VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Metric Indexing in Information Retrieval

Tomáš Skopal

Ph.D. Thesis

Ostrava

June, 2004

Metric Indexing in Information Retrieval
Ph.D. thesis
urtax.ms.mff.cuni.cz/~skopal/phd/thesis.pdf

© 2004
Mgr. Tomáš Skopal
tomas@skopal.net (feedback from the readers will be appreciating)

Thesis supervisor:

Doc. RNDr. Václav Snášel, CSc.
Department of Computer Science,
VŠB–Technical University of Ostrava
Ostrava, Czech Republic
vaclav.snasel@vsb.cz

Thesis reviewers:

Prof. RNDr. Jaroslav Pokorný, CSc.
Department of Software Engineering,
Charles University in Prague,
Prague, Czech Republic

Prof. Ing. Pavel Zezula, CSc.
Department of Information Technologies,
Masaryk University,
Brno, Czech Republic

Prof. RNDr. Peter Vojtáš, DrSc.
Department of Computer Science,
P. J. Šafárik University,
Košice, Slovakia

Last revision and update: October 17th, 2004

VŠB–Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science
www.cs.vsb.cz

17. listopadu 15, 708 33 Ostrava-Poruba,
Czech Republic

Typeset by pdfLaTeX

"If a man takes no thought about what is distant,
he will find sorrow near at hand."

Confucius

Contents

Preface

In the broad area of Information Retrieval (IR), one of the present-day most important tasks is a solution of effective and efficient similarity search in large collections of unstructured or semi-structured documents. In particular, the volume of multimedia databases as well as of textual collections grows very quickly and the needs for an efficient similarity search in such databases become stronger.

Since similarity measures are usually modelled by metrics (or distance functions), the problem of similarity search in a document collection can be transformed into the problem of searching in a metric dataset (representing the collection). Furthermore, in order to search in a metric dataset efficiently (i.e. quickly), the dataset has to be preprocessed (or indexed) so that a data structure, a *metric index*, is built and used for query processing.

In principle, there exist two ways how to search in metric datasets, the *exact search methods* and the *approximate search methods*. The exact methods search the dataset such that no false drops are allowed, i.e. all of the objects understood as relevant to a query are required to appear in the query result. On the other side, the approximate methods relax such a strong requirement, so that a small proportion of missing objects is accepted as insignificant.

The Goal

The methods of similarity search have been a subject of interest to IR and database communities for a long time. Nevertheless, the area is still rich, offering new problems, while many researchers have spent a lot of efforts to handle them.

The general goal of this Ph.D. thesis is to inventively continue the success in the field of metric indexing achieved so far. In particular, we primarily emphasize the aspect of *search efficiency*, thus all the methods proposed in this thesis are focused on speeding up the similarity search in metric datasets.

We present several contributions to metric indexing, concerning both the exact metric search and the approximate metric search. Furthermore, we apply the metric approach to handle the problem of an efficient similarity search in vector model of Information Retrieval (Text Retrieval respectively). In order to verify their properties, all of the contributions are evaluated by many experiments.

Wherever needed, examples for demonstration of the important principles are included, while the necessary algorithms are described in a form of pseudo-code listings.

Summary of Contributions

In this thesis we extend the ideas of M-tree, which is a hierarchical indexing structure for searching in large metric datasets. We present four main contributions, summarized as follows:

