split is performed in a similar way, recursively. If the root node is split, the M-tree grows by one level.

When building an M-tree by dynamic insertions, two main problems have to be solved – the leaf selection and the node splitting.

2.3.1 Leaf Selection

In the original M-tree, a process similar to a point query is performed, in order to find an appropriate leaf for object placement. However, in contrast to a point query, only one vertical path of the tree is passed. This approach is also referred to as the *single-way* (or deterministic) insertion, see Figure 2.2a. When navigating the tree, the next node in the path is chosen such that the inserted object fits the appropriate region best (for details see [Ciaccia et al., 1997; Skopal et al., 2003]).

2.3.2 Node Splitting

The node splitting policy is a significant factor of the M-tree building process. When a node is split, two new routing entries have to be created, representing new ball regions. To guarantee a compact M-tree hierarchy, the splitting

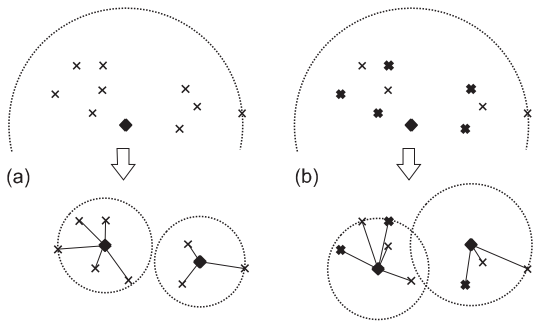


Figure 2.3: Leaf split using (a) *CLASSIC* and (b) *SAMPLING* heuristics

process must ensure the new regions are separated as much as possible, they overlap as least as possible, and they are of minimum volumes (radii).

To best fit these requirements, all the objects in the node are candidates to the routing objects. For each pair of candidate routing objects, the resulting nodes are temporarily created and radius of the greater region is determined. Such pair of candidate routing objects is finally chosen, which has the smallest radius of the greater region (so-called *mM_Rad* choice). This *CLASSIC* approach exhibits $O(m^2)$ complexity, where m is the capacity of the node. To avoid the quadratic complexity, there were alternative heuristics developed:

– The *RANDOM* approach directly selects two new routing objects at random, which cuts the complexity down to $O(m)$.
– Instead of considering all objects in the node as candidate routing objects, the *SAMPLING* approach selects randomly just s candidates ($s < m$). Then the complexity of node splitting is $O(ms)$.

In Figure 2.3 see the result of leaf splitting, comparing the CLASSIC and SAMPLING heuristics. A splitting of non-leaf nodes is similar, though it must take also the radii of the redistributed routing entries into account.

2.4 Related Work

The success of M-tree can be supported by the existence of its many descendants that have appeared during the past decade. We could chronologically name the Slim-tree [Traina Jr. et al., 2000] (discussed in Section 2.4.1), and the M$^+$-tree [Zhou et al., 2003] which employs further partitioning of

the node by a hyper-plane (i.e., an approach limited to Euclidean spaces). Furthermore, let us mention the PM-tree [Skopal, 2004; Skopal et al., 2005] which combines the M-tree with pivot-based techniques (described in Section 2.5), the M^2-tree [Ciaccia and Patella, 2000] and M^3-tree [Bustos and Skopal, 2006] which uses an aggregation of multiple metrics. As the most recent ones we point out to M^*-tree [Skopal and Hoksza, 2007], where each node is additionally equipped by a nearest-neighbor graph, and the NM-tree [Skopal and Lokoč, 2008] which allows also nonmetric distances. In the rest of the chapter we consider the original structural properties of M-tree [Ciaccia et al., 1997] (i.e., we consider modified algorithms, not the structure).

The effectiveness of query processing in M-tree heavily depends on the M-tree compactness, hence, on the construction algorithm used. Intuitively, to improve the search performance, the construction should be more expensive, and vice versa. For example, if we use the *RANDOM* node split heuristic, we obtain low construction costs, but the region volumes/overlaps will increase, and so the query costs will rapidly increase as well. In the following we present three approaches of compact M-tree construction.

2.4.1 Slim-down Algorithm

The authors of Slim-tree [Traina Jr. et al., 2000] proposed two new M-tree construction techniques. First, a node splitting policy was introduced, based on minimum spanning tree. Instead of choosing many candidate pairs to new routing entries and then temporarily partitioning the node entries for each candidate pair, in Slim-tree the complete distance graph between node entries is used to construct the minimum spanning tree (MST). Then, the longest edge in MST is removed in order to obtain two separate sets of entries – these sets directly constitute the two new nodes. Although the MST splitting still needs $O(m^2)$ distance computations due to the complete distance graph, there are $O(m)$ external CPU costs saved, otherwise used for the temporary partitioning. Second, the slim-down algorithm was presented, a post-processing method trying to redistribute ground entries into more suitable leaves. This technique produces very compact M-tree/Slim-tree hierarchies, however, it is also very expensive – up to linear with database size for a single ground entry redistribution.

The slim-down algorithm was later generalized in order to redistribute also routing entries at higher M-tree levels [Skopal et al., 2003], which leads to even more compact hierarchies (see Figure 2.4).

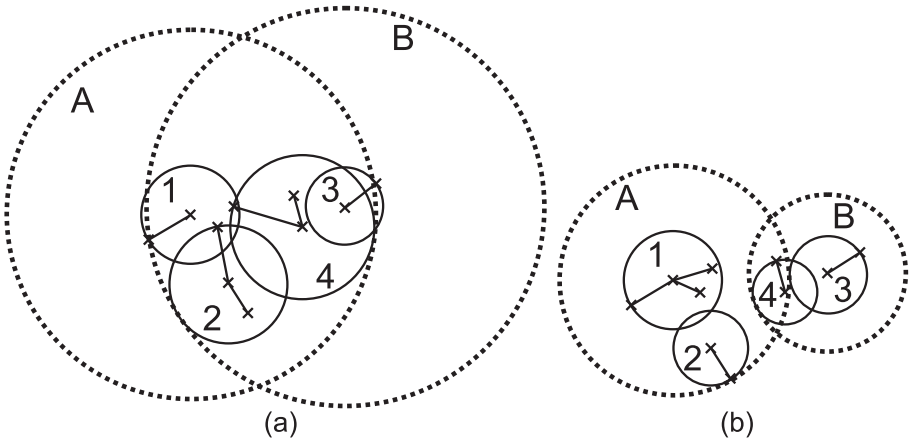


Figure 2.4: An M-tree before (a) and after (b) generalized slim-down algorithm run

2.4.2 Multi-way Leaf Selection

Another way of improving M-tree compactness is an employment of more sophisticated selection of target leaf wherein a new object will be inserted. In [Skopal et al., 2003] the *multi-way* (non-deterministic) leaf selection was proposed. The target leaf is found such that a point query (i.e., range query with $r_q = 0$) is issued having the new inserted object in the role of q. All the "touched" and non-full leaves serve as candidates to the target leaf. Among the touched non-full leaves the one is chosen which has its parent routing object closest to the inserted object (see Figure 2.2b).

The multi-way leaf selection has positive impact on the M-tree compactness, though not as large as the generalized slim-down algorithm. On the other hand, the insertion employing multi-way leaf selection is by far less expensive than the generalized slim-down algorithm, though still up to linearly expensive with database size for a single insertion.

2.4.3 Bulk Loading

The basic idea of bulk loading is to statically create the index from scratch but knowing beforehand the database. Then some optimizations may be performed to obtain a "good" index for that database. Usually, the proposed bulk loading techniques are designed for specific index structures, but there have been proposals for more general algorithms. For example, in

[den Bercken and Seeger, 2001] the authors propose two generic algorithms for bulk loading, which were tested with different index structures like the R-tree and the Slim-tree. Note that the efficiency of the index may degrade if new objects are inserted after its construction. Specific bulk loading techniques for M-tree were introduced in [Ciaccia and Patella, 1998; Sexton and Swinbank, 2004], the latter one furthermore introduces, for the first time, dynamic deletions on M-tree (renamed to SM-tree here). Another bulk loading algorithm for Slim-tree was recently proposed in [Vespa et al., 2007].

Sometimes the bulk loading is viewed as a technique for fast index construction, rather than a tool for building a compact index hierarchy.

2.5 PM-tree

The idea of PM-tree [Skopal, 2004; Skopal et al., 2005] is to enhance the hierarchy of M-tree by an information related to a static set of k global pivots $p_i \in P \subset \mathbb{U}$. In a PM-tree's routing entry, the original M-tree-inherited ball region is further cut off by a set of rings (centered in the global pivots), so the region volume becomes more compact. Similarly, the PM-tree ground entries are enhanced by distances to the pivots, which are interpreted as rings as well. Each ring stored in a routing/ground entry represents a distance range (bounding the underlying data) with respect to a particular pivot.

A routing entry in PM-tree inner node is defined as:

$$rout_{PM}(y) = [y, r_y, \delta(y, \mathrm{Par}(y)), ptr(T(y)), \mathrm{HR}],$$

where the new HR attribute is an array of k_{hr} intervals ($k_{hr} \leqslant k$), where the t-th interval HR_{p_t} is the smallest interval covering distances between the pivot p_t and each of the objects stored in leaves of $T(y)$, i.e., $\mathrm{HR}_{p_t} = \langle \mathrm{HR}^{min}_{p_t}, \mathrm{HR}^{max}_{p_t} \rangle$, $\mathrm{HR}^{min}_{p_t} = min\{\delta(o_j, p_t)\}$, $\mathrm{HR}^{max}_{p_t} = max\{\delta(o_j, p_t)\}$, $\forall o_j \in T(y)$. The interval HR_{p_t} together with pivot p_t define a ring region (p_t,HR_{p_t}); a ball region (p_t,$\mathrm{HR}^{max}_{p_t}$) reduced by a "hole" (p_t,$\mathrm{HR}^{min}_{p_t}$).

A ground entry in PM-tree leaf is defined as:

$$grnd_{PM}(z) = [z, id(z), \delta(z, \mathrm{Par}(z)), \mathrm{PD}],$$

where the new PD attribute stands for an array of p_{pd} pivot distances ($p_{pd} \leqslant p$) where the t-th distance $\mathrm{PD}_{p_t} = \delta(y, p_t)$.

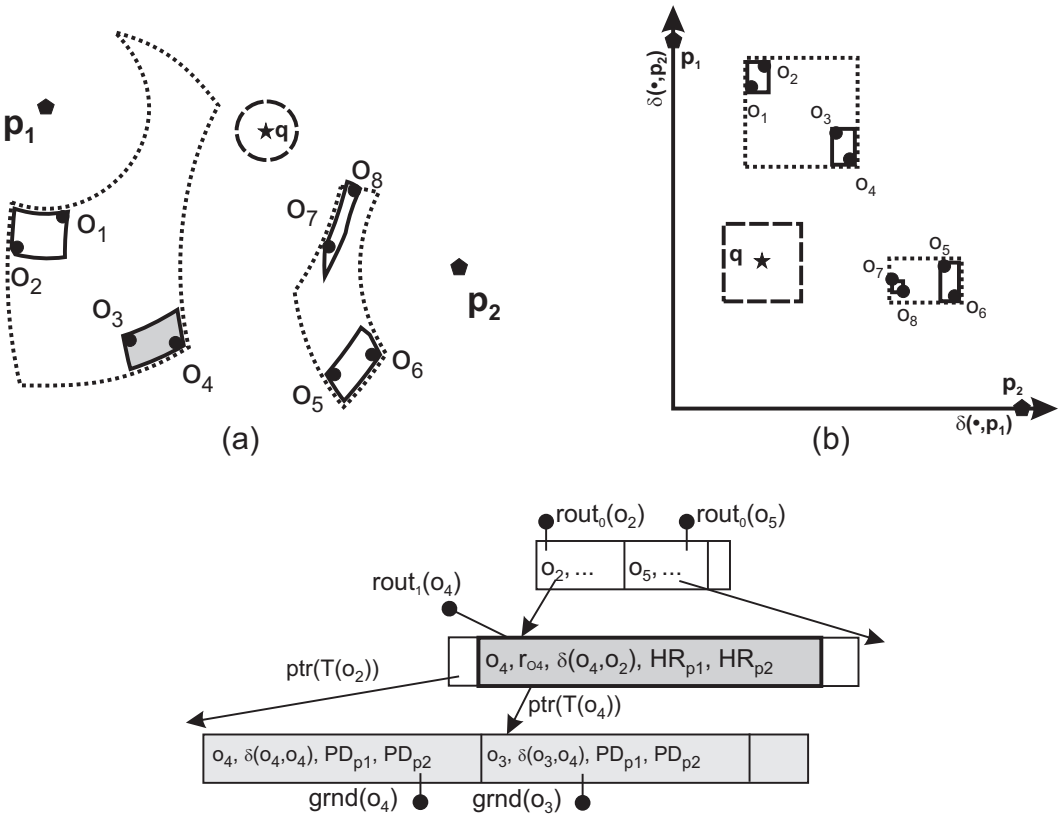


Figure 2.5: (a) PM-tree employing 2 pivots (p_1, p_2). (b) Projection of PM-tree into the "pivot space".

The combination of all the k entry's ranges produces a k-dimensional minimum bounding rectangle (MBR), hence, the global pivots actually map the metric regions/data into a "pivot space" of dimensionality k (see Figure 2.5b). The number of pivots can be defined separately for routing and ground entries – we typically choose less pivots for ground entries to reduce storage costs (i.e., $k = k_{hr} > k_{pd}$).

When issuing a range or kNN query, the query object is mapped into the pivot space – this requires p extra distance computations $\delta(q, p_i), \forall p_i \in P$. The mapped query ball (q, r_q) forms a hyper-cube $\langle \delta(q, p_1) - r_q, \delta(q, p_1) + r_q \rangle \times \cdots \times \langle \delta(q, p_k) - r_q, \delta(q, p_k) + r_q \rangle$ in the pivot space that is repeatedly utilized to check for an overlap with routing/ground entry's MBRs (see Figures 2.5a,b). If they do not overlap, the entry is filtered out without any

distance computation, otherwise, the M-tree's filtering steps (parent & basic filtering) are applied. Actually, the MBRs overlap check can be also understood as L_∞ filtering, that is, if the L_∞ distance[3] from a PM-tree region to the query object q is greater than r_q, the region is not overlapped by the query.

Note the MBRs overlap check does not require an explicit distance computation, so the PM-tree usually achieves significantly lower query costs when compared with M-tree – for more details, see [Skopal, 2004, 2007; Skopal et al., 2005].

[3]The maximum difference of two vectors' coordinate values.

Chapter 3

Forced Reinsertions and Hybrid Way Leaf Selection

In this chapter, we have focused on new methods of the dynamic M-tree construction improving the quality of the M-tree hierarchy, and which try to keep the construction costs acceptable.

3.1 Forced Reinsertions

The first contribution we propose in this thesis is an adaptation of forced reinsertions into the process of dynamic insertions in M-tree. The *forced reinsertions* is a technique well-known from the R*-tree [Beckmann et al., 1990]. The idea is based on an easy principle. Some objects are removed from a leaf to avoid a split operation and then inserted in a common way under a hope that the reinserted objects will arrive into more "suitable" leaves. There are two basic motivations to consider forced reinsertion as beneficial, considering any B-tree-based spatial/metric index structure. The straightforward (but also weaker) motivation is better node occupancy, hence, forced reinsertions lead to fuller nodes. Second, due to unavoidable node splitting over the time, the compactness of spatial/metric region hierarchy deteriorates – the region volumes and overlaps grow because of spatial aggregations mixing old and new objects/regions. Here the forced reinsertions could serve as an opportunity to move some "bad" (volume- or overlap-inflating) objects from the leaf.

In M-tree, we have to face some specific issues when implementing forced

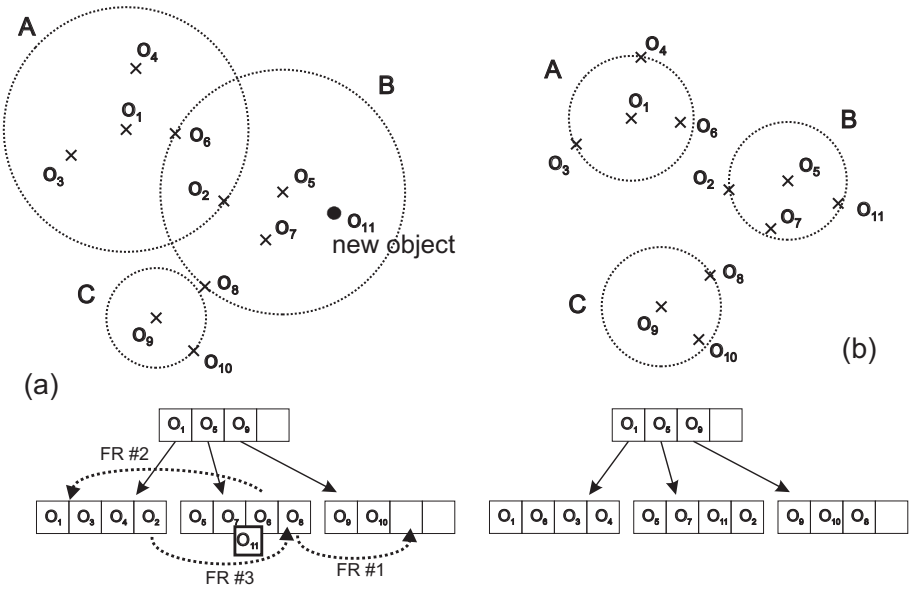


Figure 3.1: (a) Before reinsertions (b) Decreased overlaps/volumes after 3 reinsertions

reinsertions. Basically, when a new object is inserted into a leaf that is now about to split, some suitable objects from the leaf must be selected and reinserted. The crucial goal is to propose a method aiming to decrease the covering radius of the reinserted leaf as much as possible, while simultaneously aiming to enlarge the radii of leaves accepting the reinserted objects as little as possible. Here we have to take also the induced leaf splits/reinsertions into account, that is, a forced reinsertion attempt could raise a chain of reinsertions terminated by regular splits "after a while".

As a fundamental assumption, we expect objects located close to the region's "border" have higher probability to be suitably reinserted than the more "centered" ones. Since in an M-tree node the entries are ordered according to their distances to the parent routing object (region's center), we can select the furthest ones (close to the border) easily.[1] In Figure 3.1 see a motivation example – situation just before a leaf split, and how the split is avoided after a series of induced reinsertions, denoted as FR#1, FR#2, FR#3. We can see that not only the split was prevented, but the M-tree

[1]Remember the precomputed distances to the routing entry (the to-parent distances) are stored in all entries except those in the root node.

compactness was improved, too.

We propose two variants using forced reinsertions for M-tree – the full reinsertions and conservative reinsertions.

3.1.1 Full Reinsertions

As mentioned before, we assume the most suitable entries for reinserting are the furthest ones from the parent routing object. To avoid an overfull leaf split, some of its furthest entries are removed from the leaf and pushed onto a temporary main memory stack $\mathcal{S}$. The covering radius of the leaf's parent routing entry is then immediately reduced to the distance of the routing object to the new furthest entry in the leaf (and so the covering radii of all ancestors). Then, the current entry on the top of $\mathcal{S}$ is reinserted in a standard way as it would be a regular new object to be inserted. Naturally, a forced reinsertion could possibly induce further reinsertion attempts (i.e., the top of the stack grows). The reinsertions are repeated until the stack becomes empty.

Recursion Depth

Since a single reinsertion attempt could generally raise a long chain of subsequent reinsertions (the stack is inflating instead of emptying), we would like to limit the number of forced reinsertion attempts to keep the construction costs reasonable and scalable. We denote the limit as a user-defined *recursion depth* parameter. When the limit of reinsertion attempts is reached, the remaining entries on the stack are popped and reinserted such that only regular splits are allowed from now on (i.e., the stack does not grow anymore).

Entries Removing

As to the entries removing mentioned before, we remove at most k furthest entries from the leaf in the direction from closer to further ones (where k is user-defined). However, if the newly inserted object is within the k entries, we remove just the more distant of the k entries (i.e., we do not remove the new one and all closer). We called such a removing of entries as the *reverse pessimistic* entries removing, see Figure 3.2.

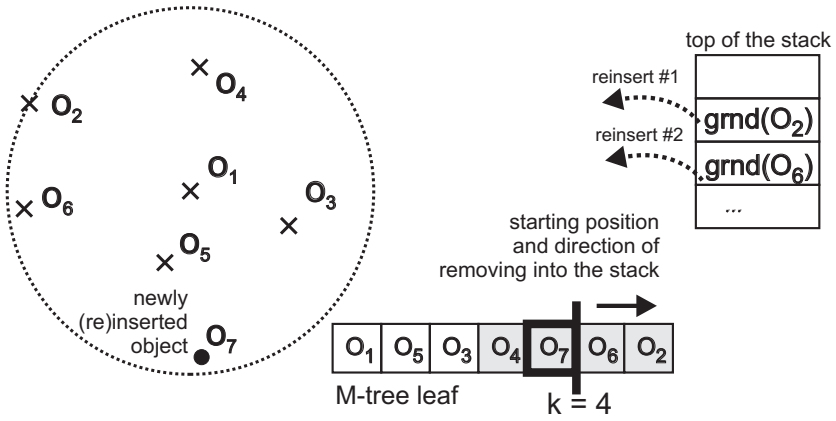


Figure 3.2: Reverse pessimistic entries removing

As a motivation for the reverse pessimistic removing, we suppose that the reinsertion of an object being just newly inserted (and all closer ones) would cause insertion back to the same leaf (the pessimistic assumption). As to the direction of entries removing, being "reverse" due to starting from the leaf's "middle", we assume the furthest entries (being the outlying "losers") should be reinserted first. This heuristics aims to increase the likelihood of finding a suitable non-full leaf, being otherwise possibly occupied by the other removed "sibling" entries on the stack. Although we tried also other variants of entries removing than the reverse pessimistic (as described and evaluated in [Lokoč and Skopal, 2008]), they performed worse, so we do not consider them anymore in this thesis.

To provide a comprehensive description, in Listing 1 see the pseudocode of dynamic insertion enhanced by full forced reinsertions.