- We propose two novel methods (published in [98]) of building the M-tree hierarchy. Both methods, the *multi-way object insertion* and the *generalized slim-down algorithm*, optimize the M-tree hierarchy so that a higher search efficiency is achieved. We also utilize the fat-factor coefficient as a measure of the quantity of overlaps among metric regions in the M-tree.
- A substantive part of the thesis presents the *PM-tree* (published in [93, 100, 99]), a metric indexing structure combining the M-tree with principles used by pivot-based methods (LAESA method respectively). In order to even more improve the efficiency of metric search, the PM-tree exploits a concept of compact hyper-ring intersections reducing the probability of a "region false hit". In particular, we present
    - the motivation, the basic concepts of PM-tree, and the PM-tree construction algorithms
    - the optimal range query as well as k-NN query algorithm
    - formulation of two cost models for range query as well as k-NN query processing (including experimental evaluation)
    - experiments made on large synthetic datasets as well as on a collection of images
- As a contribution to approximate metric search, we propose the concept of *semi-metric search* (published in [95, 97]), a method of searching in datasets of high intrinsic dimensionalities. The method is based on utilizing specific semi-metric modifications of the original metric, so that the intrinsic dimensionality is reduced. We have applied the semi-metric search for approximate searching in the framework of M-tree.
- In the last contribution, we apply the methods of metric and semi-metric search to the problem of searching in vector model of Text Retrieval (published in [96, 94, 95, 97]). We turn the problem of similarity search using cosine measure into the problem of search using the *deviation metric.* Instead of traditional sequential methods (e.g. the inverted file), the M-tree

can be used to index the document vectors so that documents are hierarchically clustered inside regions of metric space.

Thesis Organization

Besides the introducing and concluding chapters, the thesis is divided in two parts. The first part (Chapters 2,3,4,5) concerns the problem of exact metric search, while the second part (Chapters 6,7,8) deals with methods of approximate search.

In Chapter 1 an introduction to the search problem is covered under the scope of Information Retrieval. We point out that in context of IR the expressiveness of exact-matching queries is insufficient and, therefore, a kind of similarity search concept is needed. We discuss the problem of similarity search in IR and its transformation into the problem of searching in metric spaces.

Several state-of-the-art methods providing metric search are surveyed in Chapter 2. We focus only on exact methods in this chapter, i.e. such methods for which the total accuracy of metric search is guaranteed.

As a basis and a starting point of our research, in Chapter 3 we describe the ideas of M-tree. In particular, we present the M-tree structure, construction algorithms and query processing.

Two novel methods of M-tree construction are described in Chapter 4, the multi-way object insertion and the generalized slim-down algorithm, both of them providing construction of more compact M-tree hierarchies. The M-trees built by either of these methods are experimentally verified to be significantly more efficient in searching.

In Chapter 5 we introduce the PM-tree, a novel extension of the M-tree exploiting principles used by pivot-based methods. We present the basic properties of PM-tree and briefly explain the construction specifics. A comprehensive description is given to the query algorithms and to formulation and evaluation of query cost models. The properties of PM-tree are experimentally verified on large synthetic as well as real-world datasets.

The Chapter 6 introduces into the problem of approximate search. It is demonstrated that exact search in datasets of high intrinsic dimensionality is impracticably difficult. We overview several approaches to approximate but more efficient search in such datasets.

In Chapter 7 we propose a method of approximate semi-metric search. We analyse a theoretical framework concerning metric modifications and inspect semi-metric modifications having suitable clustering properties. The semi-metric modifications are utilized in M-tree, providing an efficient approximate search.

As a real-world application, we have taken advantage of metric search in the domain of Text Retrieval. In Chapter 8 the (semi-)metric approach is utilized for searching in the classic vector model as well as in the LSI model of IR.

In Chapter 9 we conclude the thesis and give an outlook into the future. Author's selected publications related to indexing in IR are listed in Appendix A.

Acknowledgements

I have to thank Václav Snášel for his supervising guidance, novel ideas and an enthusiastic motivation. I thank my parents and my girlfriend Julie for patience, love and support. An acknowledgement belongs also to my colleagues and co-authors Michal Krátký and Pavel Moravec for our mutual teamwork, strong research feedback and good fellowship. Finally, I would like to thank professor Jaroslav Pokorný for cooperation, helpful comments and a valuable insight into the database community.

Tomáš Skopal

June, 2004

Chapter 1

Introduction

Because of rapid developments in information technologies and together with the expansion of information society, the needs for a massive information exploitation grow and the importance of information retrieval becomes stronger. The broad area of Information Retrieval (IR) [86, 9] involves miscellaneous problems and approaches to information storage and retrieval. In particular, the methods of Information Retrieval deal with large data collections of various kinds. Besides the well-known and extensively used relational database technologies, there have appeared many problems recently, where the conventional techniques cannot be applied and completely new approaches have to be designed.