Listing 1. (insertion with full forced reinsertions)

```
let maxRemoved be maximal number of removed entries (user-defined)        // denoted k in Section 3.1.1
let recursionDepth be the maximal depth of recursion (user-defined)

method Insert(o_new) {
  find leaf L for o_new                                                   // "either-way" leaf selection
  insert o_new into L

  if L is not overfull then return
  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)
  exclude grnd(o_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then {
    for (j = 0; j < |𝔼|; j++) {                                           // remove furthest entries from leaf
      𝒮.Push(𝔼.GetEntry(1))
      𝔼.DeleteEntry(1)
    }
    decrease radius of L (and possibly of its ancestors)

    while (𝒮 is not empty) {                                              // reinsert removed entries
      recursionDepth = recursionDepth − 1
      Insert(𝒮.Pop())
    }
  } else {
    perform regular split of L (and possibly of its ancestors)
  }
}
```

3.1.2 Conservative Reinsertions

As observed in [Lokoč and Skopal, 2008] and as shown in experiments, the full forced reinsertion variant is effective in producing compact M-tree hierarchy, though it is still quite expensive in terms of M-tree construction costs. The reverse pessimistic strategy ensures the newly inserted entry and all closer entries will not be reinserted (probably back to the same leaf). However, there can still occur reinsertions of more distant entries into the same leaf, being thus ineffective. In this section, therefore, we introduce an improvement of full forced reinsertions, called the *conservative forced reinsertions*, better avoiding reinsertions of entries back into the same leaf (see also Listing 2 for pseudocode).

The improvement requires a slight extension of the M-tree's ground entry format, as $grnd(z) = [z, oid(z), \delta(z, Par(z)), SplitNumber]$. The $SplitNumber$ is the number of node splits occurred before (re)inserting this entry into the current leaf. In fact, the $SplitNumber$ represents a logical time related to the amount of structural changes in M-tree during its construction.

The way of removing of entries onto the stack $\mathcal{S}$ is the same as used in the full forced reinsertion variant (i.e., using the reverse pessimistic strategy). The difference is in the processing of popped entries from stack, and in the structure of a stack entry. Instead of storing pure objects $o_i \in \mathbb{S}$ on the stack, now we store the entire ground entry $grnd(o_i)$ and, additionally, a pair of co-identifiers determining wherefrom the entry came. The first identifier is an id of the source leaf, while the second one is the id of the source leaf's routing entry (see an example in Figure 3.3 right). The reason for two identifiers of a leaf is that we want to distinguish leaves whose node id remained the same but they obtained a different parent routing entry (caused by a possible split).

The Stack Processing

After the entries are removed onto the stack, the top entry is popped and reinserted in the usual way. We check whether it falls back to the leaf it came from, that is, whether the leaf+parent entry co-identifiers are equal to that of the entry popped from stack. If so, the top of stack is checked for a contiguous block of entries. In such a contiguous block all entries must become also from the same leaf as the first reinserted entry, and all must be "younger" (their $SplitNumber$ is higher). If such a block exists, its entries are popped and directly *moved* to the respective leaf, following the first reinserted entry. Actually, they are returned to their original leaf for free; no regular insertion is performed for them (no distance has to be computed). Otherwise, if such a contiguous block of entries does not exist (or its processing was finished), the entries on the top of stack are reinserted in the usual way.

For an example, in Figure 3.3 the top entry on the stack was reinserted into a leaf 34, however, the next one arrived into the same leaf 23 as it came from. Therefore, a single-entry block on the stack was identified, fulfilling the block conditions (i.e., originating also from leaf 23 and being younger by 3 splits), and moved back to the leaf. The next entry on the stack also came from leaf 23 but was older, so for this entry there is a greater probability that during its longer "sitting" in leaf 23 there appeared better leaves in the M-tree, so this one is properly reinserted to the leaf 16.

The motivation for the above described optimization is a conservative assumption, that if a set of entries was removed onto the stack from the same node at the same time, and one of them was reinserted back into the same

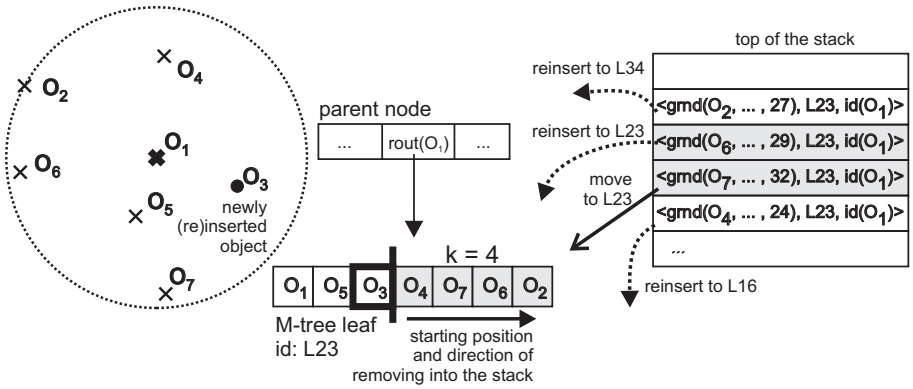


Figure 3.3: Conservative forced reinsertions

leaf, then also some other entries in this set would be probably reinserted into the same leaf as well. Hence, instead of ineffective costly reinsertions we rather "give up" and move the entries directly back.

Listing 2. (insertion with conservative forced reinsertions)

```
let maxRemoved be maximal number of removed entries (user-defined)        // denoted k in Section 3.1.1
let recursionDepth be the maximal depth of recursion (user-defined)

method Insert(o_new, SplitCount) {
  find leaf L for o_new                                                   // "either-way" leaf selection
  insert o_new into L
  let rout(o_p) be L's parent routing entry

  while (S is not empty and S.TopEntry.NodeId = L.NodeId and
      S.TopEntry.GroundEntry.RoutId = L.RoutId and S.TopEntry.GroundEntry.SplitCount ⩾ SplitCount) {
    L.Insert(S.Pop().GroundEntry)                                         // move the entries back to the original leaf
    if (L is overfull) then {
      perform regular split of L (and possibly of its ancestors)
      return
    }
  }
  if L is not overfull then return
  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)
  exclude grnd(o_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then {
    for (j = 0; j < |𝔼|; j++) {                                           // remove furthest entries from leaf
      S.Push(⟨𝔼.GetEntry(1), L.id, rout(o_p).id⟩)
      𝔼.DeleteEntry(1)
    }
    decrease radius of L (and possibly of its ancestors)

    while (S is not empty) {                                              // reinsert removed entries
      recursionDepth = recursionDepth − 1
      let entry = S.Pop().GroundEntry
```

```
    Insert(entry.object, entry.SplitCount)
   }
  } else {
   perform regular split of L (and possibly of its ancestors)
  }
}
```

Additional Notes

- The to-parent distance stored in a moved ground entry is still valid. Actually, this is another reason why we use additionally the parent routing entry identifier to co-identify the source leaf.
- When a ground entry is reinserted, its *SplitNumber* is updated only in case it has not been reinserted/moved back into the same leaf.
- Due to the reverse pessimistic strategy, the moved entries are always closer to the parent routing entry than the entry reinserted to the same leaf as first. Hence, the leaf's covering radius cannot be inflated due to entries moving. However, the covering radii of its ancestor inner nodes (from the leaf to the root) should be inflated and so the radii have to be re-checked.

3.1.3 Construction vs. Query Efficiency

The rationale for forced reinsertions is two-fold. First, reinsertions could clearly improve the compactness of M-tree (thus the query performance) at the cost of (a bit) more expensive construction. The second reason considers the trade-off between indexing and querying performance. In contrast to the first reason (speeding up querying), sometimes we would like to decrease construction costs but simultaneously keep the query costs as low as if used more expensive construction. With forced reinsertions this goal could be carried out. For example, the *CLASSIC* splitting of M-tree node is expensive but brings faster queries, while the *SAMPLING* splitting is cheaper but also leads to slower queries. Since the *CLASSIC* splitting could produce M-tree which is compact enough, at some scenarios the employment of forced reinsertions could not bring any further improvement – so only the construction costs grow, but the retrieval performance is not improved. In

such case we might rather employ the forced reinsertions together with the *SAMPLING* splitting. This way we could achieve retrieval costs similar to that of *CLASSIC* splitting, however, for cheaper construction – somewhere between *SAMPLING* and *CLASSIC* without forced reinsertions. In other words, forced reinsertions might cheaply fix the bad data partitioning caused by *SAMPLING* splitting.

Finally, we have to emphasize that when combined with the single-way leaf selection (or the hybrid-way leaf selection, see next section), the asymptotic complexity of a single insertion is still logarithmic with database size. In more detail, the number of reinsertion attempts is limited by a constant (recursion depth), the maximal number of entries in a leaf is constant, while the single/hybrid-way leaf selection is of logarithmic complexity.

3.1.4 Average Leaf Node Utilization

We also propose a new method which utilizes reinsertions to guarantee a user-defined average leaf node utilization. This method has not been published yet.

The leaf selection strategies and reinsertions affect average leaf node utilization in M-tree. Moreover, we expect that using the multi-way leaf selection for a reinserted object leads to higher leaf node utilization ϵ_{MW}, because more paths are examined to find the best fitting non-full leaf node. On the other hand, using the single-way leaf selection during reinsertions results in lower leaf node utilization ϵ_{SW}, because less information can be employed during the reinsertion of an object. We can use these observations to control (to some extent) an average leaf node utilization by employing both single-way and multi-way leaf selection strategies for reinserted objects.

In Listing 3, we present a modification of the forced reinsertions schema, that guarantees a user-defined average leaf node utilization ϵ, which can reach values from the interval $\langle \epsilon_{SW}, \epsilon_{MW} \rangle$. We may also observe, that a larger/smaller value of ϵ leads to more/less multi-way leaf selections and thus to more/less expensive indexing.

Listing 3. (insertion guaranteeing average leaf node utilization)

```
let maxRemoved be maximal number of removed entries (user-defined)        // denoted k in Section 3.1.1
let recursionDepth be the maximal depth of recursion (user-defined)
let epsilon be the requested average leaf node utilization (user-defined)

method Insert(o_new, SplitCount, LeafSelection) {
  find leaf L for o_new using LeafSelection strategy
  insert o_new into L
  let rout(o_p) be L's parent routing entry

  while (S is not empty and S.TopEntry.NodeId = L.NodeId and
      S.TopEntry.GroundEntry.RoutId = L.RoutId and S.TopEntry.GroundEntry.SplitCount ⩾ SplitCount) {
    L.Insert(S.Pop().GroundEntry)                          // move the entries back to the original leaf
    if (L is overfull) then {
      perform regular split of L (and possibly of its ancestors)
      return
    }
  }
  if L is not overfull then return
  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)
  exclude grnd(o_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then {
    for (j = 0; j < |𝔼|; j++) {                            // remove furthest entries from leaf
      S.Push(⟨𝔼.GetEntry(1), L.id, rout(o_p).id⟩)
      𝔼.DeleteEntry(1)
    }
    decrease radius of L (and possibly of its ancestors)

    while (S is not empty) {                                // reinsert removed entries
      recursionDepth = recursionDepth − 1
      let entry = S.Pop().GroundEntry
      if epsilon > GetAverageLeafNodeUtilization()
        Insert(entry.object, entry.SplitCount, 'Multi-Way')
      else
        Insert(entry.object, entry.SplitCount, 'Single-Way')
    }
  } else {
    perform regular split of L (and possibly of its ancestors)
  }
}
```

3.2 Hybrid-way Leaf Selection

As the second contribution, we introduce so-called *hybrid-way leaf selection*. The rationale for this effort was the performance gap between single-way and multi-way leaf selection (see Sections 2.3.1 and 2.4.2). On the first hand, the single-way selection is very cheap (logarithmic with database size) but often selects a leaf that is not optimal, thus the resulting M-tree compactness

is not very good. On the other hand, the multi-way technique selects an optimal leaf (though only a non-full one), but it is expensive, up to linear with database size.

Instead of the multi-way's expensive traversal to all leaves whose regions could cover the new object, the hybrid-way technique selects only a limited number of the "best" candidate nodes at each level. Such nodes become the candidates, the regions of which cover the newly inserted object and their routing objects are as close to the new object as possible. All covering child nodes of the selected candidate nodes are then followed down to the next M-tree level, while, again, only a limited number of the best ones are selected as the candidate nodes, and so on. After the pre-leaf level is reached, the candidate pre-leaves are checked for the best routing entry and the respective leaf is returned as the finally selected leaf. In the rather unlikely situation when no candidate nodes are selected at a level (i.e., the new object is not covered by any node's ball), the hybrid-way technique gives up and selects the leaf by single-way selection. The limits of candidates at all levels are described by so-called *branching vector*. The branching vector determines how many paths in M-tree the hybrid-way selection traverses, see an example in Figure 3.4.

The hybrid-way solution represents a scalable technique, actually generalizing both the single- and multi-way selections. If the branching vector contains only 1s, we obtain a single-way-like behavior, though not the same – the single-way always selects a node at any level, the hybrid-way does not have to. If the branching vector contains only ∞s, we obtain a multi-way-like behavior (though the original multi-way selects only non-full leaves). If all the numbers in the vector are equal, we can talk just about a *branching factor* $f \in \langle 1, \infty)$. In Listing 4 see the hybrid-way leaf selection. Note the

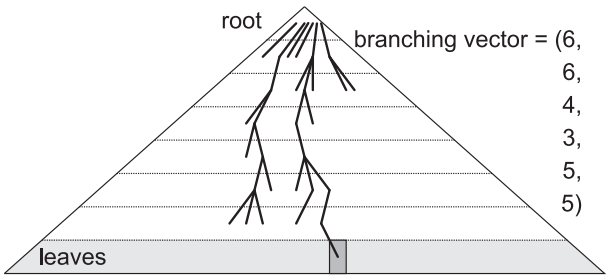


Figure 3.4: Hybrid-way leaf selection

algorithm uses the parent filtering (Section 2.1) to efficiently filter out the non-covering nodes. Because the number of traversed paths is limited by the branching vector (consisting of constants), the hybrid-way selection is still of logarithmic complexity, though more expensive than the single-way selection by a constant factor.

Listing 4. (hybrid-way leaf selection)

```
method FindLeafHybridWay(o_new, branching vector v) {
  let nodeCandidates = {⟨root; 0⟩} // the 1st value in ⟨·,·⟩ is denoted .Node, the 2nd .ObjectToParentDistance
  let targetLeaf = ∅
  let minDistance = ∞

  for (level = 0; level < treeHeight; level++) {
    let levelCandidates = ∅
    for each can in nodeCandidates {
      for each entry in can.Node {
        if |can.ObjectToParentDistance − entry.RoutingToParentDistance| ⩽ entry.radius then { // parent filt.
          compute δ(entry.object, o_new)
          if δ(entry.object, o_new) < entry.radius then {                          // basic filtering
            if level = treeHeight − 1 then {                    // at pre-leaf level select the winning leaf
              if δ(entry.object, o_new) < minDistance then {
                minDistance = δ(entry.object, o_new)
                targetLeaf = entry.childNode
              }
            } else {                                            // at higher levels follow the child nodes
              read entry.childNode
              add ⟨entry.childNode; δ(entry.object, o_new)⟩ to levelCandidates
            } /* end if level */
          }
        }
      } /* for each entry */
    } /* for each can */
    if level = treeHeight − 1 {
      if targetLeaf is ∅ then return FindLeafSingleWay(o_new) else return targetLeaf
    }
    sort levelCandidates by .ObjectToParentDistance ASC
    let nodeCandidates = levelCandidates[0..v[level]−1]                  // pick the best node candidates
  } /* for each level */
}
```

3.3 Experimental Evaluation

We performed an extensive experimentation with the two new techniques and their combination. We compared them against the original M-tree dynamic construction and also against the previously proposed techniques including multi-way leaf selection and generalized slim-down algorithm. Only the distance computation costs are included in the experiments. Since the I/O costs

correlate with the computation costs, their inclusion would be redundant.

3.3.1 Databases

We have used two databases, a subset of the *CoPhIR* database [Falchi et al., 2008] of MPEG7 image features extracted from images downloaded from `flickr.com`, and a synthetic database of polygons. The CoPhIR subset consisted of 1,000,000 feature vectors formed by two MPEG7 features (12-dimensional color layout and 64-dimensional color structure, i.e., total 76 dimensions). As a distance function the Euclidean (L_2) distance has been employed. The *Polygons* database was a synthetic randomly generated set of 250,000 2D polygons, each polygon consisting of 5–15 vertices. The Polygons should serve as a non-vectorial analogy to uniformly distributed points. The first vertex of a polygon was generated at random. The next one was generated randomly, but the distance from the preceding vertex was limited to 10% of max. distance. We used the Hausdorff distance for measuring two polygons (where the order of vertices does not matter), so here a polygon could be interpreted as a cloud of points.

3.3.2 Experiment Settings

The query costs were always averaged over 200 query objects, while the queries followed the distribution of database objects. We did not perform an inter-MAM comparison; we focused just on various configurations of M-tree – with or without forced reinsertions under single-, multi-, or hybrid-way leaf selection. As the parameters we observed various data dimensionalities, database sizes, M-tree node capacities, hybrid-way branching factor, as well as various forced reinsertion settings. The M-tree node capacities ranged from 20 to 80, the index sizes took 1–138 MB, the M-tree heights were 2–5 (3–6 levels). The minimal M-tree node utilization was set to 20% of node capacity. On average, the methods utilizing forced reinsertions achieved 80% leaf utilization (87% in case of multi-way leaf selection), while the "non-reinserting" ones got to 70% (75% for multi-way selection). The index size is connected with the leaf utilization, so the forced reinsertions produced indexes smaller by 15%. Unless otherwise stated, the database size in experiments was set to 250,000 objects.

stage of insertion	*label*	*description*
leaf selection	Single	single-way leaf selection (default)
	Multi	multi-way LS
	Hybrid(b)	hybrid-way LS, b stands for the branching factor
	Hybrid(b).Nonfull	hybrid-way LS, restricted to select only non-full leaves, i.e., Hybrid(∞).Nonfull = Multi
node splitting	Classic	classic *mM_Rad* node splitting (default)
	Sampling	sampling *mM_Rad* (10% of node's entries in sample)
forced reinsertions	Full_RI	full FR, recursion depth = 10, removed entries = 4
	Cons_RI(x,y)	conservative FR, x is recursion depth, y is number of removed entries – if not specified, $x = 10, y = 4$
	Cons_RI(x,y).NoHistory	moving of entries is not affected by *SplitCount*
	(nothing specified)	FR not used (default)
generalized slim-down algorithm	GeneralizedSlimDown	generalized slim-down algorithm used on M-tree built using Single.Classic

Table 3.1: Description of labels in the figures' legends

Because of the many tested M-tree construction variants, we have formed a set of labels denoting certain alternatives within each stage of the insertion process (leaf selection, node splitting, forced reinsertions), see Table 1. A combination of labels belonging to each stage of construction constitutes a complete variant of insertion, these composed labels are used in the following figure legends. Within each stage a default value is marked, which applies in case that no of the respective stage's possibilities is specified in the composed label. Hence, the very original M-tree dynamic insertion methods [Ciaccia et al., 1997] are denoted as Single.Classic and Single.Sampling.

3.3.3 The Results

In the first experiment (see Figure 3.5) we have observed varying branching factor b applied to hybrid-way leaf selection. The greater the b, the better the query performance of Hybrid(b) variants but also the slower construction. For example, Hybrid(∞).Cons_RI is 35% faster in querying than Multi, but slower in construction in a similar proportion. Nevertheless, Hybrid(50).Cons_RI beats Multi and Multi.Cons_RI in both construction and query performance. The results for Multi.Cons_RI and Hybrid(∞).Cons_RI differ because the multi-way selects only non-full leaves, while hybrid-way selects also the full ones.

In the second experiment we have examined varying database dimensionalities, ranging from 8 to 64 (see Figure 3.6). We can observe that with increasing dimensionality the queries become less efficient – almost exponentially with the dimension. Hence, we experience the effects of dimensionality

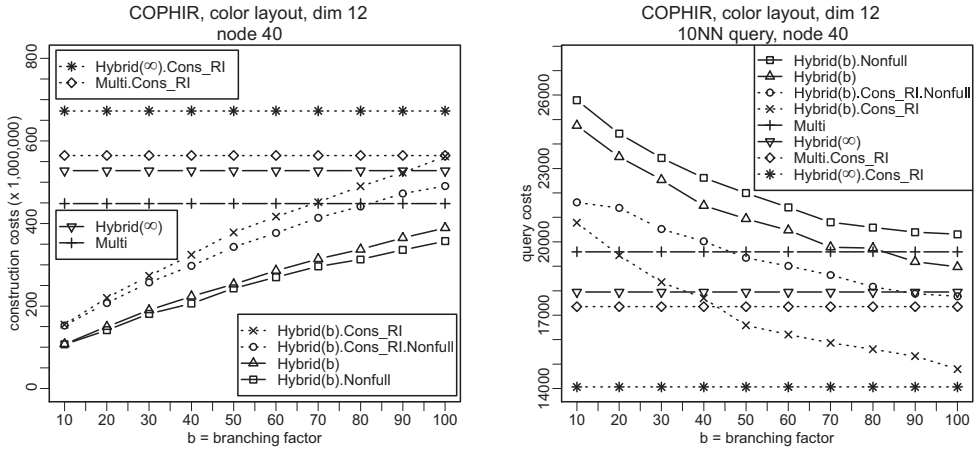


Figure 3.5: Hybrid-way branching factor: (a) Construction costs (b) 10NN query costs

curse. However, note the construction costs of all methods except GeneralizedSlimDown and Hybrid(∞).Cons_RI are almost constant. This observation is a nice evidence of the logarithmic construction complexity of hybrid-way leaf selection and forced reinsertions, which is further supported by the worse results of GeneralizedSlimDown and Hybrid(∞).Cons_RI (being super-logarithmic methods). Moreover, note that for dimension 64 the method Hybrid(10).Cons_RI is 1.3× slower in query processing than GeneralizedSlimDown and Hybrid(∞).Cons_RI, but 20× (10×, respectively) faster in construction.

COPHIR constructions costs, size 500,000, dim 12, node 40

Generalized SlimDown	Hybrid(∞) .Cons_RI	Hybrid(10) .Cons_RI	Classic	Classic .Cons_RI	Sampling .Cons_RI	Sampling
3,981,370,880	2,182,645,760	371,376,416	52,274,784	66,667,772	55,477,300	36,678,464

POLYGONS constructions costs, size 250,000, dim 30, node 40

Generalized SlimDown	Hybrid(10)	Hybrid(10) .Cons_RI	Classic	Classic .Cons_RI	Sampling .Cons_RI	Sampling
103,621,584	59,779,960	87,028,288	22,256,964	28,538,876	22,451,152	15,371,245

Table 3.2: Construction costs for Figure 3.7

The third experiment was focused on varying query selectivity – in Figure 3.7 see the results for kNN and range queries (for the construction costs see Table 2). Note that for Hybrid(∞).Cons_RI in case of COPHIR database the achieved query costs are exactly the same as for GeneralizedSlimDown, but the construction is almost 50% cheaper. Also note that in case of COPHIR

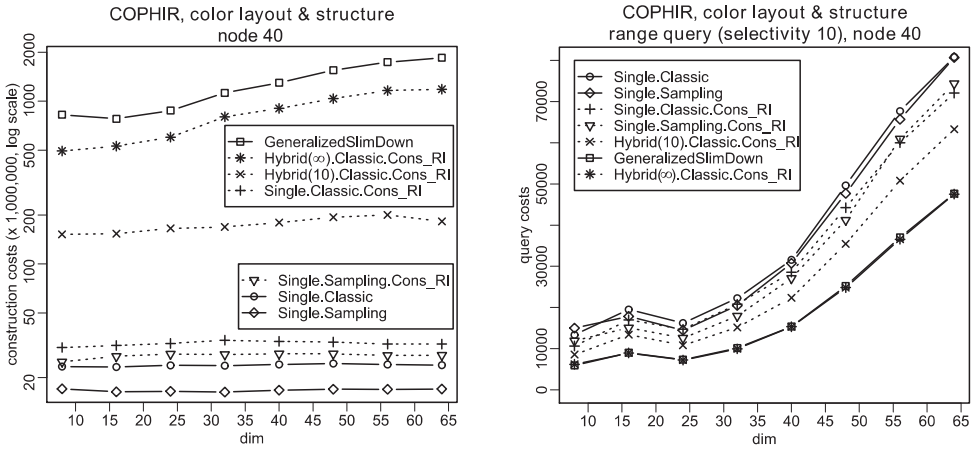


Figure 3.6: Varying dimensionality: (a) Construction costs (b) Range query costs

the Single.Sampling.Cons_RI is significantly faster in query processing than Single.Classic, while having almost the same construction costs.

In the fourth experiment we have observed the impact of database size on indexing (see Figure 3.8). In the upper part the construction/query costs for the Single.* variants are presented relatively to the baseline Classic method. We can see that the improvement over the baseline is quite stable with increasing database size in terms of construction costs, however, the query performance improves a bit faster with increasing database size. Also note the Single.Cons_RI(10,4) (proposed in the Section 3.1.2) clearly beats the Single.Full_RI (proposed in the Section 3.1.1) in both construction and query costs. In the bottom part of Figure 3.8 the construction/query costs are presented in absolute numbers but now for the Hybrid.*, *.Sampling.* and GeneralizedSlimDown variants.

The fifth experiment (Figure 3.9) inspected the impact of the maximal number of reinserted entries (ground entries removed onto the stack, respectively) on the conservative forced reinsertion strategy. We considered various node capacities (20–60), while the results show that a reasonable value is around 3 or 4. Higher values slightly increase construction costs but do not bring a clear improvement in querying (there is cca 0.5% variance in query costs).

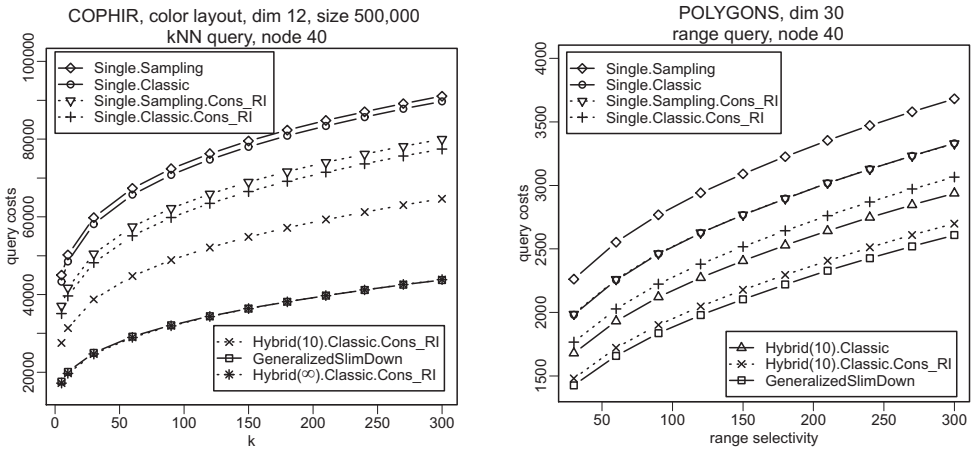


Figure 3.7: Varying query selectivity: (a) kNN queries (b) range queries

costs	Hybrid(∞) .Cons_RI	Multi	Classic .Cons_RI	Sampling .Cons_RI	Classic	Sampling
construction	3,535,264,768	2,990,771,968	106,378,144	93,059,856	74,684,496	57,775,288
10NN query	31,132	39,540	81,019	80,749	94,081	102,898

Table 3.3: Results for COPHIR, one million objects, dim 12, node size 20

In Table 3 see the results of the sixth experiment, considering the largest COPHIR database in the testbed – one million objects, indexed within 6-level M-trees, node capacity 20. We can observe the queries on Hybrid(∞).Cons_RI performed 3× faster than those on Classic, however, for 47× higher construction costs. Nevertheless, the Sampling.Cons_RI achieved 15% reduction in query costs with respect to Classic for just 125% of Classic's construction costs.

In the following experiment we have tested the impact of various M-tree node capacities, see Figure 3.10. An interesting observation is that the expensive techniques improve the construction costs with growing node capacity, while, on the other hand, the less expensive techniques slightly improve the query performance with growing node capacity.

The Figure 3.11 is maybe the most important outcome of the experimental results. Here the graphs from Figure 3.10 are aggregated into one, in order to show the construction vs. query performance trade-off. The closer to the bottom-left origin a technique is, the better the overall performance trade-off is. Hence, we can see the Single.Sampling.Cons_RI is clearly better than Single.Classic, and that Hybrid(∞).Classic.Cons_RI is much better than the

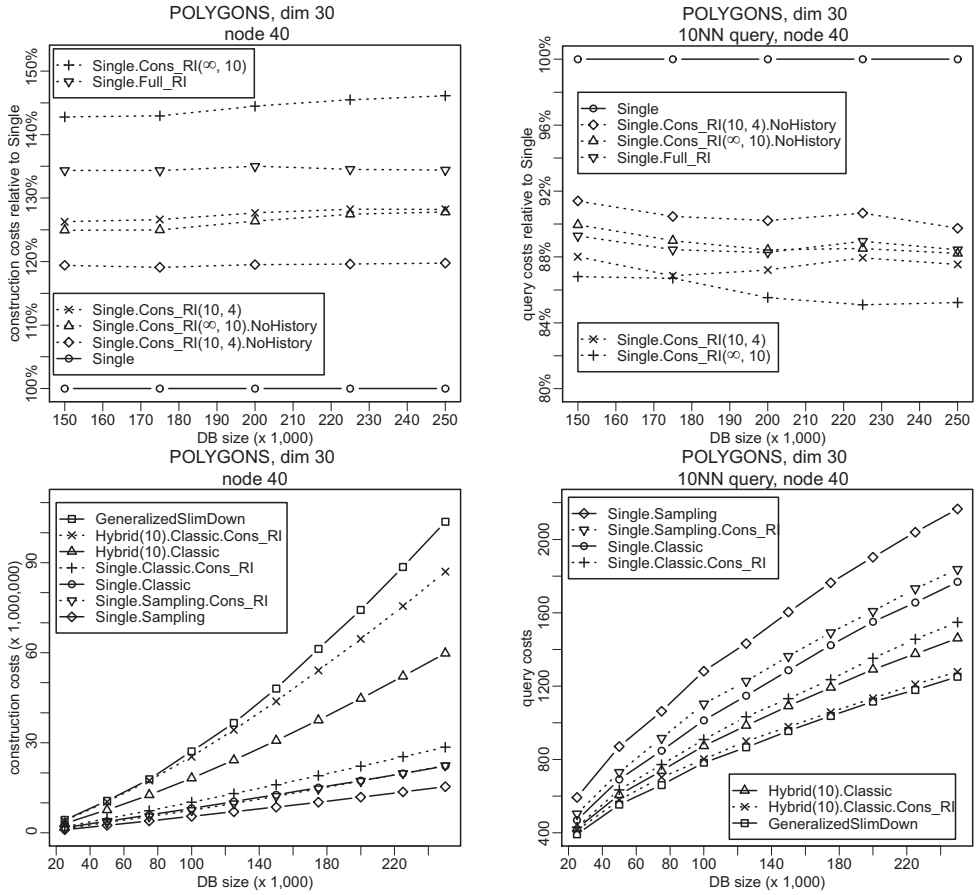


Figure 3.8: Varying database size: (a) Construction costs (b) 10NN query costs

GeneralizedSlimDown. All the other techniques lie on a sort of skyline, hence, they represent meaningful trade-off choices applicable to various scenarios.

In the last experiment we have tested the modification of the reinserting algorithm guaranteeing average leaf node utilization (ALNU). The results are presented in the Table 3.4. We may observe, that the requested average leaf node utilization corresponds with the real average leaf node utilization. It is also obvious that "multi-way" reinsertions, used to increase leaf node utilization, make the indexing more expensive and lead to more compact M-tree hierarchies.

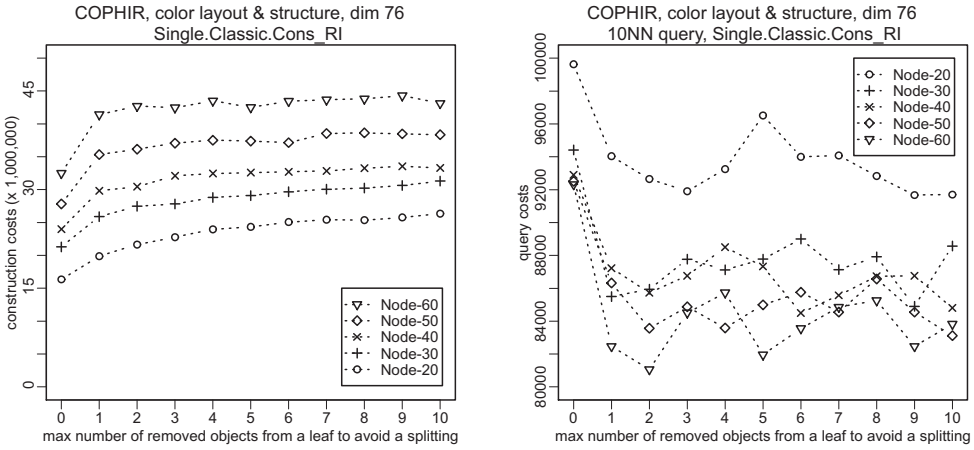


Figure 3.9: Max. number of removed objs: (a) Construction costs (b) 10NN query costs

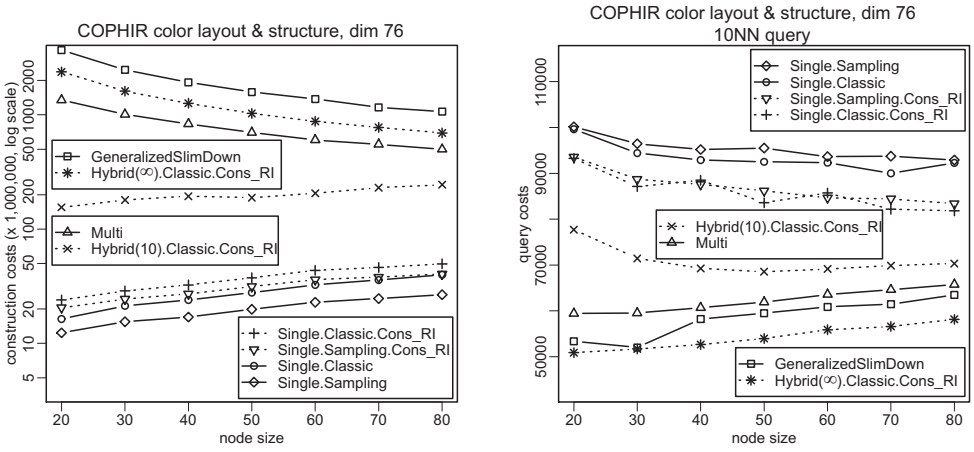


Figure 3.10: Varying node capacity (a) Construction costs (b) 10NN query costs

3.3.4 Summary

The conservative forced reinsertions proved their usability when combined with "either-way" leaf selection. The query performance is always higher than that of techniques without forced reinsertions, while the construction is usually a few tens of percent more expensive. The conservative forced reinsertions also showed better results in both querying and construction when

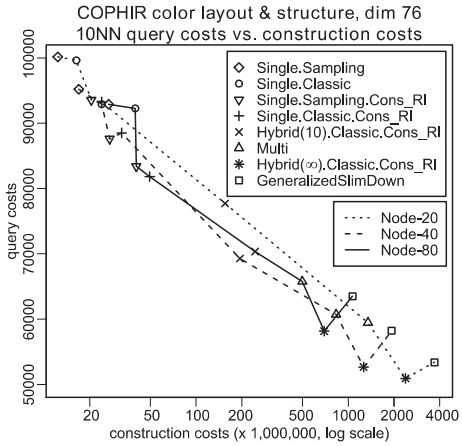


Figure 3.11: Construction vs. query costs – aggregate results

requested ALNU	real ALNU	construction costs	query costs
65.0%	65.9%	7 569 751	43 357
70.0%	70.0%	30 661 166	42 771
75.0%	75.1%	48 093 457	41 910
80.0%	80.0%	65 098 265	40 880
85.0%	85.0%	86 540 299	40 560
90.0%	89.9%	110 449 493	39 730

Table 3.4: Indexing based on average leaf node utilization

compared to the full forced reinsertions (as proposed in [Lokoč and Skopal, 2008]). When used with single-way leaf selection, the conservative reinsertions are suitable also for indexing large and high-dimensional databases, where the feasibility of index construction is the crucial task. The hybrid-way leaf selection is beneficial for its scalability, while when used with unlimited branching factor and conservative forced reinsertions, it becomes a clear winner over the generalized slim-down algorithm, being much cheaper in construction and comparable or better in query costs.

3.4 Discussion

In this chapter we have proposed two new techniques improving dynamic insertions in M-tree – the forced reinsertions and the hybrid-way leaf selection.

Both of the techniques preserve logarithmic complexity of a single insertion, while they aim to produce more compact M-tree hierarchies. The proposed techniques experimentally proved their benefits. The experiments have also shown the problem of constructing compact M-trees cannot be solved by a simple solution or by a brute force. The increasing complexity of M-tree-related techniques developed over the last decade indicates it is worth to continue in designing even more complex algorithms within the realm of M-tree. Moreover, the techniques can be simply adopted by other members of the M-tree family, especially by the PM-tree.

Chapter 4

Parallel Dynamic Batch Loading

Although metric access methods (see section 1.5) proved their capabilities when performing efficient similarity search, their further performance improvement is needed due to extreme growth of data volumes. Since multi-core processors become widely available, it is justified to employ parallelism for indexing. However, taking into account the Gustafson's law [Gustafson, 1988], it is necessary to find tasks suitable for parallelization. Such a task could be M-tree construction. Unfortunately, parallelism during an object insertion in hierarchical index structures is limited by a node capacity. It is much less restrictive to run several independent insertions in parallel. However, synchronization problems occur whenever a node is about to split. In this chapter we present our new technique of M-tree construction. The technique postpones splitting of overfull nodes and thus allows simple parallelization of M-tree construction. Our experiments confirm the new technique guarantees significant speedup of M-tree construction and also improves the quality of the index.

4.1 Parallel Processing in MAMs

New approaches like distributed computing or parallelism have been successfully implanted to MAMs. Let us remember several approaches based on hierarchical metric structures.

In [Batko et al., 2005], distributed index GHT* was presented, consisting

of nodes located on multiple computers. The authors use a structure called address search tree (AST) to minimize the network traffic and to prune non-relevant data. The AST is distributed over all nodes and used to find nodes where requested data may be located or new objects should be inserted. When a local node is split, the AST is modified, such that only necessary changes of the AST are distributed to other nodes. A local node can utilize an arbitrary MAM (e.g., LAESA, M-tree, PM-tree, etc.) to store data. Several different distributed indexes have been developed since GHT* introduction. For their comparison, see [Batko et al., 2006].

Since multi-core architectures have become standard, parallelization on a local machine can be employed, resulting in significant performance boost. In [Zezula et al., 1998] authors provide simple extensions of basic M-tree algorithms (inserting and querying) employing parallelism and discuss their limitations. The authors propose a pre-fetch strategy selecting data from different disks and propose four de-clustering algorithms allocating data to disks. For CPU parallelism the authors provide algorithms using just single parallel distance evaluation in one node, since the algorithms have to decide which node should be visited consecutively. However, parallel processing of a node is limited by its capacity m, so that any further speedup cannot be achieved using more than m cores (Amdahl's law [Amdahl, 1967]).

4.2 Parallel Opportunities in the M-tree Construction

The original M-tree construction consists of subtasks for which a parallel implementation is limited. During a leaf node selection, a node cannot be processed until the routing item of its parent node is selected. Therefore, a parallelism during a leaf node selection is restricted just to node processing. This is the reason why we prefer parallel batch loading utilizing multiple concurrent leaf node selections. Parallelism during a node splitting is less limited – distance matrix evaluation and promotion is limited by one half of square of a node capacity m, and thus much more cores can be used.

In the following text, 'task' denotes a procedure which can be assigned to a physical thread. In this section we review node access synchronization (necessary for our new method), parallel distance matrix evaluation and parallel promotion of new routing entries.

4.2.1 Node Locking

In the M-tree, access to a node must be synchronized to guarantee correct results. In case a task only reads a node (e.g., querying), it is suitable to utilize shared locks. Thus, more than one task can access the node for reading. However, in the case a task modifies a node, the task has to exclusively lock the node, so that no task can access the node until the exclusive lock is released. In some cases it is advantageous to exclusively lock just an entry in a node, thus other tasks can access the rest of the node.

4.2.2 Parallel Distance Matrix Evaluation

To promote two new routing entries during a node splitting, distances between all objects have to be evaluated and stored as a distance matrix[1]. The computation of a distance matrix can be effectively parallelized. Each cell of a distance matrix determines two objects which distance has to be evaluated. Our algorithm divides the matrix to distinct parts (sets of cells) and distributes them to tasks. Each task determines two objects from an assigned cell, evaluates their distance and stores the result in the cell. Hence, no synchronization is needed during execution since each task works in its unique part of the matrix.

To achieve the maximal speedup, it is usually advantageous to divide the matrix to nontrivial (more than one cell) parts of the same size[2]. Since the matrix is symmetric with nulls on the main diagonal, only the upper triangle has to be evaluated. It might be difficult to divide the triangle to arbitrary number of parts of the same size. Therefore, we utilize the one-dimensional interval $[0, l)$ for partitioning of the upper triangle. The interval's length l is set to the number of cells in the upper triangle – l can be computed using the node capacity m ($l = m \times (m - 1)/2$). Then, the interval is split to a desired number of parts of the same size. Finally, a mapping is used to assign values from the interval to cells from the upper triangle. And thus the perfect partitioning of the upper triangle is achieved. An example of a mapping is illustrated in Figure 4.1.

[1] We do not consider SAMPLING heuristics (see section 2.3.2).

[2] In cases where distance evaluation costs are nearly the same for each pair of objects.

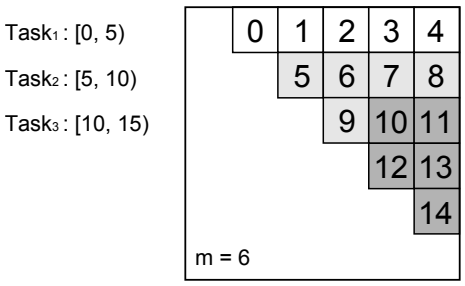


Figure 4.1: Partitioning of the upper triangle among three tasks

4.2.3 Parallel Promotion

After a distance matrix is evaluated, all pairs of routing entries are tested against some criterion and the most fitting pair is promoted. For each tested pair there exists a corresponding cell in the 'upper triangle'. Effective parallelization of promotion is also based on the partitioning of the upper triangle using the one-dimensional interval. Each promotion test has assigned a value v from the interval. Thus, it is possible to effectively distribute promotion tests to several tasks and run them in parallel. After all tasks finish their local promotions, the global best pair of candidates is selected. If more than one best pair exists, the pair with the lowest v is promoted, so the promotion is deterministic, i.e., produces the same results regardless of parallel or serial[3] processing.

4.3 Parallel Dynamic Batch Loading

The dynamic batch loading introduced in [Chen et al., 1998] is a technique used for faster R-tree [Guttman, 1984] construction. Several new objects are organized to small trees and then appended to the R-tree as whole sub trees. The method is called STLT (*small-tree-large-tree*). The indexing using dynamic batch loading is much faster and thus it is possible to upload larger data collections runtime. However, this kind of dynamic batch loading leads to a deterioration of the index quality, that is, more overlaps occur between index regions. In this section, we introduce a completely different approach of batch loading suitable for MAMs. The approach is based on parallelism and improves the quality of an index. Moreover, the approach is not limited

[3] "Serial" stands for the single-thread nonparallel processing.

by a node capacity, and is designed for expensive distance functions. In the following text we describe dynamic parallel batch loading in the M-tree.