In the past few decades, the scope of Information Retrieval was rather restricted to areas of Text Retrieval and related disciplines – like Pattern Matching, String Processing, etc. However, during the last years the development in IT has brought many other forms of information representation and storage, thus nowadays we can understand the Information Retrieval as a set of disciplines for management and searching in databases of various kinds – that it, in large collections of *unstructured* documents. Such document collections usually consist of:

- text documents (Text Retrieval [15, 49])
- web pages, web sites (Text Retrieval, Semantic Web [14])
- multimedia documents, i.e. images, audio, video (Multimedia Databases [4])
- XML documents or XML subtrees (XML Databases [27, 82])
- etc.

The difference between structured[1] databases (e.g. relational DBs) and unstructured databases (e.g. text documents) is in the description of data structure

[1]Besides the structured and unstructured documents we also distinguish *semi-structured* documents, where the structure is defined only partially (e.g. XML documents complying a DTD or XML schema).

and data semantics. The content of structured databases is always tightly related to an unambiguous database schema defining the structure (e.g. the size and number of attributes) as well as the semantics (e.g. attribute type). On the other side, for an unstructured database there usually exists only a loosely defined schema (e.g. a schema definition like ”text consisting of terms”). Due to that fact, the loosely described unstructured data can be modelled in many ways, depending on the application domain.

1.1 Fields of Interest

The scope of Information Retrieval is very broad, it covers searching, browsing, data modelling, document classification and categorization, system architectures, data visualization, data storage and compression, user interfaces, filtering, modelling and query languages, etc.

Among many problems solved within the scope of Information Retrieval, we focus on the issue of searching in large document collections. In IR the retrieval of stored documents can be usually provided by two different modalities:

1. **Browsing.** The user can browse and navigate through the stored documents in an interactive way, until the retrieved document(s) meet his/her requirements.

2. **Querying/Searching.** Another possibility is a formulation of query which can be issued to the IR system. The query is expressed by means of a query language or by submitting a query document (specified in a given model). Then, the system has to evaluate the query on a given document collection in order to present the result to the user.

In the thesis we deal with the latter modality, i.e. we are concerned with the problem of **querying a large document collection**. Since in database area the meaning of term ”large” is constantly changing over the time, we generally assume that large document collection is required to be stored in secondary (or even tertiary) memory.

1.2 Search in Information Retrieval

Unlike the well-structured databases, where the data semantics is given by a database (or collection) schema, the semantics of unstructured documents is hidden in the document content itself. Given a query to database, we usually want to retrieve all the documents which are ranked as *relevant* to the query. However, the relevance of an unstructured document to a query cannot be determined without an implicit knowledge about the collection semantics. For this reason a *feature extraction* is necessary, converting the unstructured collection of documents $\mathcal{C}$ into a structured dataset $\mathbb{S}$ of *objects* (usually represented by vectors) where each object $O_i \in \mathbb{S}$ describes just one document. Naturally, the feature extraction is dependent on the application domain as well as on particular query semantics (different query types exploit different features).

Example 1.1

Consider a collection of gray-scaled images. A common feature extraction for images is a creation of gray frequency histogram for each image in the collection (e.g. each image is represented by a 256-dimensional feature vector). Such a feature extraction implies that semantics of each query needs just the histogram to qualify an image as relevant/irrelevant to the query.

Example 1.2

The feature extraction can be also viewed as a preprocessing step when a relational database has to be populated. Consider a web catalogue of car models where each web page contains a full-text description of one car model. For retrieval of all the objects relevant to a reasonable query, we need to know the basic technical parameters of each car model. Hence, the feature extraction processing must recognize those parameters in each catalogue page and form a feature vector (a row in a table) for each page in the catalogue. The vector coordinates (columns in a table) represent the extracted features or attributes (e.g. `MODEL(string)`, `CONSUMPTION(float)`, `MAXSPEED(int)`, `WEIGHT(int)`, `PRICE(int)`,...). Such a feature extraction implies the query semantics to be related just to the extracted attributes of car models.