The basic idea is simple. Batches of objects are inserted to the M-tree in consecutive iterations, where each iteration consists of three basic steps. First, a limited number of new objects is collected in a set. Then, as the second step, objects from the set are inserted in parallel to the M-tree. However, there have to be restrictions imposed to obtain correct M-tree hierarchy. In particular, traditional inserting and split handling cannot be used during parallel insertions. Not all objects have to be inserted to the M-tree during the step two. Moreover, some objects may be removed from the M-tree and inserted later. In the third step, objects, not inserted or removed during the step two, are handled. Some of them are inserted to the M-tree in the traditional manner. The remaining objects are processed during the second step of the next iteration. All steps are discussed in the following paragraphs.

4.3.1 Iteration Steps

Gathering

New objects are collected in the set until their number reaches a user-defined value (size of the batch). Since users may perform queries during the first step, algorithms evaluating queries have to search also in the set used for gathering. To avoid a sequential scan in the set for large size of the batch, the LAESA or another simple MAM can be employed. However, a simple list can be utilized for small size of the batch (see Figure 4.2a).

Parallel Inserting

After a desired number of objects is accumulated, the iteration switches to the second step (see Figure 4.2b). Objects are distributed to several tasks which try to insert them into leaf nodes. Here, a synchronization must be utilized in a case when multiple tasks concurrently access the same leaf node.

An insertion of a new object by one task consists of two steps. The first step is a leaf node selection. During this step, other tasks can access the same node to determine the best fitting routing entry. Only shared node locks are applied to maximize the parallel throughput during the routing entry selection. However, before the best fitting routing entry is updated (in case its radius has to be enlarged), the routing entry has to by exclusively

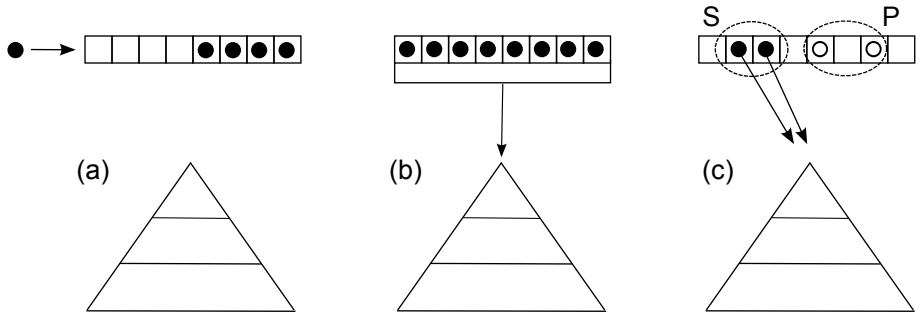


Figure 4.2: (a) Gathering, (b) parallel inserting and (c) traditional inserting.

locked. In the second step of the insertion, the selected leaf node is modified and thus the exclusive lock is necessary. All other tasks requiring the node are blocked until the first task releases the lock. Since each task exclusively locks just one node at once, no deadlocks are possible.

During parallel inserting, the algorithm postpones splits to reduce synchronization. Otherwise, exclusive locks would be necessary for changes in the parent node and also in the grand-parent node. Thus, split postponement increases throughput of parallel insertions. Moreover, the quality of the index can be improved analogously as in section 3.1 (the reinsertions). To avoid the split, we use a similar heuristic – if a leaf node is about to split, the most distant ground entry (from the parent routing entry) is removed from the leaf node and stored in the temporary memory list. Since the removed objects are not inserted immediately (though later during the next step or even the next iteration), we denote this technique as *postponed reinsertions*. Note that all objects, not stored in the second step, have been removed from an overfull leaf node. Let us denote these objects as *RO* (remaining objects).

Traditional Inserting

Although the parallel inserting during the second step is very scalable, the splits cannot be postponed interminably. Sooner or later, the filled leaf nodes will become overfull and thus we need to "create" space for new objects by allowing several splits. Hence, several objects from the temporary memory list have to be inserted in the traditional (non-scalable) manner. Moreover, we would like to select such objects the insertion of which will cause the split.

In the third step of the iteration, remaining objects from *RO* are divided

into two sets P and S. Objects from the first set P will be inserted in parallel during the second step of the next iteration. Objects from the second set S are inserted to the M-tree in the traditional manner (see Figure 4.2c, where four not inserted or removed objects are divided into two groups S and P). Since a lot of splits is supposed to occur, the third step could be denoted as a split generating or tree reorganizing step. The objects from S are inserted one by one, and thus no synchronization during splitting is needed. Moreover, as mentioned in section 4, split can be effectively parallelized. Hence, splits-causing serial insertions do not slow the parallel M-tree construction much. Parallelism during a node processing can be also employed to speedup traditional object insertion. To divide objects from RO into two sets we propose two heuristics – *random* and *minDistance*.

- The *random* heuristic is very simple – it arbitrarily moves some objects (e.g. 10%) from RO to S. The remaining objects are moved to P.
- The *minDistance* heuristic tries to select objects that will probably cause split. The heuristic divides objects from RO into groups. Each group G_N consists of objects that were assigned to the same leaf node N during the previous step. For each object from G_N the distance to the parent object o_r (from the routing entry pointing to N) was evaluated during the previous step and can be utilized. From each group G_{N_i}, an object o_i with the lowest $\delta(o_i, o_{r_i})$ is moved to S. The object is supposed to be probably re-inserted to the same leaf node and thus can cause the split. Remaining objects are moved to P. An example of the *minDistance* heuristic is depicted in Figure 4.3, where filled objects outside the two balls are moved to S and the remaining objects (marked by ellipses) are moved to P.

4.3.2 Scalability Notes

To estimate scalability of our new method, three parts of our algorithm have to be analyzed – the parallel batch inserting, the traditional single-way inserting and the splitting of a node (the last two also limit the original parallel algorithm [Zezula et al., 1998]). Parallelism during the splitting is limited quadratically with the node size – up to $m \times (m-1)$ cores can be used to speed-up the splitting. As mentioned before, parallelism during the single-way leaf selection is limited by the node capacity m. Scalability of

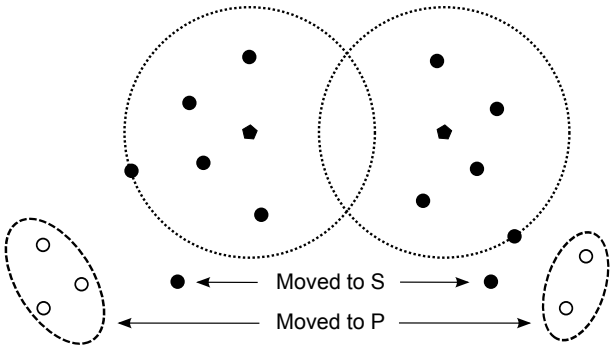


Figure 4.3: The *minDistance* heuristic for objects from *RO* (outside of the two balls).

the parallel batch inserting is limited by the batch size. However, there is another limiting factor for the batch size. There are more objects re-inserted for a bigger size of the batch, because more leaf nodes become overfull[4] and also more objects become assigned to the same leaf node. Thus, most of the processed objects are inserted again in the next iteration. Moreover, the behavior probably depends also on the node size and on the number of leaf nodes which makes the estimation more complicated. Therefore, we have to prove that the parallel batch loading (which is not directly limited by the node size) is more scalable than the original method. A model for the scalability estimation is one of the topics of our future research.

To provide clear summary, in Listing 5 see the pseudo code of dynamic batch loading algorithm.

Listing 5. (parallel dynamic batch loading)

```
let
A, P and S be sets of objects
let RO be a thread-safe set of objects
let Insert(obj, taskCount) be a traditional insertion
let T_i be a task representing method InsertWithoutSplit()

method BatchInsert(o_new, batchSize, tasksCount, doublePI) {
  insert o_new into A

  if A.Count is lower than batchSize then
    return

  split A to taskCount subsets A_i
```

[4]Leaf nodes are just filled with new objects during the parallel batch inserting step

```
  for (i = 1; i ⩽ taskCount; i++)
    assign A_i to T_i

  //parallel batch inserting with suppressed splitting
  run tasks T_i in parallel

  split RO to S and P

  for each obj in S
    //traditional inserting, splits may occur
    Insert(obj, taskCount)

  move objects from P to A
}

method InsertWithoutSplit() {
  for each obj in A_i
    let leafNode be a leaf node found for obj using single-way strategy

    exclusively lock leafNode

    insert obj to leafNode
    if leafNode is overfull
      let remObj be the furthest object in leafNode
      insert remObj to RO

    release exclusive lock from leafNode
}
```

4.4 Experimental Evaluation

In our experiments we have focused on the time spent during the M-tree construction. The parallel batch loading algorithm consists of several significant parts – parallel batch inserting (PBI), traditional inserting causing split (ICS), traditional inserting not causing split (INCS) and remaining operations (Residue)[5]. Each part has been measured separately, thus parallelism bottlenecks can be highlighted. All parts together form an M-tree construction time, while all presented times are in seconds. We do not present the time spent by the Residue part since the time is insignificant. The reader can simply derive the time from the overall time and remaining presented times. The batch size used by the batch loading algorithm was set in all tests to 200 (for higher values there was no speed improvement). Since parallelism leads to a nondeterministic M-tree construction, we have built M-tree five

[5]Residue includes reading new objects from a source file, writing nodes to the disk and secondary CPU costs.

times for parallel batch loading methods using more than one core. To minimize the construction time for tests parametrized with more cores we have employed the parallel node splitting and the parallel routing item selection during the traditional inserting. We have also performed several query tests to check the index quality, i.e., the number of distance computations spent during query processing. Each query test consisted of 200 randomly selected query objects from a database.

We have performed our tests on 64bit Windows Vista and using processor Intel Core 2 Quad Q9550 2.83GHz. We have also performed several tests on the linux platform, using 16 core cluster, each CPU 2.83Ghz, to detect the "real" speedup potential of our method. We have used our own C++ M-tree implementation compiled for x64 processors, utilizing Intel Threading Building Blocks 2.1.

4.4.1 Databases

We have used the same two databases as in the previous chapter (see Section 3.3.1). As a distance measure for CoPhIR [Falchi et al., 2008] database, the Lp metric has been employed. We have arbitrarily set p to 5.123456 to simulate an expensive distance function[6]. Note that since we are concerned with database tasks, the particular semantics of a similarity is not relevant for our experiments. For polygons database, we have used the Hausdorff distance.

4.4.2 M-tree Settings

For both databases we have selected two node sizes determining inner and leaf node capacities, summarized in Table 4.1. The minimal M-tree node utilization was set to 20% of node capacity, as a promotion criterion was used the mM_Rad choice[7].

4.4.3 S Generating Heuristics

In the first set of tests we have created two variants of M-tree using the parallel batch loading method and four cores. We have focused on heuristics

[6]The fractional power is an expensive operation.

[7]Such pair of candidate routing objects is chosen, which has the smallest radius of the greater region.

database	node size	node cap.	height
POLYGONS	4kB	30/28	4
POLYGONS	6kB	46/42	3
CoPhIR	8kB	25/24	5
CoPhIR	12kB	38/37	4

Table 4.1: M-tree configurations and corresponding reached height

that fill S with objects from RO. We have tested our proposed heuristics – *random* and *minDistance* (denoted as RAND and minD). The amount of randomly selected objects from RO in *random* heuristic was set to 10%. In Table 4.2, overall time of M-tree construction (CTime), the number of inserted batches (iterations), insertions causing split (ICS) and not causing split (INCS), and the number of objects moved from RO to P (RO2P) are presented. The *minDistance* heuristic results in the less number of objects moved to P, i.e., less number of iterations (= less time).

POLYGONS, node size 4kB					
heuristic	CTime	iterations	ICS	INCS	RO2P
RAND	30.6	1 371	12 802	15 146	24 271
minD	29.7	1 279	12 893	15 029	5 028
CoPhIR, node size 12kB					
heuristic	CTime	iterations	ICS	INCS	RO2P
RAND	274.9	5 254	44 782	60 357	50 877
minD	266.6	5 139	45 029	57 019	24 291

Table 4.2: Number of serial insertions causing and not causing split

The variant minD reached the lowest times in all tests as presented in Table 4.2, and thus it has been used in the rest of our experiments.

4.4.4 Parallel Batch Loading Speedup

The speedup is an important feature of parallel algorithms. Our algorithm consists of three parts employing parallelism and one serial part (Residue). Construction times spent by parallel parts are presented in Table 4.4. We also present overall time which is the sum of all parts. We may observe that the parallel bulk inserting and split generating insertions consume most of

POLYGONS, node size 6kB				
heuristic	CTime	PBI time	ICS time	INCS time
RAND	35.8	14.2	14.7	1.2
minD	35.5	13.5	15.1	1.2
CoPhIR, node size 8kB				
heuristic	CTime	PBI time	ICS time	INCS time
RAND	256.2	155.6	62.3	10.7
minD	222.4	121.9	62.5	10.3

Table 4.3: Time spent during parallel insertions and insertions (not) causing split

the overall time. The speedup of the parallel batch inserting (PBI) using four cores is greater than 3.47 in all tests. The speedup of insertions causing split (ICS) depends on a node size – lower node size leads to lower speedup, but also to lower impact on the overall time. The overall speedup is greater than 2.95 in all tests and reached 3.33 for CoPhIR and node size 12 288.

4.4.5 Comparison with the Original Algorithm

We also show the comparison with the original M-tree building method (denoted as CLASSIC_numberOfCores) utilizing parallel splitting, promotion and node processing (routing item selection). Our new technique (denoted as BATCH_numberOfCores) also utilizes parallel splitting, promotion and node processing for the traditional inserting. Table 4.5 summarizes overall index construction times. As you can observe, our new technique using one core does not slow down construction time significantly (in comparison with the original M-tree method) and guarantees significant speedup. Moreover, for four cores our method is faster in all cases.

We have also checked quality of all the created indexes. We have used range queries with radii set to 75 for CoPhIR database and 15 for POLYGONS. The average cardinality of the answer was 74 objects for CoPhIR and 47 objects for POLYGONS. As we can see from the results presented in Table 4.6, the quality of an index built by our parallel batch loading technique slightly increases.

Finally, we have also performed several tests on the linux platform, using 16 core cluster, to detect the "real" speedup potential of our method. We have indexed 1 million 76-dimensional vectors from the CoPhIR database

POLYGONS, node size 4kB					
proc. cores	PBI time	ICS time	INCS time	CTime	speedup
1	46.4	33.2	2.7	88.1	1.00
2	23.9	17.6	1.8	49.2	1.79
4	12.6	10.2	1.2	29.8	2.95
POLYGONS, node size 6kB					
proc. cores	PBI time	ICS time	INCS time	CTime	speedup
1	49.0	53.0	2.5	110.9	1.00
2	25.9	28.3	1.6	61.5	1.80
4	14.1	15.1	1.1	36.1	3.07
CoPhIR, node size 8kB					
proc. cores	PBI time	ICS time	INCS time	CTime	speedup
1	452.4	194.3	29.6	703.3	1.00
2	227.7	106.3	16.0	377.1	1.87
4	124.4	62.7	10.9	225.0	3.13
CoPhIR, node size 12kB					
proc. cores	PBI time	ICS time	INCS time	CTime	speedup
1	537.2	284.2	29.8	877.8	1.00
2	281.4	148.9	16.8	474.9	1.84
4	143.5	83.2	9.4	263.6	3.33

Table 4.4: Time of parallel batch insert construction using 1, 2 and 4 cores

using both methods, and compared results in the table 4.7. See that although BATCH_1 is slower than CLASSIC_1 when using single core, the scalability of the BATCH method brings BATCH_16 is nearly 2x faster than CLASSIC_16.

The scalability of the BATCH method is determined by the very scalable second step of a parallel batch loading iteration. See column PBI time in the table 4.8, where originally dominant time 675 seconds is reduced just to 48 seconds. We may also observe, that traditional insertions causing and not causing split cannot efficiently use more than 8 cores (for this experiment), which is the reason, why the CLASSIC_16 (using just ICS and INCS) lags behind the BATCH_16.

	POLYGONS		CoPhIR	
node size	4kB	6kB	8kB	12kB
CLASSIC_1	83.4	111.6	648.7	837.5
CLASSIC_2	54.2	70.2	374.8	476.4
CLASSIC_4	35.6	43.1	246.6	302.8
BATCH_1	88.1	110.9	703.3	877.8
BATCH_2	49.2	61.5	377.1	474.9
BATCH_4	29.8	36.1	225.0	263.6

Table 4.5: Index construction time

	POLYGONS		CoPhIR	
node size	4kB	6kB	8kB	12kB
CLASSIC	2 013	2 035	314 074	293 110
BATCH_1	1 900	1 869	303 833	275 848
BATCH_2	1 958	1 819	297 074	285 677
BATCH_4	1 935	1 837	303 490	276 090

Table 4.6: Number of distance computations spent by range queries

4.5 Discussion

In this chapter, we have proposed a new M-tree building method which utilizes the power of multiple processor cores and the re-inserting idea. This approach provides significant speedup of M-tree construction and more compact M-tree hierarchies, resulting in better query performance. The approach is not limited by a node capacity and thus can be applied in multiprocessor architectures with quite high number of cores. This allows fast dynamic index construction of very large data collections with expensive similarity function even on a single machine.

	CTime	speedup
CLASSIC_1	938	1.00
CLASSIC_16	179	5.24
BATCH_1	1013	1.00
BATCH_2	594	1.71
BATCH_4	304	3.33
BATCH_8	157	6.45
BATCH_16	104	9.74

Table 4.7: Index construction time and speedup

	PBI time	ICS time	INCS time
BATCH_1	675	284	46
BATCH_2	383	174	29
BATCH_4	186	93	17
BATCH_8	91	48	11
BATCH_16	48	39	10

Table 4.8: Index construction time in detail

Part II

Beyond the Metric Space Model

Chapter 5

Nonmetric Similarity Search

The contribution of this part of the thesis is a follow-up of indexing based on the TriGen algorithm, a previously proposed approach to efficient nonmetric similarity search.

5.1 Motivation for Nonmetric Search

Nowadays, the research in many scientific domains leans to digitization of physical phenomena and data processing. A perfect example here is the shift from molecular biology to bioinformatics, where a part of the biological research moved from "wet lab" to computerized analysis of protein/DNA models [Bourne and Weissig, 2003]. Hence, there are continuously emerging new extremely complex data types which cannot be easily modeled in "flat" Euclidean or L_p spaces. Simultaneously, the metric similarity functions became way too restrictive for the domain experts, as they do not allow robust and local behavior. Thus, the long-ago more or less theoretical objections against the metric postulates introduced by various psychological theories appear as strong arguments in many real-world applications nowadays.

In particular, the theories have refuted non-negativity ($\delta(x, y) \geqslant 0$) and identity ($\delta(x, y) = 0 \Leftrightarrow x = y$) by claiming that different objects could be differently self-similar [Tversky, 1977], [Krumhansl, 1978]. For instance, in Figure 5.1a, the image of a leaf on a trunk can be viewed as positively self-dissimilar if we consider a similarity which measures the less similar parts of the objects (here the trunk and the leaf). Or, alternatively, in Figure 5.1b the leaf-on-trunk and leaf could be treated as identical if we consider the most

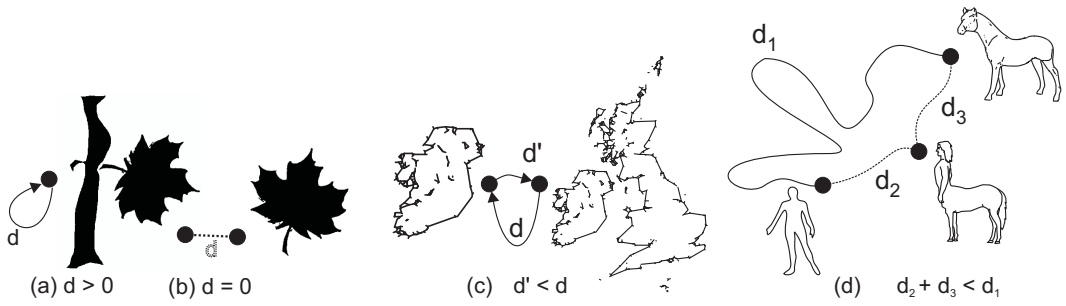


Figure 5.1: Arguments against metric properties (a) reflexivity (b) nonnegativity (c) symmetry (d) triangle inequality.

similar parts of the objects (the leaves). The symmetry ($\delta(x, y) = \delta(y, x)$) was questioned by showing that a prototypical object can be less similar to an indistinct one than vice versa [Rothkopf, 1957], [Rosch, 1975]. In Figure 5.1c, the more prototypical "Great Britain and Ireland" image is more distant to the "Ireland alone" image than vice versa (i.e., a subset is included in its superset but not vice versa). The triangle inequality ($\delta(x, y) + \delta(y, z) \geqslant \delta(x, z)$) is the most attacked property. Some theories point out that similarity has not to be transitive [Tversky and Gati, 1982], [Ashby and Perrin, 1988] as shown by the well-known example: a man is similar to a centaur, the centaur is similar to a horse, but the man is completely dis-similar to the horse (see Figure 5.1d).

In real-world applications, the lack of metric postulates could increase the richness of similarity modeling. In particular, a similarity measure that is nonmetric allows to model the following crucial properties:

- *Robustness.* A robust function is resistant to outliers (noise or deformed objects), that would otherwise distort the similarity distribution within a given set of objects [Donahue et al., 1996], [Howarth and Ruger, 2005]. In general, having objects x and y and a robust function δ, then an extreme change in a small part of x's descriptor should not imply an extreme change of $\delta(x, y)$.

- *Locality.* A locally sensitive function is able to ignore some portions of the compared objects. As illustrated in Figure 5.1a,b, we could model a "smart" similarity function that decides which portions of object x are relevant when evaluating its similarity with object y. This property leads not only to potential violation of non-negativity, but also to the

violation of triangle inequality. For example, consider the centaur and the man (see Figure 5.1d); here we perform such a locally sensitive matching - we compare either the human-like or horse-like parts of the two images. The locality is usually used to privilege similarity before dissimilarity, hence, we rather search for similar parts in two objects than for dissimilar parts [Smith and Waterman, 1981], [Robinson et al., 2000]. As in real world, highly similar objects are not very common when compared at a global scale. An "augmentation" of similarity by locally sensitive functions provides a way how to distinctly separate similar and dissimilar objects.

In the following paragraphs, we will recall several popular nonmetric distances used to model robust dissimilarity measures. Examples of nonmetric distances and their combinations are depicted in Figure 5.2a-d.

Fractional L_p Distances

The fractional L_p distances [Aggarwal et al., 2001] extend the Minkowski metrics (see Section 1.4.1) for $0 < p < 1$, however, such L_p distances are only semimetrics (see Figure 1.4). The fractional L_p distances are employed in such cases, where it is desirable to inhibit extreme differences in coordinate values. The fractional L_p distances have been employed for robust image matching [Donahue et al., 1996] and retrieval [Howarth and Ruger, 2005].

Cosine Measure

The cosine measure, defined as

$$\sigma_{cos}(x, y) = \frac{\sum_{i=1}^{n} x_i, y_i}{\sqrt{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2}}$$

is very popular in the vector model of text retrieval [Baeza-Yates and Ribeiro-Neto, 1999]. To obtain an equivalent semimetric distance, a simple transformation $\delta_{cos} = 1 - \sigma_{cos}$ can be used.

K-median Distances

The k-median distances, measuring the kth most similar portions of the compared objects, are usually employed to provide the robust behavior. The

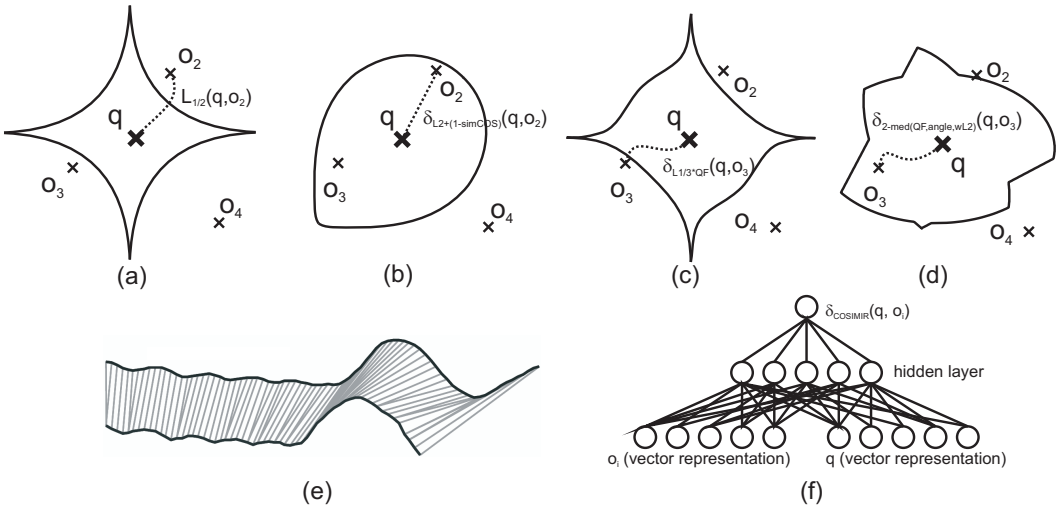


Figure 5.2: (a-d) Region balls of some nonmetric distances, (e) DTW distance and (f) COSIMIR model.

k-median distances can be utilized in cases, where a requested distance measure δ employs partial distances $d_i(.,.)$ considering ith portions of compared objects. The k-median distances can be used for example in connection with the Hausdorff distance in order to measure the kth most similar elements of two compared sets [Huttenlocher et al., 1993].

Dynamic Time Warping Distance

The dynamic time warping distance (DTW) is very popular in the area of sequence matching, because it is quite resistant to the sampling frequency or a time shift (see Figure 5.2e). The DTW is based on the minimization of the sum of distances between aligned elements, which can be efficiently computed by dynamic programming. The DTW has been used in time series retrieval [Yi et al., 1998] and even in shape retrieval [Bartolini et al., 2005]. To eliminate some undesired alignments, there were also developed various constraints on DTW [Keogh and Ratanamahatana, 2005].

COSIMIR Model

An employed distance measure can be also a nontrivial and complex algorithm, where a "manual" enforcement of metric properties is nearly impossi-

ble. The COSIMIR model [Mandl, 1998] is based on a three-layer backpropagation neural network, that can be trained to model an arbitrary user-defined similarity measure (see Figure 5.2f). However, in such general model, it is very hard to train a metric distance.

5.2 Similarity Search in Nonmetric Spaces

In this section, we will review several basic approaches for nonmetric similarity search. For more details, see a survey [Skopal and Bustos, 2010] summarizing the motivation and the state-of-the-art techniques for efficient nonmetric search. The techniques, so called *nonmetric access methods* (or shortly NAMs), can be divided into three groups, as follows:

- *Mapping methods.* Search problems in nonmetric spaces can be transformed into metric or vector spaces, where sufficient tools (metric or spatial access methods [Samet, 2006]) have already been developed. However, mapping methods are usually expensive and bring inaccuracy to the retrieval. A mapping from a nonmetric to a metric space is accomplished by turning the employed semimetric into a metric. Hence, the task for such mapping, is enforcing the triangle inequality. Among several approaches solving this problem, we name the *constant shifting embedding* [Roth et al., 2002] and the *TriGen* algorithm described in Chapter 6. A mapping to the vector spaces brings two-fold beneficial effect – first, a cheap metric can be utilized in the target vector space and second, the mapping may serve as a dimensionality reduction tool when mapping from a high-dimensional L_p space. The nonmetric multidimensional scaling and the query sensitive embeddings [Athitsos et al., 2007] are two examples of transforming the nonmetric space to the vector space.

- *General NAMs.* Although general NAMs reuse mapping techniques from the first group, they consider more complex scenarios (e.g. indexing, querying). The *local constant embedding* [Chen and Lian, 2008], the *QIC-M-tree* [Ciaccia and Patella, 2002] and our contribution, the *NM-tree* [Skopal and Lokoč, 2008] described in Chapter 7, belong among these general methods.

- *Specific NAMs.* A very efficient search model can be established by considering specific properties of a particular dissimilarity measure. How-

ever, these domain-specific models are not reusable as general methods for arbitrary nonmetric spaces. A typical example of a specific nonmetric method is the *inverted file*, used for implementation of vector model in the text retrieval under the cosine measure [Berry and Browne, 1999]. Since a typical query consists of only a few terms, there must be only a small part of the database processed. The inverted file is an excellent example that a NAM (cosine measure) could be more effective than a MAM (angle distance), even for the same application.

5.3 Distance modifications

An approach to efficient nonmetric similarity search is the utilization of a dissimilarity-to-dissimilarity transformation f, that is, a transformation of the original dissimilarity δ into a modified (nearly) metric δ^f. The motivation for such a transformation is utilization of MAMs also for efficient nonmetric search. We also show that the concept of turning nonmetric into metric for the purpose of similarity search does not harm the robust nonmetric behavior mentioned in section 5.1. In this chapter, we will remember the basic definitions, lemmas and theorems, which are necessary for the framework proposed in [Skopal, 2007]. Our contribution presented in Chapter 7 is a follow-up to the previously proposed TriGen algorithm and indexing by MAMs.

We will also assume, that δ is a semimetric, that is, it satisfies only the *identity*, *non-negativity* and *symmetry*. The first two properties can be achieved easily – the non-negativity is satisfied by a constant shift, while for the identity property it is required that every two non-identical objects are at least d^--distant. The lower-bound and upper-bound distances d^-, d^+ can be provided either by the structure of the input universe $\mathbb{U}$, or by sampling a number of distances $\delta(o_i, o_j), o_i, o_j \in \mathbb{S}$. The sampling may result only in approximate lower-/upper-bound distances d^-, d^+, hence, outlier distances $d < d^-$ or $d > d^+$ are usually represented directly by d^- or d^+. During the search, the possibly relevant objects involved in distances d^-, d^+ are treated individually. The upper bound d^+ can be further used to normalize the original semimetric δ such that it produces distances from $\langle 0, 1 \rangle$.

Finally, we show how to satisfy the symmetry property. If an original measure d is asymmetric, the search can be partially provided by a symmetric measure δ, e.g., $\delta(o_i, o_j) = min\{d(o_i, o_j), d(o_j, o_i)\}$. After the modified measure δ filters out some non-relevant objects, the original measure d is

used to rank the remaining non-filtered objects.

The hard task is enforcing the triangle inequality. We will summarize necessary formalism for a transformation of dissimilarity δ in the following sections.

5.3.1 Similarity-Preserving Modifiers

A transformation function f could substantially change the original space, hence it is necessary to formally specify constraints for such transformation, in order to preserve some properties of the original space. Especially, the relations between objects should be preserved, that is, each object should see the same neighborhood both in the original space and in the modified space. In the following two definitions, the similarity neighborhood of an object and a suitable transformation are formalized.

Definition 13. *We define* $\text{SimOrder}_\delta : \mathbb{U} \mapsto 2^{\mathbb{U}\times\mathbb{U}}$, $\forall o_i, o_j, o_k \in \mathbb{U}$ *as* $\langle o_i, o_j\rangle \in \text{SimOrder}_\delta(o_k) \Leftrightarrow \delta(o_k, o_i) < \delta(o_k, o_j)$*, i.e.,* SimOrder_δ *orders objects by their distances to* o_k*.*

Definition 14. *Given a dissimilarity measure* δ*, we call* $\delta^f(o_i, o_j) = f(\delta(o_i, o_j))$ *a* similarity-preserving modification of δ *(or* SP-modification*), where* f*, called the* similarity-preserving modifier *(*SP-modifier*), is a strictly increasing function for which* $f(0) = 0$*. For clarity reasons we assume* f *is bounded, i.e.,* $f : \langle 0, 1\rangle \mapsto \langle 0, 1\rangle$*.*

The utilization of similarity preserving modifiers does not change the similarity retrieval model, as shown in the following lemma.

Lemma 1.
Given a dissimilarity δ and any δ^f, $SimOrder_\delta(q) = SimOrder_{\delta^f}(q), \forall q \in \mathbb{U}$.

Proof: As f is increasing, then $\forall q, o_i, o_j \in \mathbb{U}$ it follows that
$\delta(q, o_i) > \delta(q, o_j) \Leftrightarrow f(\delta(q, o_i)) > f(\delta(q, o_j))$. ■

Hence, it does not matter whether we use for a sequential query processing either δ or δ^f, because both induce the same similarity ordering. Naturally, if δ^f is used for a range query processing, the query radius r_q must be modified to $f(r_q)$.

In the following two Sections, we will recall two important types of similarity preserving modifiers – the triangle-generating and the triangle-violating

modifiers, or simply *T-modifiers*. The T-modifiers can be utilized to partially enforce the triangle inequality in a finite database, which may be further employed for an efficient indexing. The following definition helps us to formally describe the amount of the triangle inequality fulfillment.

Definition 15. *A triplet of distances (a, b, c) is triangle triplet if $a + b \geqslant c, b + c \geqslant a, a + c \geqslant b$. If $a \leqslant b \leqslant c$, we call a triangle triplet the ordered triangle triplet.*

The triangle triplets can be viewed as witnesses of triangle inequality of a distance δ – if all triplets $(\delta(o_i, o_j), \delta(o_j, o_k), \delta(o_i, o_k))$ on all possible objects $o_i, o_j, o_k \in \mathbb{S}$ are triangle triplets, then δ satisfies the triangle inequality in the database $\mathbb{S}$. For the sake of simplicity, we can view a triplet (a, b, c) as ordered triplet by permutation of distances $a \leqslant b \leqslant c$, where only inequality $a + b \geqslant c$ has to be checked.

5.3.2 Triangle-Generating Modifiers

The class of triangle-generating modifiers is a special subclass of so-called *metric-preserving functions* [Corazza, 1999]. Within the class of metric-preserving functions, some are simultaneously SP-modifiers, as follows:

Definition 16. *An SP-modifier f is* metric-preserving *if for every metric δ the SP-modification δ^f preserves the triangle inequality, i.e., δ^f is also metric.*[1]

In the following lemma, two interesting properties of SP-modifiers are shown.

Lemma 2.

(a) Every concave SP-modifier f is metric-preserving.
(b) Let (a, b, c) be a triangle triplet and f be a metric-preserving SP-modifier, then $(f(a), f(b), f(c))$ is a triangle triplet as well.

Proof: For the proof and for more about general metric-preserving functions (not only the continuous/monotonous ones) we refer to [Corazza, 1999]. ■

Definition 17. *Let a strictly concave SP-modifier f be called a* triangle-generating modifier *(or* TG-modifier*). Having a TG-modifier f, let a δ^f be called a* TG-modification*.*

[1] Such an SP-modifier is additionally *subadditive* $(f(x) + f(y) \geqslant f(x + y), \forall x, y \in \mathbb{R}_0^+)$.

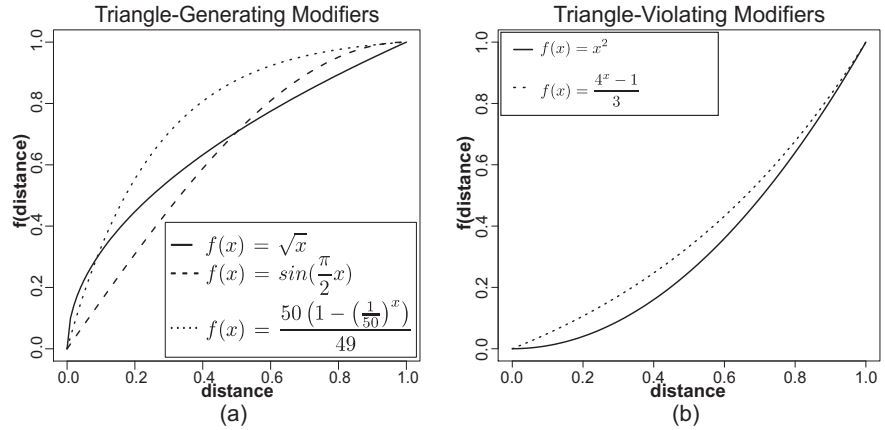


Figure 5.3: Several T-modifiers: (a) TG-modifiers (b) TV-modifiers

By applying a TG-modifier on a metric δ, the metric access methods can be still employed to perform efficient exact search. Moreover, if δ is a semimetric, we can employ TG-modifiers to increase the amount of the triangle inequality fulfillment or even to turn the original semimetric δ into a full metric δ^f (see Theorem 1). Several examples of TG-modifiers are depicted in the Figure 5.3a.

Lemma 3.
Let f be an increasing concave function, such that $f(0) = 0$. Let us have $a, c \in \mathbb{R}^+$ such that $a < c$. Then

$$\frac{f(a)}{f(c)} > \frac{a}{c}$$

Proof: Consider a linear function

$$g(x) \quad = \quad \frac{f(c)}{c}x \tag{5.1}$$

According to this function, we get $g(a)$ which divides the interval $\langle 0, f(c)\rangle$ in the same proportion as a divides the interval $\langle 0, c\rangle$ (this is a direct consequence of $g(x)$'s linearity, for illustration see figure 5.4a). By substitution of $x = a$ in (5.1) we get

$$\frac{g(a)}{f(c)} \quad = \quad \frac{a}{c} \tag{5.2}$$

Because $f(x)$ is concave, $g(x) < f(x), \forall x \in (0, c)$ (by definition of concave functions), hence,

$$g(a) \quad < \quad f(a) \tag{5.3}$$

Finally, by (5.2) and (5.3), it follows that

$$\frac{f(a)}{f(c)} > \frac{a}{c}$$

■

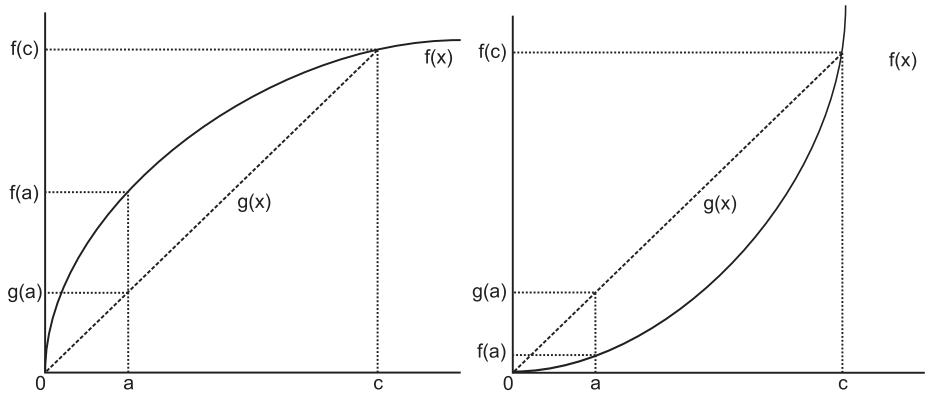


Figure 5.4: (a) Concave and (b) convex function

Theorem 1.

Given a semimetric δ, then there always exists a TG-modifier f, such that the SP-modification δ^f is a metric.

Proof: We show that every ordered triplet (a, b, c) generated by δ can be turned by a single TG-modifier f into an ordered triangle triplet.

1. As every semimetric is reflexive and non-negative, it generates ordered triplets just of forms $(0, 0, 0)$, $(0, c, c)$, and (a, b, c), where $a, b, c > 0$. Among these, just the triplets (a, b, c), $0 < a \leqslant b < c$, can be non-triangular. Hence, it is sufficient to show how to turn such triplets by a TG-modifier into triangular ones.

2. Suppose an arbitrary TG-modifier f_1. From TG-modifiers' properties it follows that $\frac{f_1(a)}{f_1(c)} > \frac{a}{c}$, $\frac{f_1(b)}{f_1(c)} > \frac{b}{c}$, hence $\frac{f_1(a)+f_1(b)}{f_1(c)} > \frac{a+b}{c}$ (application of

Lemma 3). If $(f_1(a) + f_1(b))/f_1(c) \geqslant 1$, the triplet $(f_1(a), f_1(b), f_1(c))$ becomes triangular (i.e., $f_1(a) + f_1(b) \geqslant f_1(c)$ is true). In case there still exist triplets which have not become triangular after application of f_1, we take another TG-modifier f_2 and compose f_1 and f_2 into $f^*(x) = f_2(f_1(x))$. The compositions (or nestings) $f^*(x) = f_i(\dots f_2(f_1(x))\dots)$ are repeated until f^* turns all triplets generated by δ into triangular ones – then f^* is the single TG-modifier f we are looking for. ■

As was shown in the proof, the more concave TG-modifier is applied, the more triplets become triangular, however, the more the objects forming a triplet become also equidistant. Unfortunately, these two important properties of a modified space go against each other. On one hand, more triangle inequality fulfillment can be employed for exact metric indexing. On the other hand, equidistant objects do not form separated clusters and thus filtering rules during search usually fail.

To visualize triangular triplets and the effect of a transformation, consider a space $\langle 0,1\rangle \times \langle 0,1\rangle \times \langle 0,1\rangle$ of all possible distance triplets. Dimensions in the space are represented by distance values a,b,c. Each point in the space represents one triplet and sets of (non-)triangular triplets form 3D regions. In Figures 5.5a,b see examples of region[2] Ω of all triangular triplets (depicted as the dotted-line-bounded area). The super-region Ω^f (depicted as the solid-line-bounded area) represents all the triplets which become (or remain) triangular after the application of TG-modifier $f(x) = x^{\frac{3}{4}}$ and $f(x) = sin(\frac{\pi}{2}x)$.

By application of TG-modifications, the distances are larger and more equidistant which implies the mean of distances increases and the variance decreases. Hence the intrinsic dimensionality of TG-modified distance δ^f is always greater than that of δ. In Figure 5.6b see an example of distance distribution of TG-modified L_2 distance (for the not modified L_2 distance distribution see Figure 5.6a). Let us denote, that the same behavior appears also in vector spaces[3], where the higher dimensionality leads to more equidistant vectors.

5.3.3 Triangle-Violating Modifiers

The triangle-violating modifiers and modifications can be seen as "inverse" transformations to the triangle-generating modifiers and modifications.

[2]The 2D representations of Ω and Ω^f regions are c-cuts of the real 3D regions.

[3]The *curse of dimensionality* problem [Böhm et al., 2001].

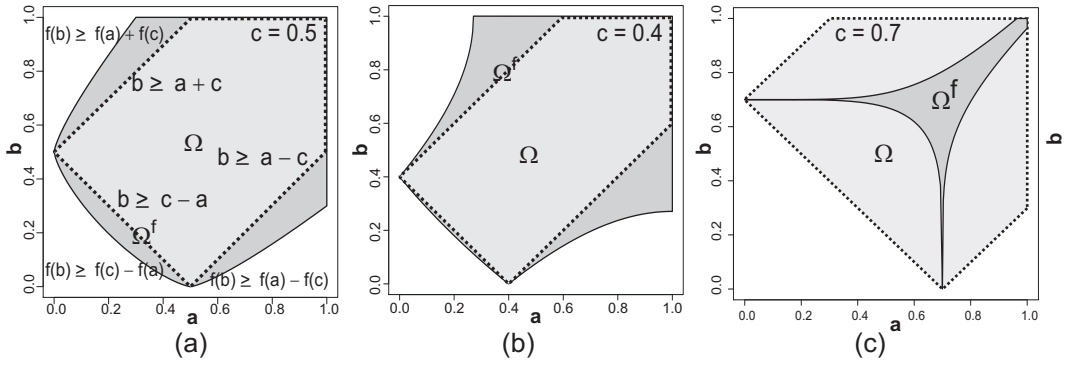


Figure 5.5: Triplet regions of (a, b) TG-modification and (c) TV-modification in the c-cut planes

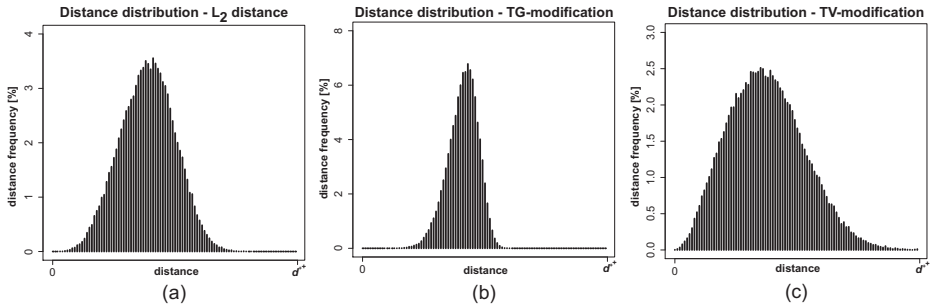


Figure 5.6: Distance distribution using (a) L_2 (b) L_2^f, $f(x) = x^{\frac{1}{2}}$ (c) L_2^f, $f(x) = x^{\frac{3}{2}}$

Definition 18. *Let a strictly convex SP-modifier f be called a* triangle-violating modifier *(or* TV-modifier*). Having a TV-modifier f, let a δ^f be called a* TV-modification*.*

By application of the TV-modifiers (see Figure 5.3b) some triangular triplets may become non-triangular, which turns an original metric to a semi-metric. In the following, we will recall a lemma used in Theorem 2.

Lemma 4.

Let f be an increasing convex function, such that $f(0) = 0$. Let's have $a, c \in \mathbb{R}^+$ such that $a < c$. Then

$$\frac{f(a)}{f(c)} < \frac{a}{c}$$

Proof: Consider a linear function

$$g(x) \;=\; \frac{f(c)}{c}x \tag{5.4}$$

According to this function, we get $g(a)$ which divides the interval $\langle 0, f(c)\rangle$ in the same proportion as a divides the interval $\langle 0, c\rangle$ (this is a direct consequence of $g(x)$'s linearity, for illustration see figure 5.4b). By substitution of $x = a$ in (5.4) we get

$$\frac{g(a)}{f(c)} \;=\; \frac{a}{c} \tag{5.5}$$

Because $f(x)$ is convex, $g(x) > f(x), \forall x \in (0, c)$ (by definition of convex functions), hence,

$$g(a) \;>\; f(a) \tag{5.6}$$

Finally, by (5.5) and (5.6), it follows that

$$\frac{f(a)}{f(c)} < \frac{a}{c}$$

■

Theorem 2.

Given a metric δ, then there always exists a TV-modifier f, such that the SP-modification δ^f is a semimetric.

Proof: The proof is exactly the opposite to that of Theorem 1. Let f_1 be an arbitrary TV-modifier. From TV-modifiers' properties it follows that $\frac{f_1(a)}{f_1(c)} < \frac{a}{c}$, $\frac{f_1(b)}{f_1(c)} < \frac{b}{c}$, hence $\frac{f_1(a)+f_1(b)}{f_1(c)} < \frac{a+b}{c}$ (application of Lemma 4). If $(f_1(a) + f_1(b))/f_1(c) < 1$, the triplet $(f_1(a), f_1(b), f_1(c))$ becomes non-triangular (i.e., $f_1(a) + f_1(b) < f_1(c)$ is true). By composing TV-modifiers we increase the variance among distances within distance triplets, so that some of them sooner or later become NOT triangular triplets (we omit the exotic case where all the objects in $\mathbb{U}$ are equidistant). ■

In consequence, the properties of TG-modifiers discussed in the previous section hold inversely for TV-modifiers. In Figure 5.5c see examples of region Ω of all triangular triplets. The region Ω^f represents all the triplets which remain triangular after the application of TV-modifier $f(x) = x^5$. Furthermore, every (strictly convex) TV-modification exhibits lower intrinsic dimensionality than the original (semi)metric δ (with respect to $\mathbb{S}$), see the distance distribution in Figure 5.6c (compare to Figure 5.6a).

5.4 Discussion

In this chapter, we have summarized motivation for the nonmetric search and presented several examples of popular nonmetric measures. In nonmetric spaces a dissimilarity measure δ is not constrained by any properties, so we have no clue for indexing. Hence, we have discussed the way how to turn any nonmetric to a semimetric. Furthermore, we have also discussed TV-/TG-modifiers which can either increase or decrease the amount of triangle inequality of a particular dissimilarity measure. An automatic tool for finding suitable modifiers is presented in the following chapter.

Chapter 6

TriGen Algorithm

In this chapter, we will remember the TriGen algorithm [Skopal, 2006, 2007] that can put more or less of the triangle inequality into any *semimetric* δ. Thus, any semimetric distance can be turned into an equivalent full metric, or to a semimetric which satisfies the triangle inequality to some user-defined extent. Conversely, TriGen can also turn any full metric into a semimetric which preserves the triangle inequality only partially; this is useful for faster but only approximate search. For its functionality the TriGen needs a (small) sample $\mathbb{S}^* \subset \mathbb{S}$ of the database objects.

6.1 T-error

Using MAMs in combination with a TV-/TG-modification of δ affects both the retrieval efficiency and the retrieval error. Hence, some measure is needed, providing an estimation of how MAMs will behave when using a modified distance. The *T-error* (proposed in [Skopal, 2007]) represents such a measure - it is a degree of triangle inequality violation in a particular database equipped by a semimetric δ. The *T-error* is computed as the proportion of non-triangle triplets in all examined distance triplets, i.e.,

$$\epsilon_{\delta,\mathbb{S}} = \frac{m_{nt}}{m}$$

where m is the total number of sampled distance triplets and m_{nt} is the number of non-triangular triples.

In order to decrease a T-error evaluation time, a small sample of the distance triplets is needed. Without sampling, all possible triplets have to be

checked, resulting in exhaustive processing $n \times (n-1)$ distance computations and $\binom{n}{3}$ triplet checks ($n = |\mathbb{S}|$). On the other hand, the total number of triplets m sampled from a database sample $\mathbb{S}^* \subset \mathbb{S}$ affects the precision of T-error. Ideally, we would like to obtain $\epsilon_{\delta,\mathbb{S}^*} = \epsilon_{\delta,\mathbb{S}} = \epsilon_{\delta,\mathbb{U}}$. Hence, rather than simple random techniques, a suitable sampling technique minimizing discrepancy between $\epsilon_{\delta,\mathbb{S}^*}$ and $\epsilon_{\delta,\mathbb{U}}$ must be incorporated. Since the simple random techniques could miss some *anomalous ordered triplets* (a, b, c) where the ratio $\frac{a+b}{c}$ is extremely low, the zero T-error does not have to guarantee that the modified dissimilarity measure is metric. Hence, to guarantee exact search for a modified semimetric with zero T-error, also the anomalous triplets should be considered and sampled in addition to the randomly sampled ones. For more details and for anomalous triplets sampling heuristic, see [Skopal, 2007].

The T-error is expected to be correlated with the real retrieval error exhibited by any MAM when used with a semimetric distance. The real retrieval error can be defined as a kind of *Jaccard distance*, the normed overlap distance E_{NO}. Let QR_{MAM} be the query result returned by a MAM and QR_{SEQ} be the query result obtained by sequential search of the database, then the retrieval error E_{NO} is defined as:

$$E_{NO} = 1 - \frac{|QR_{MAM} \cap QR_{SEQ}|}{max(|QR_{MAM}|, |QR_{SEQ}|)}$$

Let θ be a user-defined *threshold value* for T-error. For our retrieval task, we would like to find such TV-/TG-modifier f of δ, that $\epsilon_{\delta^f,\mathbb{S}} \leqslant \theta$. To automatically find such modifiers, we can utilize the TriGen algorithm. However, before we remember the TriGen algorithm, we have to show its cornerstone – the T-bases.

6.2 T-bases

The principle behind TriGen is a usage of triangle triplets and T-bases, which generate a subset of TV-/TG-modifiers.

Definition 19. *Let* $f : \langle 0, 1\rangle \times \mathbb{R} \mapsto \mathbb{R}_0^+$ *such that* $f(x, 0) = x$, $f(x, w)$ *is a TG-modifier for* $w > 0$ *and it is a TV-modifier for* $w < 0$, *where* w *is called the* concavity-convexity weight *(CC-weight). Furthermore, if* $w_1, w_2 > 0 \wedge w_1 > w_2$, *then* $f(x, w_1) > f(x, w_2), \forall x \in (0, 1)$. *Conversely, if* $w_1, w_2 <$

$0 \wedge w_1 < w_2$, *then* $f(x, w_1) < f(x, w_2), \forall x \in (0, 1)$. *We also require* f *is continuous in sense* $lim_{w_1 \to w_2} f(x, w_1) = f(x, w_2), \forall x \in \langle 0, 1 \rangle$. *Then we call* f *a* base of T-modifiers *(or T-base).*

A *T-base* $f(x, w)$ is an increasing function (where $f(0, w) = 0$ & $f(1, w) = 1$) which turns a value $x \in \langle 0, 1 \rangle$ of an input (semi)metric δ into a value of a target (semi) metric δ^f, i.e., $\delta^f(\cdot, \cdot) = f(\delta(\cdot, \cdot), w)$. Besides the input distance value x, the T-base is parameterized also by a fixed weight $w \in \langle -\infty, \infty \rangle$ which determines how concave or convex f should be. The higher $w > 0$, the more concave f, which means also the lower T-error of any δ^f. Conversely, the lower $w < 0$, the more convex f and the higher T-error of any δ^f ($w = 0$ means f is identity).

For example, in Figure 6.1 see two T-bases, the *fractional power T-base* (FP-base) defined as:

$$\mathrm{FP}(x, w) = \begin{cases} x^{\frac{1}{1+w}} & \text{for } w > 0 \\ x^{1-w} & \text{for } w \leqslant 0 \end{cases}$$

and one of the *rational Bézier quadratic T-bases* (RBQ-bases), defined as:

$$\mathrm{RBQ}_{(a,b)}(x, w) = \begin{cases} rbq(x, w, a, b) & \text{for } w > 0 \\ rbq(x, -w, b, a) & \text{for } w \leqslant 0 \end{cases}$$

where $rbq(x, w, a, b) =$

$$\frac{-(\Psi - x + wx - aw) \cdot (-2bwx + 2bw^2x - 2abw^2 + 2bw - x + wx - aw + \Psi(1 - 2bw))}{(-1 + 2aw - 4awx - 4a^2w^2 + 2aw^2 + 4aw^2x + 2wx - 2w^2x + 2\Psi(1 - w))}$$

and $\Psi = \sqrt{-x^2 + x^2w^2 - 2aw^2x + a^2w^2 + x}$.

The fractional power T-base is an example of a modifier with the global control of concavity/convexity (controlled just by w), while the rational Bézier quadratic T-bases are furthermore locally adjustable by the second point of the Bézier quadratic curve[1].

[1]The three Bézier points are specified as $P_1(0, 0)$, $P_2(a, b)$ and $P_3(1, 1)$.

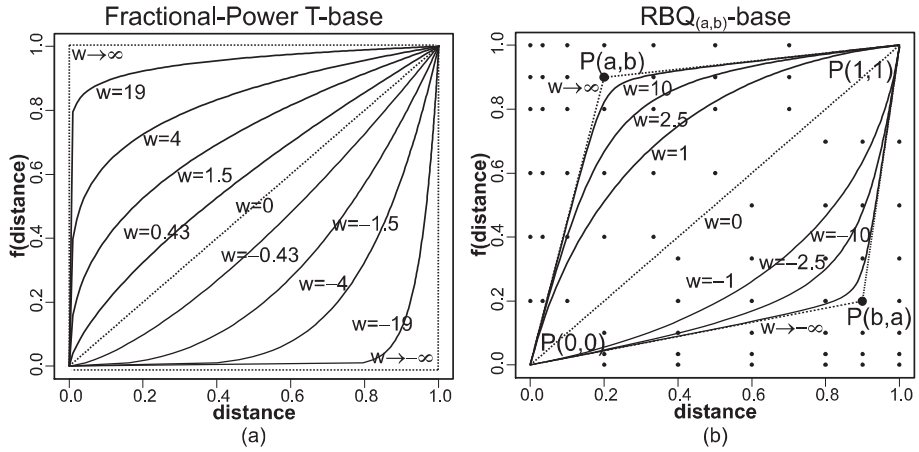


Figure 6.1: T-bases: (a) FP-base (b) RBQ(a,b)-base

6.2.1 Indexability

When choosing very high w (i.e., very concave f), we could turn virtually any semimetric δ into a full metric δ^f. However, such a modification is not very useful from the indexability point of view. The more concave f, the higher *intrinsic dimensionality* of the data space (for definition see Section 1.4.5), which leads to poor partitioning/indexing by any MAM. On the other hand, the more convex f, the lower intrinsic dimensionality of the data space but also the higher the T-error – this results in fast but only approximate searching, because now the MAMs' assumption on fully preserved triangle inequality is incorrect. Hence, we have to make a trade-off choice – whether to search quickly but only approximately using a dissimilarity measure with higher T-error, or to search slowly but more precisely.

6.3 TriGen algorithm

Given a user-defined T-error tolerance θ, a sample $\mathbb{S}^*$ of the database, and an input (semi)metric δ, the TriGen's job is to find a modifier f so that the T-error of δ^f is kept below θ and the intrinsic dimensionality of $(\mathbb{S}^*, \delta^f)$ is minimized[2]. For each of the predefined T-bases the minimal w is found (by halving the weight interval), so that the weight w cannot be further decreased

[2]As shown in [Skopal, 2007, 2006], the real retrieval error of a MAM using δ^f is well estimated by the T-error of δ^f, hence, θ can be directly used as a retrieval precision

without T-error exceeding θ. Among all the processed T-bases and their final weights, the one is chosen which exhibits the lowest intrinsic dimensionality, and returned by TriGen as the *winning T-modifier* (for details of TriGen see [Skopal, 2007]). In Listing 6, see the TriGen algorithm which seeks for the optimal TV-/TG-modifier.

Listing 6. (the generalized TriGen algorithm)

Input: (semi)metric δ, pool $\mathcal{F}$ of T-bases, sample $\mathbb{S}^*$, T-error tolerance threshold θ, iteration limit iterLimit, number of sampled triplets m
Output: optimal f, w

```
f = w = null; maxIndexability = −∞

T = SampleTriplets(m, S*, δ)

for each f* in F
    w* = 0; iter = 0
    error = TriangleError(f*, w*, T)        // compute the initial error of non-modified δ, i.e., w* = 0
    if error < θ then                       // TV-modifiers will be used
        w_LB = 0; w_UB = −∞; w_best = 0
        err_best = error; w* = -1
    else                                    // TG-modifiers will be used
        w_LB = 0; w_UB = ∞; err_best = 0
        w_best = -1; w* = 1
    end if

    while (iter < iterLimit)                // the main algorithm – halving/doubling the weight w*
        error = TriangleError(f*, w*, T)
        if w_UB > 0 then
            if error ⩽ θ then
                w_UB = w_best = w*
                err_best = error;
                w* = (w_LB + w_UB) / 2
            else
                w_LB = w*
                if w_UB ≠ ∞ then
                    w* = (w_LB + w_UB) / 2
                else
                    w* = 2 * w*
                end if
            end if
        else
            if error ⩽ θ then
                w_LB = w_best = w*
                err_best = error;
                if w_UB ≠ −∞ then
                    w* = (w_LB + w_UB) / 2
                else
                    w* = 2 * w*
                end if
            else
```

threshold.

```
            w_UB = w*
            w* = (w_LB + w_UB) / 2
        end if
    end if
    iter++
end while

if (w_UB > 0 and w_best > -1) or (w_UB < 0 and w_best < 1) then // store the best (f,w) found
    w* = w_best
    indexability = ComputeIndexability(f*, w*, T)
    if indexability > maxIndexability then
        f = f*; w = w_best
        maxIndexability = indexability
    end if
end if
end for each
```

6.3.1 Experimental Results

Let us summarize the results presented in [Skopal, 2007]. The author has examined 26 dissimilarity "black-box" measures on four different datasets. Both semimetrics and metrics have been considered for exact (T-error $\theta = 0$) and approximate search (T-error $\theta > 0$). All the distances have been normed to return values from $\langle 0, 1 \rangle$. As T-bases, the FP-base and 115 RBQ-bases have been employed. The sampled triplets consisted from randomly selected triplets and also from anomalous triplets. As the optimality criterion for the best T-bases, the intrinsic dimensionality (iDim) and the ball overlap factor (BOF) have been utilized. The results can be summarized as follows:

- The modified distances (for $\theta > 0$) indicate lower iDim and BOF than their metric variants. The iDim was in some cases even 32 times lower. It can be caused also by the fact, that preserving fully metric behavior for some semimetrics can dramatically increase the iDim. On the other hand, there appeared distances, that satisfied $\theta = 0$ even after TV-modifications, so they did not harm their metric behavior.

- In contrast to iDim-driven TriGen, the BOF-driven TriGen produces TV-/TG-modifications, which perform better in M-tree based indexes, however, with higher retrieval errors.

- The increasing k in kNN queries does not considerably change the retrieval error, hence, we can expect stable behavior under various query selectivities.

- The distance modifications can be successfully applied also on high-dimensional data. For example, although the time series are hard to index for zero retrieval error, for T-error tolerance $\leqslant 0.01$ the performance improves quite significantly and the retrieval error is still reasonably small.

- The size of the database sample used for triplets sampling has impact both on the performance and the retrieval error. Nevertheless, for sample consisting of 7% of the database size, the behavior is stable in all tested databases. The limited size of the sample has also an impact on the required ratio of anomalous triplets, which are used to stabilize the retrieval error. Hence, applications where a strictly guaranteed level of retrieval error is required have to use larger sizes of the sample and larger anomalous triplets sets.

The TriGen is a general and useful framework, that can connect well established MAMs with the nonmetric search problems. Moreover, the TriGen can be used to tune the performance of the existing applications, where slight modifications of distance measures improve the retrieval efficiency and do not harm the requested effectiveness.

6.3.2 Discussion

The winning T-modifier could be subsequently employed by any MAM to index and query the database using δ^f. However, at this moment a MAM's index built using δ^f is not usable if we want to change the approximation level (the T-error of δ^f), that is, to use another f. This is because MAMs accept the distance δ^f as a black box; they do not know it is a composition of δ and f. In such case we have to throw the index away and reindex the database by a δ^{f_2}.

In the following chapter we propose the NM-tree, a NAM based on M-tree natively utilizing TriGen. In NM-tree, any of the precomputed T-modifiers f_i can be flexibly chosen at query time, allowing the user to trade performance for precision.

Chapter 7

NM-tree

In the following, we will present the NM-tree which is a combination of M-tree and the TriGen algorithm. Basically, the NM-tree is an extension of M-tree in terms of algorithms, while the data structure itself is unchanged. The difference in indexing relies in encapsulating the M-tree insertion algorithm by the more general NM-tree insertion (see Section 7.1). The query algorithms have to be slightly redesigned (see Section 7.2).

7.1 Indexing

The distance values (to-parent distances and covering radii) stored in NM-tree are all metric, that is, we construct a regular M-tree using a full metric. Since the NM-tree's input distance δ is generally a semimetric, the TriGen algorithm must be applied before indexing, in order to turn δ into a metric δ^{f_M} (i.e., searching for a T-modifier f_M under T-error $\theta = 0$). However, because at the beginning of indexing the NM-tree is empty, there is no database sample available for TriGen. Therefore, we distinguish two phases of indexing and querying on NM-tree.

For the whole picture of indexing in NM-tree, see Listing 7. The first phase just gathers database objects until we get a sufficiently large set of database objects. In this phase a possible query is solved sequentially, but this is not a problem because the indexed database is still small. When the database size reaches a certain volume (say, $\approx 10^4$ objects, for example), the TriGen algorithm is used to compute f_M using the database obtained so far. At this moment we run the TriGen also for other, user-defined θ_i

values, so that alternative T-modifiers will be available for future usage (for approximate querying). Finally, the first phase is terminated by indexing the gathered database using a series of the original M-tree insertions under the metric δ^{f_M} (instead of δ). In the second phase the NM-tree simply forwards the insertions to the underlying M-tree.

Listing 7. (dynamic insertion into NM-tree)

```
method InsertObject(o_new) {
 if database size < smallDBlimit then
   store o_new into sequential file
 else
   insert o_new into NM-tree (using original M-tree insertion algorithm under δ^{f_M})
 endif
 if database size = smallDBlimit then
   run TriGen algorithm on the database, having θ_M = 0, θ_1, θ_2, ..., θ_k > 0 ⇒
     obtaining T-bases f_M, f_{e_1}, f_{e_2}, ..., f_{e_k} with weights w_M, w_{e_1}, w_{e_2}, ..., w_{e_k}
   for each object o_i in the sequential file
     insert o_i into NM-tree (using original M-tree insertion algorithm under δ^{f_M})
   empty the sequential file
 end if }
```

In contrast to the original TriGen [Skopal, 2007], in NM-tree we require the T-bases f_i to be additionally *inversely symmetric*, defined as:

Definition 20. *A T-base f_i is called inversely symmetric if $f_i(f_i(x, w), -w) = x$.*

In other words, when knowing a T-base f_i with some weight w, we know also the inverse $f_i^{-1}(\cdot, w)$, which is determined by the same T-base and $-w$.

Lemma 5.

The FP-base and all RBQ-bases (see Section 6.2) are inversely symmetric.

Proof: **1.** Let us start with the FP-base, defined as:

$$\mathrm{FP}(x, w) = \begin{cases} x^{\frac{1}{1+w}} & \text{for } w > 0 \\ x^{1-w} & \text{for } w \leqslant 0 \end{cases}$$

For $w = 0$, the FP-base is the identity, hence, $FP(FP(x, 0), 0) = x$ is satisfied.

For $w \neq 0$ we have to prove, that the composition of $x^{\frac{1}{1+w}}$ (for $w > 0$) and x^{1-w} (for $w < 0$) is also the identity.

For $w > 0$ we get $x^{\frac{1}{1+w}} = x^{\frac{1}{1+|w|}}$ and for $w < 0$ we get $x^{1-w} = x^{1+|w|}$. Then, by composition of $FP(x, w)$ and $FP(x, -w)$ we always obtain $\left(x^{\frac{1}{1+|w|}}\right)^{1+|w|} = x^{\frac{1+|w|}{1+|w|}}$, which is the identity.

2. We will also show, that RBQ-bases are inversely symmetric, however, there exist some values of x, where the RBQ-bases are undefined.

For $w = 0$, $rbq(x, 0, a, b) = -\frac{(\Psi - x)^2}{2\Psi - 1} = \frac{x(1-2\Psi)}{1-2\Psi}$, $\Psi = \sqrt{x - x^2}$, a RBQ-base does not depend on values a, b and is the identity, which is not defined only for $x = 0.5$.

For $w \neq 0$ we have to prove, that the composition of $rbq(x, w, a, b)$ and $rbq(x, -w, b, a)$ is also the identity. Nevertheless, the definitions of functions RBQ are already based on symmetric use of rbq, so the proof is trivial. ■

To keep not total RBQ-bases evaluation correct $\forall x \in \langle 0, 1 \rangle$, a possible division by zero or $\Psi^2 < 0$ have to be prevented by slight shift of a or w.

7.2 Query processing

When querying, we distinguish two cases – exact search and approximate search.

7.2.1 Exact search.

The exact case is simple, when the user issues a query with zero desired retrieval error, the NM-tree is searched by the original M-tree algorithms, because of employing δ^{f_M} for searching, which is the full metric used also for indexing. The original user-specified radius r_q of a range query (q, r_q) must be modified to $f_M(r_q)$ before searching. After the query result is obtained, the distances of the query object q to the query result objects o_i must be modified inversely, that is, to $f_M^{-1}(\delta^{f_M}(q, o_i))$ (regardless of range or kNN query).

7.2.2 Approximate search.

The approximate case is more difficult, while here the main qualitative contribution of NM-tree takes its place. Consider a query that has to be processed

with a retrieval error $e_i \in \langle 0, 1\rangle$, where for $e_i = 0$ the answer has to be precise (with respect to the sequential search) and for $0 > e_i \geqslant 1$ the answer may be more or less approximate. The e_i value must be one of the T-error tolerances θ_i predefined before indexing (we suppose the T-error models the actual retrieval error, i.e., $e_i = \theta_i$).

An intuitive solution for approximate search would be a modification of the required δ^{f_M}-based distances/radii stored in NM-tree into $\delta^{f_{e_i}}$-based distances/radii. In such case we would actually get an M-tree indexed by $\delta^{f_{e_i}}$, as used in [Skopal, 2007], however, a dynamic one – a single NM-tree index would be interpreted as multiple M-trees indexed by various $\delta^{f_{e_i}}$ distances. Unfortunately, this "online interpretation" is not possible, because NM-tree (M-tree, actually) stores not only direct distances between two objects (the to-parent distances) but also radii, which consist of aggregations. In other words, except for the two deepest levels (leaf and pre-leaf level), the radii stored in routing entries are composed from two or more direct distances (a consequence of node splitting). To correctly re-modify a radius into the correct one, we need to know all the components in the radius, but these are not available in the routing entry.

Instead of "emulating" multiple semimetric M-trees as mentioned above, we propose a technique performing the exact metric search at higher levels and approximate search just at the leaf and pre-leaf level. In Figure 7.1 see all the distances/radii which are modified to semimetric ones during the search, while the modification is provided by T-bases associated with their user-defined retrieval errors. Besides the to-parent distances, we also consider the query radius and covering radii at the pre-leaf level, because these radii actually represent real distances to a furthest object in the respective query/data region. The query radius and entry-to-query distances (computed as $\delta(\cdot, \cdot)$) are not stored in NM-tree, so these are modified simply by f_{e_i} (where f_{e_i} is a T-base modifier obtained for retrieval error e_i). The remaining distances stored in NM-tree ($\delta^{f_M}(\cdot, \cdot)$-based to-parent distances and covering radii), have to be modified back to the original ones and then re-modified using f_{e_i}, that is, $f_{e_i}(f_M^{-1}(\delta^{f_M}(\cdot, \cdot)))$.

In Listing 8 see the modified range query algorithm. In the pseudocode the "metrized" distances/radii stored in the index are denoted as $\delta^{f_M}(\cdot, \cdot)$, $r_{o_i}^{f_M}$, while an "online" distance/radius modification is denoted as $f_{e_k}(\cdot)$, $f_M^{-1}(\cdot)$. If removed f_M, f_M^{-1}, f_{e_k} from the pseudocode, we would obtain the original M-tree range query, consisting of parent and basic filtering steps (see

Section 2.1).

Listing 8. (NM-tree range query)

```
RangeQuery(Node N, RQuery (q, r_q), retrieval error e_k) {
  let o_p be the parent routing object of N // if N is root then δ(o_i, o_p)=δ(o_p, q)=0

  if N is not a leaf then {
    if N is at pre-leaf level then {                                          // pre-leaf level
      for each rout(o_i) in N do {
        if |f_{e_k}(δ(o_p, q)) − f_{e_k}(f_M^{-1}(δ^{f_M}(o_i, o_p)))| ⩽ f_{e_k}(r_q) + f_{e_k}(f_M^{-1}(r_{o_i}^{f_M})) then {   // (parent filt.)
          compute δ(o_i, q)
          if f_{e_k}(δ(o_i, q)) ⩽ f_{e_k}(r_q) + f_{e_k}(f_M^{-1}(r_{o_i}^{f_M})) then   // (basic filtering)
            RangeQuery(ptr(T(o_i)), (q, r_q), e_k)
        }
      } // for each ...
    } else {                                                                  // higher levels
      for each rout(o_i) in N do {
        if |f_M(δ(o_p, q)) − δ^{f_M}(o_i, o_p)| ⩽ f_M(r_q) + r_{o_i}^{f_M} then {   // (parent filtering)
          compute δ(o_i, q)
          if f_M(δ(o_i, q)) ⩽ f_M(r_q) + r_{o_i}^{f_M} then                   // (basic filtering)
            RangeQuery(ptr(T(o_i)), (q, r_q), e_k)
        }
      } // for each ...
    }
  } else {                                                                    // leaf level
    for each grnd(o_i) in N do {
      if |f_{e_k}(δ(o_p, q)) − f_{e_k}(f_M^{-1}(δ^{f_M}(o_i, o_p)))| ⩽ f_{e_k}(r_q) then {   // (parent filtering)
        compute δ(o_i, q)
        if δ(o_i, q) ⩽ r_q then
          add o_i to the query result
      }
    } // for each ...
  }
```

In Figure 7.2 see a visualization of exact and approximate search in NM-tree. In the exact case, the data space is inflated into a (nearly) metric space, so the regions tend to be huge and overlap each other. On the other hand, for approximate search the leaf regions (and the query region) become much tighter, the overlaps are less frequent, so the query processing becomes more efficient.

7.3 Experimental Results

To examine the NM-tree capabilities, we performed experiments with respect to the efficiency and retrieval error, when compared with multiple M-trees – each M-tree constructed using a fixed modification of δ, related to a user-defined T-error tolerance. We have focused just on the querying, since the

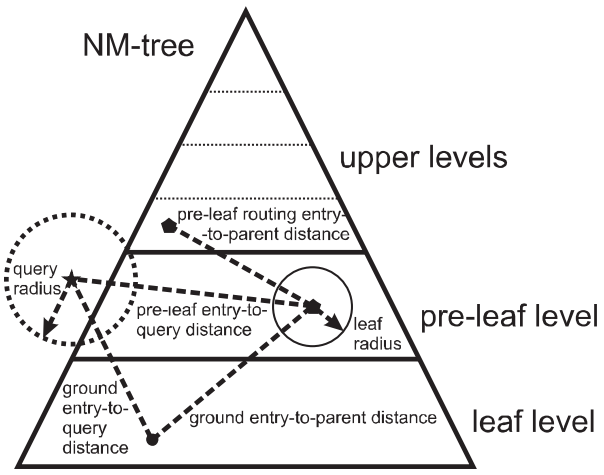


Figure 7.1: Dynamically modified distances when searching approximately

NM-tree's efficiency of indexing is the same as that of M-tree. The query costs were measured as the number of δ computations needed to answer a query. Each query was issued 200 times for different query objects and the results were averaged. The precision of approximate search was measured as the real retrieval error (instead of just T-error); the normed overlap distance E_{NO} between the query result QR_{NMT} returned by the NM-tree (or M-tree) and the correct query result QR_{SEQ} obtained by sequential search of the database, i.e., $E_{NO} = 1 - \frac{|QR_{NMT} \cap QR_{SEQ}|}{max(|QR_{NMT}|, |QR_{SEQ}|)}$.

7.3.1 Databases

We have examined 4 dissimilarity measures on two databases (images, polygons), while the measures δ were considered as black-box semimetrics. All the distances were normed to $\langle 0, 1 \rangle$. The database of images consisted of 68,040 32-dimensional *Corel features* [Hettich and Bay, 1999] (the color histograms were used). We have tested one semimetric and one metric on the images: the $L_{\frac{3}{4}}$ distance (denoted L0.75), and the Euclidean distance (L2). As the second, we created a synthetic database of 250,000 2D polygons, each consisting of 5 to 15 vertices. We have tested one semimetric and one metric on the polygons: the dynamic time warping distance with the L_2 inner distance on vertices [Skopal, 2007] (denoted DTW) and the Hausdorff distance, again with the L_2 inner distance on vertices [Skopal, 2007] (denoted

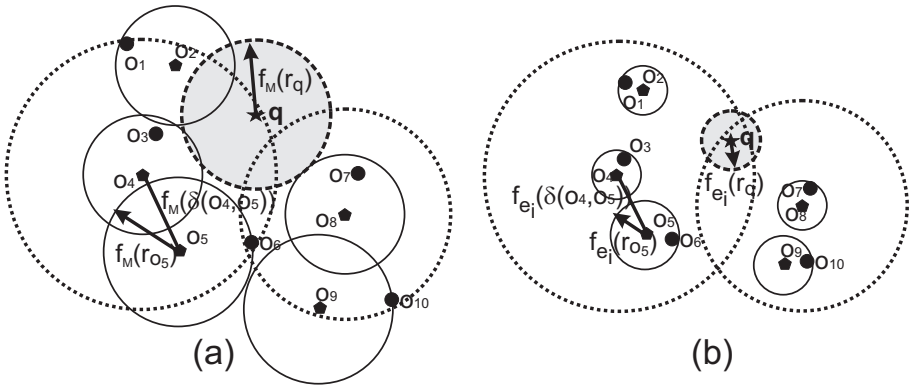


Figure 7.2: (a) Exact (metric) search (b) Approximate search

Hausdorff).

The TriGen inside NM-tree was configured as follows: the T-base pool consisting of the FP-base and 115 RBQ-bases (as in [Skopal, 2007]), sample size 5% of Corel, 1% of Polygons. The NM-tree construction included creation of $4 \cdot 10 = 40$ T-modifiers by TriGen (concerning all the dissimilarity measures used), defined by T-error tolerances [0, 0.0025, 0.005, 0.01, 0.015, 0.02, 0.04, 0.08, 0.16, 0.32] used by querying. The node capacity of (N)M-tree was set to 30 entries per node (32 per leaf); the construction method was set to Single.Classic. The (N)M-trees had 4 levels (leaf + pre-leaf + 2 higher) on both Corel and Polygons databases. The leaf/inner nodes were filled up to 65%/69% (on average).

7.3.2 Querying

In the first experiment we have examined query costs and retrieval error of range queries on Polygons under DTW, where the range query selectivity (RQS) ranged from 0.05% to 0.5% of the database size (i.e., 125–1250 objects), see Figure 7.3. We can see that a single NM-tree index can perform as good as multiple M-tree indexes (each M-tree specifically created for a user-defined retrieval error). In most cases the NM-tree is even slightly better in both observed measurements – query costs and retrieval error.

Note that here the NM-tree is an order of magnitude faster than sequential file when performing exact (nonmetric!) search, and even two orders of magnitude faster in case of approximate search (while keeping the retrieval

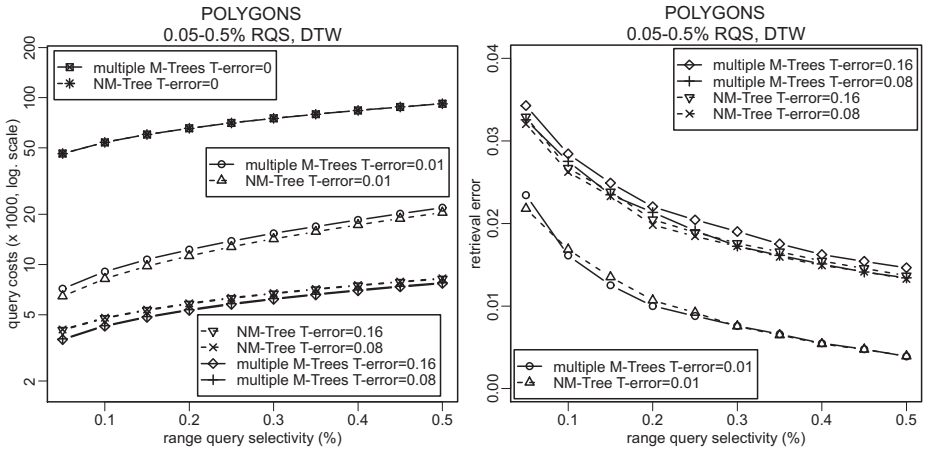


Figure 7.3: Range queries on Polygons under DTW

error below 1%). The Figure 7.3 also shows that if the user allows a retrieval error as little as 0.5–1%, the NM-tree can search the Polygons an order of magnitude faster, when compared to the exact (N)M-tree search.

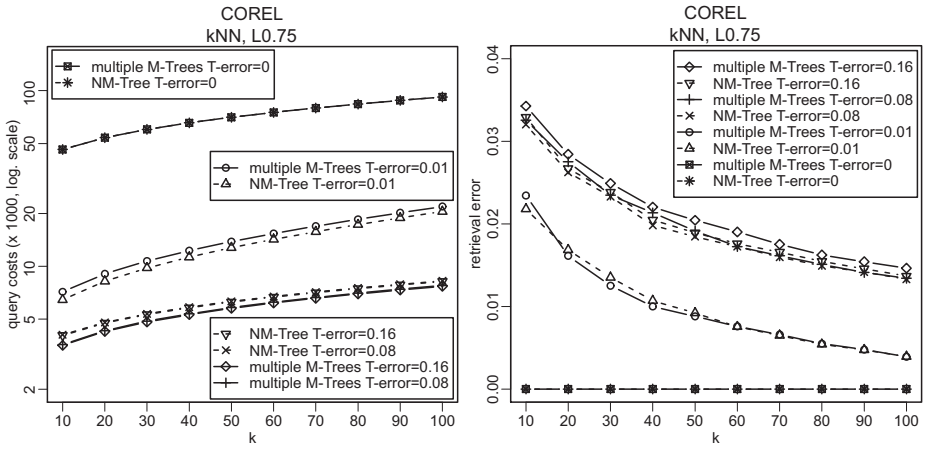


Figure 7.4: kNN queries on Corel under $L_{\frac{3}{4}}$

In the second experiment we have observed the query costs and retrieval error for kNN queries on the Corel database under nonmetric $L_{\frac{3}{4}}$ (see Figure 7.4). The results are very similar to the previous experiment. We can also notice (as in the first experiment) that with increasing query result the retrieval error decreases.

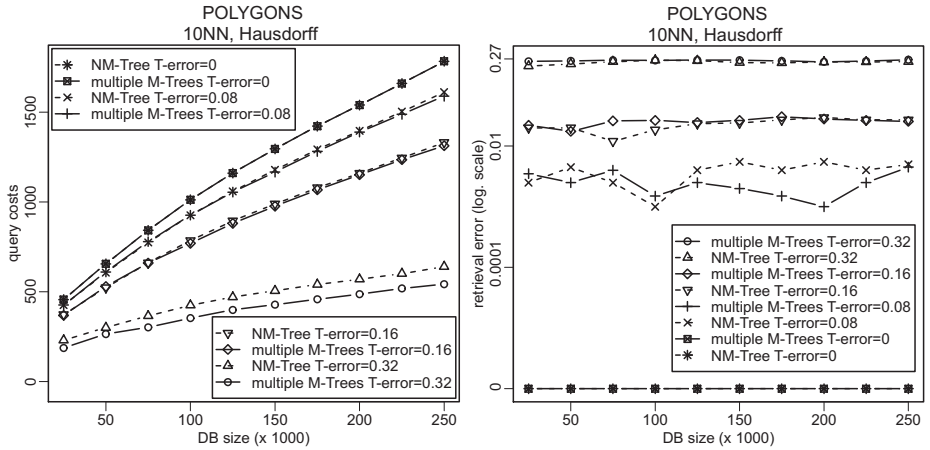


Figure 7.5: 10NN queries on varying size of Polygons under Hausdorff

Third, we have observed 10NN queries on Polygons under Hausdorff, with respect to the growing database size, see Figure 7.5. The query costs growth is slightly sub-linear for all indexes, while the retrieval errors remain stable. Note the T-error tolerance levels (attached to the labels in legends) specified as an estimation of the maximal acceptable retrieval error are apparently a very good model for the retrieval error.

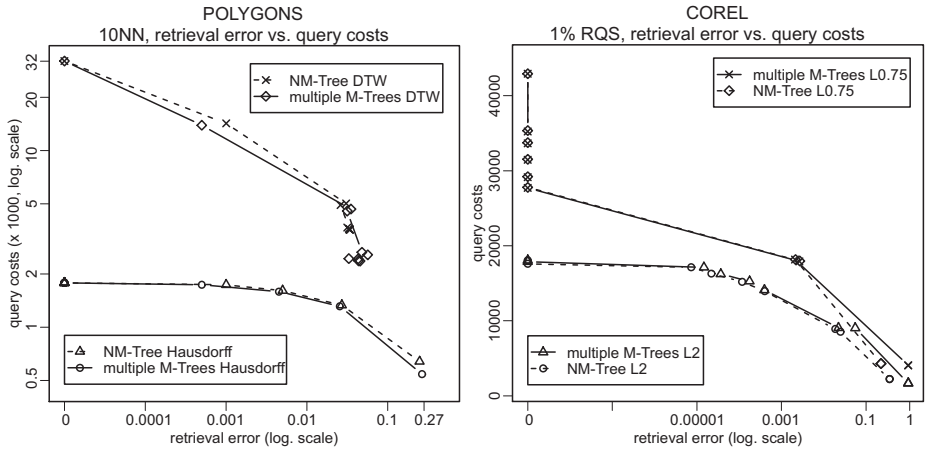


Figure 7.6: Aggregated performance of 10NN queries for Polygons and Corel

In the last experiment (see Figure 7.6) we have examined the aggregated performance of 10NN queries for both Polygons and Corel and all the dis-

similarity measures. These summarizing results show the trade-off between query costs and retrieval error achievable by an NM-tree (and the respective M-trees).

7.4 Discussion

We have introduced the NM-tree, an M-tree-based access method for exact and approximate search in metric and nonmetric spaces, which incorporates the TriGen algorithm to provide nonmetric and/or approximate search. The main feature on NM-tree is its flexibility in approximate search, where the user can specify the approximation level (acceptable retrieval error) at query time. The experiments have shown that a single NM-tree index can search as fast as if used multiple M-tree indexes (each built for a certain approximation level). From the general point of view, the NM-tree, as the only access method for flexible exact/approximate nonmetric similarity search can achieve up to two orders of magnitude faster performance, when compared to the sequential search.

Chapter 8

Conclusions

In this thesis, we have focused on the hierarchical indexing in metric and nonmetric spaces by the M-tree [Ciaccia et al., 1997]. We have proposed several novel methods of dynamic M-tree construction and we have also shown a way how to extend the M-tree to a nonmetric access method NM-tree. All the presented improvements were aiming at more effective and efficient content-based similarity search in very large collections of unstructured data.

The contributions of the thesis can be summarized as follows:

- *Cheap construction of more compact M-tree hierarchies.* The compactness of the metric regions' hierarchy in the M-tree heavily depends on the order of new objects insertions. This fact has been utilized earlier just by the bulk loading algorithm (static method) and by the post-processing slim-down algorithm (static and very expensive method). Hence, we have focused on a new method of dynamic M-tree construction, that performs local redistribution of previously inserted objects, which can better fit some of the newly created regions. These rearrangements decrease unnecessarily big "volumes" and overlaps between regions, which decrease also the probability of their intersection with a query region, and which result in more efficient search. As shown in the experiments, our new method utilizing forced reinserting improves M-tree hierarchy and keeps the construction costs acceptable.

  The quality of the M-tree hierarchy depends also on the leaf selection strategy. Since the cheap single-way leaf selection selects often a nonoptimal leaf and the multi-way leaf selection finding optimal leaf is conversely too expensive, we have introduced the hybrid-way leaf

selection strategy to fill the performance gap between the former two strategies. In the hybrid-way leaf selection, only a limited number of branches (determined by the user-defined branching vector) are visited during the search for a suboptimal leaf node. We have also shown, that forced reinserting and leaf selection strategies can be further combined to control the average leaf node utilization and thus we can control the index size.

- *Scalable M-tree construction method.* The multicore processors become widely available and thus parallel processing can be utilized to significantly speedup M-tree construction. However, parallel processing of hierarchical structures is generally a big problem, because algorithms in such structures consist of consecutive steps, that are usually determined by the previous ones. Hence, the parallelism in tree-based indexes is often limited just to a processing of one tree node. In spite of it, we have found a way how to overcome this limitation in an M-tree construction algorithm and proved, that our new algorithm guarantees significant speedup. We assume that fast indexing is needed, because more than one object has to be indexed at the moment. This assumption justifies utilization of very scalable concurrent insertions. We have also introduced a postponed reinsertions to avoid synchronization problems. Such a scalable algorithm schema is an important step forward, because, with a growing number of available cores, we can quickly (re-)index large collections and we can also employ more sophisticated (and expensive) leaf selection strategies.

- *Efficient nonmetric search.* The metric search model has been introduced as a very general concept for the similarity search. However, as we have discussed in Section 5.1, for some applications the metric space model is still not general enough. For example, in the image retrieval, the requested similarity measures are supposed to be robust and locally sensitive. Such measures are often nonmetrics or (after a simple transformation) just semimetrics. To enable efficient (non-sequential) search for such applications, we have employed a distance transformation approach [Skopal, 2007], that can turn an arbitrary semimetric measure into another semimetric, satisfying triangle inequality to some user-defined extent. Using such a transformation, we can reuse MAMs (applying mainly the triangle inequality) for searching in nonmetric

spaces. The MAMs can be reused either for the exact search or for the approximate search with a user-controlled approximation level (degree of triangle inequality violation). However, MAM's indexes are fixed to the employed dissimilarity measure which is supplied as a "black-box" algorithm. Even if we want to change just the approximation level of the same measure, we have to throw the index away and reindex the database. Hence, we have proposed the NM-tree which overcomes this limitation and allows to specify the approximation level at query time. The NM-tree accepts the original semimetric measure and by use of the TriGen algorithm it evaluates the set of transformation functions, each for a user-defined approximation level. Moreover, we have shown a way how to provide the exact nonmetric search by the NM-tree for an arbitrary input semimetric measure. Thus, a user can decide at query time, whether to trade fast approximate search for a slower but exact one, or vice versa.

8.1 Outlook

In the future, we would like to continue in the research presented in this thesis. We would like to focus not only on direct extensions of the presented work, but we would also like to investigate the related areas, as follows:

- A tight M-tree hierarchy, built using the hybrid-way leaf selection and forced reinserting, is very efficient for similarity search, however, still very expensive to construct. Hence, we would like to employ a suitable combination of presented dynamic strategies of M-tree construction and parallelism to reach acceptable costs. Moreover, we would like to adopt presented algorithms into the PM-tree, that is the successful descendant of the M-tree.

- The effectiveness and efficiency of the NM-tree depends on the selected space transformation. Finding more suitable transformation functions may improve the performance of the NM-tree. We would also like to reuse other MAMs for similarity search in the nonmetric model.

- The D-file employing distance caching proved to be an effective index-free MAM. Hence, we have investigated the impact of the distance caching on the other metric access methods and verified, that distance

caching can improve the query performance in various MAMs (including M-tree) [Skopal et al., 2010].

- Furthermore, we would like to employ the M-tree as a dynamic clustering technique in general metric spaces and compare it with other clustering techniques designed for metric spaces.

Bibliography

Aggarwal, C. C., Hinneburg, A., and Keim, D. A. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Spaces. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*. Springer-Verlag, London, UK, 420–434.

Aggarwal, C. C. and Yu, P. S. 2000. The IGrid Index: Reversing the Dimensionality Curse for Similarity Indexing in High Dimensional Space. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM Press, New York, NY, USA, 119–129.

Amdahl, G. M. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, New York, NY, USA, 483–485.

Angeles-Yreta, A., Solís-Estrella, H., Landassuri-Moreno, V., and Figueroa-Nazuno, J. 2004. Similarity Search in Seismological Signals. In *ENC '04: Proceedings of the Fifth Mexican International Conference in Computer Science*. IEEE Computer Society, Washington, DC, USA, 50–56.

Arevalillo-Herráez, M., Ferri, F. J., and Domingo, J. 2010. A Naive Relevance Feedback Model for Content-based Image Retrieval Using Multiple Similarity Measures. *Pattern Recogn. 43,* 3, 619–629.

Ashby, E. G. and Perrin, N. A. 1988. Toward a Unified Theory of Similarity and Recognition. *Psychological Review 95*, 124–150.

Ashby, F. G. 1992. *Multidimensional Models of Perception and Cognition*. Lawrence Erlbaum Associates, NJ.

Athitsos, V., Hadjieleftheriou, M., Kollios, G., and Sclaroff, S. 2007. Query-sensitive Embeddings. *ACM Trans. Database Syst. 32,* 2, 8.

Baeza-Yates, R. A. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bartolini, I., Ciaccia, P., and Patella, M. 2005. WARP: Accurate Retrieval of Shapes Using Phase of Fourier Descriptors and Time Warping Distance. *IEEE Trans. Pattern Anal. Mach. Intell. 27,* 1, 142–147.

Batko, M., Gennaro, C., and Zezula, P. 2005. A Scalable Nearest Neighbor Search in P2P Systems. In *Proceedings of the the 2nd International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2004), Toronto, Canada*. Lecture Notes in Computer Science, vol. 3367. Springer, 79–92.

Batko, M., Novak, D., Falchi, F., and Zezula, P. 2006. On Scalability of the Similarity Search in the World of Peers. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. ACM, New York, NY, USA, 20.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Rec. 19,* 2, 322–331.

Berry, M. and Browne, M. 1999. *Understanding Search Engines, Mathematical Modeling and Text Retrieval*. Siam.

Blanken, H. M., de Vries, A. P., Blok, H. E., and Feng, L. 2007. *Multimedia Retrieval*. Springer.

Böhm, C., Berchtold, S., and Keim, D. 2001. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys 33,* 3, 322–373.

Bourne, P. and Weissig, H. 2003. *Structural Bioinformatics*. Wiley-Liss.

Brin, S. 1995. Near Neighbor Search in Large Metric Spaces. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases,*

September 11-15, 1995, Zurich, Switzerland, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, 574–584.

Brunelli, R. and Poggio, T. 1993. Face Recognition: Features Versus Templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15,* 10, 1042–1052.

Bustos, B., Keim, D. A., Saupe, D., Schreck, T., and Vranić, D. V. 2005. Feature-based Similarity Search in 3D Object Databases. *ACM Comput. Surv. 37,* 4, 345–387.

Bustos, B., Navarro, G., and Chávez, E. 2003. Pivot Selection Techniques for Proximity Searching in Metric Spaces. *Pattern Recogn. Lett. 24,* 14, 2357–2366.

Bustos, B., Pedreira, O., and Brisaboa, N. 2008. A Dynamic Pivot Selection Technique for Similarity Search. In *SISAP '08: Proceedings of the First International Workshop on Similarity Search and Applications (sisap 2008)*. IEEE Computer Society, Washington, DC, USA, 105–112.

Bustos, B. and Skopal, T. 2006. Dynamic Similarity Search in Multi-Metric Spaces. In *Proceedings of ACM Multimedia, MIR workshop*. ACM Press, 137–146.

CERN. 2010. The Large Hadron Collider [http://public.web.cern.ch/public/en/LHC/LHC-en.html].

Chávez, E., Figueroa, K., and Navarro, G. 2005. Proximity Searching in High Dimensional Spaces with a Proximity Preserving Order. In *MICAI*. 405–414.

Chávez, E., Marroquín, J. L., and Baeza-Yates, R. 1999. Spaghettis: An Array Based Algorithm for Similarity Queries in Metric Spaces. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*. IEEE Computer Society, Washington, DC, USA, 38.

Chávez, E. and Navarro, G. 2001. A Probabilistic Spell for the Curse of Dimensionality. In *ALENEX'01, LNCS 2153*. Springer, 147–160.

Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. 2001. Searching in Metric Spaces. *ACM Computing Surveys 33,* 3, 273–321.

Chen, L., Choubey, R., and Rundensteiner, E. A. 1998. Bulk Insertions into R-trees. Tech. rep., In Proceedings of ACM International Workshop on Advances in Geographic Information Systems.

Chen, L. and Lian, X. 2008. Efficient Similarity Search in Nonmetric Spaces with Local Constant Embedding. *IEEE Trans. on Knowl. and Data Eng. 20,* 3, 321–336.

Chen, L. and Lian, X. 2009. Efficient Processing of Metric Skyline Queries. *IEEE Trans. on Knowl. and Data Eng. 21,* 3, 351–365.

Ciaccia, P. and Patella, M. 1998. Bulk Loading the M-tree, In Proceedings of the 9th Australasian Database Conference (ADC'98), pages 15–26, Perth, Australia.

Ciaccia, P. and Patella, M. 2000. The M2-tree: Processing Complex Multi-Feature Queries with Just One Index. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries* (2002-01-03).

Ciaccia, P. and Patella, M. 2002. Searching in Metric Spaces with User-defined and Approximate Distances. *ACM Trans. Database Syst. 27,* 4, 398–437.

Ciaccia, P., Patella, M., and Zezula, P. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97.* 426–435.

Corazza, P. 1999. Introduction to Metric-preserving Functions. *American Mathematical Monthly 104,* 4, 309–23.

Datta, R., Joshi, D., Li, J., and Wang, J. Z. 2008. Image Retrieval: Ideas, Influences, and Trends of the New Age. *ACM Computing Surveys 40,* 2, 1–60.

Deb, S. 2004. *Multimedia Systems and Content-Based Image Retrieval.* Information Science Publ.

den Bercken, J. V. and Seeger, B. 2001. An Evaluation of Generic Bulk Loading Techniques. In *Proc. 27th International Conference on Very Large Data Bases (VLDB'01)*. Morgan Kaufmann, 461–470.

Dohnal, V., Gennaro, C., Savino, P., and Zezula, P. 2003. D-Index: Distance Searching Index for Metric Data Sets. *Multimedia Tools Appl. 21,* 1, 9–33.

Donahue, M., Geiger, D., Liu, T.-L., and Hummel, R. 1996. Sparse Representations for Image Decomposition with Occlusions. In *CVPR '96: Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*. IEEE Computer Society, Washington, DC, USA, 0007.

Fagin, R. 1999. Combining Fuzzy Information from Multiple Systems. *J. Comput. Syst. Sci. 58,* 1, 83–99.

Falchi, F., Lucchese, C., Perego, R., and Rabitti, F. 2008. CoPhIR: COntent-based Photo Image Retrieval [http://cophir.isti.cnr.it/CoPhIR.pdf].

Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. 1994. Fast Subsequence Matching in Time-series Databases. *SIGMOD Rec. 23,* 2, 419–429.

Figueroa, K. and Fredriksson, K. 2009. Speeding Up Permutation Based Indexing with Indexing. In *SISAP '09: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*. IEEE Computer Society, Washington, DC, USA, 107–114.

Fu, T.-c., Chung, F.-l., Luk, R., and Ng, C.-m. 2007. Stock Time Series Pattern Matching: Template-based vs. Rule-based Approaches. *Eng. Appl. Artif. Intell. 20,* 3, 347–364.

Gantz, J. 2008. The Diverse and Exploding Digital Universe [http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf].

Gennaro, C., Savino, P., and Zezula, P. 2001. Similarity Search in Metric Databases Through Hashing. In *MULTIMEDIA '01: Proceedings*

of the 2001 ACM workshops on Multimedia. ACM, New York, NY, USA, 1–5.

Gustafson, J. L. 1988. Reevaluating Amdahl's Law. *Communications of the ACM 31*, 532–533.

Guttman, A. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, B. Yormark, Ed. ACM Press, 47–57.

Hettich, S. and Bay, S. 1999. The UCI KDD archive [http://kdd.ics.uci.edu].

Howarth, P. and Ruger, S. 2005. Fractional Distance Measures for Content-based Image Retrieval. In *In 27th European Conference on Information Retrieval*. Springer, 447–456.

Huttenlocher, D. P., Klanderman, G. A., Kl, G. A., and Rucklidge, W. J. 1993. Comparing Images Using the Hausdorff Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 850–863.

Jacox, E. H. and Samet, H. 2008. Metric Space Similarity Joins. *ACM Trans. Database Syst. 33,* 2, 1–38.

Jagadish, H. V., Ooi, B. C., Tan, K.-L., Yu, C., and Zhang, R. 2005. iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search. *ACM Trans. Database Syst. 30,* 2, 364–397.

Kelley, J. L. 1975. *General Topology*. Springer-Verlag, New York :.

Keogh, E. and Ratanamahatana, C. A. 2005. Exact Indexing of Dynamic Time Warping. *Knowl. Inf. Syst. 7,* 3, 358–386.

Keogh, E., Wei, L., Xi, X., Vlachos, M., Lee, S.-H., and Protopapas, P. 2009. Supporting Exact Indexing of Arbitrarily Rotated Shapes and Periodic Time Series under Euclidean and Warping Distance Measures. *The VLDB Journal 18,* 3, 611–630.

Korn, F. and Muthukrishnan, S. 2000. Influence Sets Based on Reverse Nearest Neighbor Queries. *SIGMOD Rec. 29,* 2, 201–212.

Krumhansl, C. L. 1978. Concerning the Applicability of Geometric Models to Similar Data: The Interrelationship between Similarity and Spatial Density. *Psychological Review 85,* 5, 445–463.

Levenshtein, V. I. 1965. Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones. *Problems of Information Transmission 1*, 8–17.

Lokoč, J. 2009. Parallel Dynamic Batch Loading in the M-tree. In *SISAP '09: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*. IEEE Computer Society, Washington, DC, USA, 117–123.

Lokoč, J. and Skopal, T. 2008. On Reinsertions in M-tree. In *SISAP '08: Proceedings of the First International Workshop on Similarity Search and Applications (sisap 2008)*. IEEE Computer Society, Washington, DC, USA, 121–128.

Mandl, T. 1998. Learning Similarity Functions in Information Retrieval. *EUFIT '98. 6th European Congress on Intelligent Techniques and Soft Computing*, 771–775.

Micó, L., Oncina, J., and Carrasco, R. C. 1996. A Fast Branch & Bound Nnearest Neighbour Classifier in Metric Spaces. *Pattern Recogn. Lett. 17,* 7, 731–739.

Micó, M.L., O. J. V. E. 1992. An Algorithm for Finding Nearest Neighbours in Constant Average Time with a Linear Space Complexity. In *11th International Conference on Pattern Recognition*. Den Haag, Holanda.

Moghaddam, B. and Pentland, A. 1999. Bayesian Image Retrieval in Biometric Databases. *Multimedia Computing and Systems, International Conference on 2*, 610.

Navarro, G. 1999. Searching in Metric Spaces by Spatial Approximation. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*. IEEE Computer Society, Washington, DC, USA, 141.

Navarro, G. 2002. Searching in Metric Spaces by Spatial Approximation. *The VLDB Journal 11,* 1, 28–46.

Navarro, G. and Reyes, N. 2002. Fully Dynamic Spatial Approximation Trees. In *SPIRE 2002: Proceedings of the 9th International Symposium on String Processing and Information Retrieval*. Springer-Verlag, London, UK, 254–270.

Novak, D. and Batko, M. 2009. Metric Index: An Efficient and Scalable Solution for Similarity Search. In *SISAP '09: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*. IEEE Computer Society, Washington, DC, USA, 65–73.

Novak, D. and Zezula, P. 2006. M-Chord: a Scalable Distributed Similarity Search Structure. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. ACM, New York, NY, USA, 19.

Papadias, D., Tao, Y., Fu, G., and Seeger, B. 2003. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 467–478.

Porkaew, K., Mehrotra, S., Ortega, M., and Chakrabarti, K. 1999. Similarity Search Using Multiple Examples in MARS. In *VISUAL '99: Proceedings of the Third International Conference on Visual Information and Information Systems*. Springer-Verlag, London, UK, 68–75.

Robinson, D. D., Lyne, P. D., and Richards, W. G. 2000. Partial Molecular Alignment via Local Structure Analysis. *Journal of Chemical Information and Computer Sciences 40,* 2, 503–512.

Rosch, E. 1975. Cognitive Reference Points. *Cognitive Psychology 7*, 532–47.

Roth, V., Laub, J., Buhmann, J. M., and Müller, K.-R. 2002. Going Metric: Denoising Pairwise Data. In *NIPS*. 817–824.

Rothkopf, E. 1957. A Measure of Stimulus Similarity and Errors in some Paired-associate Learning Tasks. *J. of Experimental Psychology 53,* 2, 94–101.

Rubner, Y. and Tomasi, C. 2001. *Perceptual Metrics for Image Database Navigation*. Kluwer Academic Publishers, Norwell, MA, USA.

Rubner, Y., Tomasi, C., and Guibas, L. J. 1998. A Metric for Distributions with Applications to Image Databases. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*. IEEE Computer Society, Washington, DC, USA, 59.

Samet, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

Santini, S. and Jain, R. 1999. Similarity Measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence 21*, 871–883.

Seidl, T. and Kriegel, H.-P. 1997. Efficient User-Adaptable Similarity Search in Large Multimedia Databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 506–515.

Sexton, A. P. and Swinbank, R. 2004. Bulk Loading the M-Tree to Enhance Query Performance. In *BNCOD*. 190–202.

Shieh, J. and Keogh, E. 2009. iSAX: Disk-aware Mining and Indexing of Massive Time Series Datasets. *Data Min. Knowl. Discov. 19,* 1, 24–57.

Skala, M. 2008. Counting Distance Permutations. In *SISAP '08: Proceedings of the First International Workshop on Similarity Search and Applications (sisap 2008)*. IEEE Computer Society, Washington, DC, USA, 69–76.

Skopal, T. 2004. Pivoting M-tree: A Metric Access Method for Efficient Similarity Search. In *Proceedings of the 4th annual workshop DATESO, Desná, Czech Republic, ISBN 80-248-0457-3, also available at CEUR, Volume 98, ISSN 1613-0073,* `http://www.ceur-ws.org/Vol-98`. 21–31.

Skopal, T. 2006. On Fast Non-metric Similarity Search by Metric Access Methods. In *Proc. 10th International Conference on Extending Database Technology (EDBT'06)*. LNCS 3896. Springer, 718–736.

Skopal, T. 2007. Unified Framework for Fast Exact and Approximate Search in Dissimilarity Spaces. *ACM Trans. Database Syst. 32,* 4.

Skopal, T. and Bustos, B. 2009. On Index-Free Similarity Search in Metric Spaces. In *DEXA '09: Proceedings of the 20th International Conference on Database and Expert Systems Applications*. Springer-Verlag, Berlin, Heidelberg, 516–531.

Skopal, T. and Bustos, B. 2010. On Nonmetric Similarity Search Problems in Complex Domains. *ACM Computing Surveys*.

Skopal, T., Dohnal, V., Batko, M., and Zezula, P. 2009. Distinct Nearest Neighbors Queries for Similarity Search in Very Large Multimedia Databases. In *WIDM '09: Proceeding of the eleventh international workshop on Web information and data management*. ACM, New York, NY, USA, 11–14.

Skopal, T. and Hoksza, D. 2007. Improving the Performance of M-Tree Family by Nearest-Neighbor Graphs. In *ADBIS, LNCS 4690*. 172–188.

Skopal, T. and Lokoč, J. 2008. NM-Tree: Flexible Approximate Similarity Search in Metric and Non-metric Spaces. In *DEXA, LNCS 5181*. 312–325.

Skopal, T. and Lokoč, J. 2009. New Dynamic Construction Techniques for M-tree. *Journal of Discrete Algorithms, Elsevier 7,* 1, 62–77.

Skopal, T. and Lokoč, J. 2010. Answering Metric Skyline Queries by PM-tree. In *Proceedings of the Dateso 2010 Workshop*. Vol. 567. MATFYZPRESS, 22–37.

Skopal, T., Lokoč, J., and Bustos, B. 2010. D-cache: Universal Distance Cache for Metric Access Methods. *Submitted as journal paper*.

Skopal, T., Pokorný, J., Krátký, M., and Snášel, V. 2003. Revisiting M-tree Building Principles. In *ADBIS, Dresden*. LNCS 2798, Springer, 148–162.

Skopal, T., Pokorný, J., and Snášel, V. 2005. Nearest Neighbours Search using the PM-tree. In *DASFAA '05, Beijing, China*. LNCS 3453, Springer, 803–815.

Smith, T. and Waterman, M. 1981. Identification of common molecular subsequences. *Jurnal of molecular Biology 147*, 195–197.

Tahaghoghi, S. M. M., Thom, J. A., and Williams, H. E. 2001. Are Two Pictures Better than One? In *ADC '01: Proceedings of the 12th Australasian database conference*. IEEE Computer Society, Washington, DC, USA, 138–144.

Tahaghoghi, S. M. M., Thom, J. A., and Williams, H. E. 2002. Multiple Example Queries in Content-Based Image Retrieval. In *SPIRE 2002: Proceedings of the 9th International Symposium on String Processing and Information Retrieval*. Springer-Verlag, London, UK, 227–240.

Tang, J. and Acton, S. 2003. An Image Retrieval Algorithm Using Multiple Query Images. In *Signal Processing and Its Applications, 2003. Proceedings. Seventh International Symposium on*. Vol. 1. 193 – 196 vol.1.

Traina, Jr., C., Filho, R. F., Traina, A. J., Vieira, M. R., and Faloutsos, C. 2007. The Omni-family of All-purpose Access Methods: a Simple and Effective Way to Make Similarity Search more Efficient. *The VLDB Journal 16,* 4, 483–505.

Traina Jr., C., Traina, A., Seeger, B., and Faloutsos, C. 2000. Slim-Trees: High Performance Metric Trees Minimizing Overlap between Nodes. *1777*, 51–65.

Tversky, A. 1977. Features of Similarity. *Psychological Review 84*, 327–352.

Tversky, A. and Gati, I. 1982. Similarity, Separability, and the Triangle Inequality. *Psychological Review 89,* 2, 123–154.

Uhlmann, J. K. 1991. Satisfying General Proximity/Similarity Queries with Metric Trees. *Inf. Process. Lett. 40,* 4, 175–179.

Uribe, R., Navarro, G., Barrientos, R. J., and Marín, M. 2006. An Index Data Structure for Searching in Metric Space Databases. In *International Conference on Computational Science (1)*. 611–617.

Vespa, T. G., Traina, C., and Traina, A. J. M. 2007. Bulk-loading Dynamic Metric Access Methods. In *Proceedings of the 22nd Brazilian Symposium on Databases (SBBD'07), ACM SIGMOD DiSC*. 160–174.

Vidal, E. 1994. New Formulation and Improvements of the Nearest-neighbour Approximating and Eliminating Search Algorithm (AESA). *Pattern Recogn. Lett. 15,* 1, 1–7.

Watson, S. 1999. The Classification of Metrics and Multivariate Statistical Analysis. *Topology and its Applications 99,* 2-3, 237 – 261.

Yi, B.-K., Jagadish, H. V., and Faloutsos, C. 1998. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering.* IEEE Computer Society, Washington, DC, USA, 201–208.

Yianilos, P. N. 1999. Excluded Middle Vantage Point Forests for Nearest Neighbor Search. In *DIMACS Implementation Challenge, ALENEX'99.*

Zezula, P., Amato, G., Dohnal, V., and Batko, M. 2005. *Similarity Search: The Metric Space Approach (Advances in Database Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Zezula, P., Savino, P., Amato, G., and Rabitti, F. 1998. Approximate Similarity Retrieval with M-trees. *The VLDB Journal 7,* 4, 275–293.

Zezula, P., Savino, P., Rabitti, F., Amato, G., and Ciaccia, P. 1998. Processing M-Tree with Parallel Resources. In *In Proceedings of the 6th EDBT International Conference.*

Zhang, K. and Shasha, D. 1997. Tree Pattern Matching. 341–371.

Zhou, X., Wang, G., Xu, J. Y., and Yu., G. 2003. M+-tree: A New Dynamical Multidimensional Index for Metric Spaces. In *ADC.* 161–168.