1.2.1 Exact-Match Queries

Given a dataset $\mathbb{S}$ and a query predicate $\mathcal{Q}$, the database system has to determine which objects in $\mathbb{S}$ are relevant (in some sense) to the predicate $\mathcal{Q}$. In classic relational database systems, the fundamental operation is *exact matching*, thus only such objects from $\mathbb{S}$ are retrieved, which exactly match the predicate $\mathcal{Q}$. Each object from $\mathbb{S}$ is classified as *strictly relevant* or *strictly irrelevant* to the query predicate.

Example 1.3

Consider the catalogue of cars described in Example 1.2. An exact-match query can be formulated by the standard SQL SELECT statement as:

```
SELECT * FROM CARS WHERE MAXSPEED > 120 AND CONSUMPTION < 6
          OR PRICE < 10000 OR MODEL LIKE 'BMW*'
```

All of the retrieved objects (car pages in the catalogue) exactly match the query predicate (the WHERE statement) and there is no further relevance ranking within the query result.

1.2.2 Limitations of Exact-Match Queries

In the area of Information Retrieval, the usage of exact-match queries is limited by several paradigmatic factors:

1. In order to retrieve just the objects of his or her interest, the user must be able to specify a correct query predicate. On the other side, in the diversified area of Information Retrieval the queries (the query predicates consequently) are required to capture a variety of relevance interpretations. Unfortunately, in many cases a strict separation of all the documents between two classes of relevant and irrelevant documents is impracticable thus, consequently, a formulation of an appropriate query predicate is often impossible.

2. To specify a query predicate, the user must be able to interpret the features (attributes) of an object description. In case of the car dataset (see Example 1.2) the attribute interpretation is easy, but in case of the image dataset (see Example 1.1) the attributes, representing frequencies of the gray shades in an image, are not so clear to cope with.

3. The objects in a query result are all of the same relevance to the query predicate. There is no further relevance ordering in the query result. Consequently, when a huge query result is obtained and the user cannot choose a few most significant objects among the others, then a more specific query predicate has to be constructed and the query evaluation repeated.

In the context of Information Retrieval, the exact-match queries are not expressive enough. The reason is that in the environment of loosely defined information semantics we rather want to retrieve those objects from the dataset $\mathbb{S}$, which are somehow *partially* relevant to a *query object*[2] (or pattern object). That is, there is a strong need for a generalization of the exact-match queries beyond the resolution of binary relevance.

More specifically, what is usually needed is a notion of *similarity* – the user could query the system to assess a similarity value between each object in the dataset and a given query object. In such a scenario, the result of a query is a list where all the retrieved objects are sorted by decreasing values of their similarity with respect to the query object.

1.3 Similarity Search

The questions concerning similarity have been intensively investigated throughout the 20th century in the field of psychology as well as computer science, attempting

[2]The query object is represented the same way as a data object is.

to define a theory consistent with the huge amount of experimental real-world data. A major discrepancy between the similarity concepts in psychology and computer science has been discovered [87], differentiating the *human* concept of similarity (used by psychologist) and the *computer* concept of similarity (used by computer scientists). For the discussion about various similarity theories, we refer to the classical work [104] and to a more recent work [87].

Suppose that a collection (or database) $\mathcal{C}$ consisting of n documents is represented by a dataset $\mathbb{S}$, which is a subset of universe $\mathbb{U}$ of objects (i.e. $\mathbb{S} \subseteq \mathbb{U}$). We left the universe $\mathbb{U}$ further unspecified but, however, in most cases it is a multidimensional vector space. For the sake of common tasks solved in the scope of Information Retrieval, the similarity between two objects $O_i, O_j \in \mathbb{U}$ can be modelled by a similarity function as follows.

Definition 1.1 (similarity function)

Given a universe $\mathbb{U}$ of objects, a *similarity function* s is defined as $s : \mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}$ where value of s represents a similarity score between two objects in $\mathbb{U}$. $\square$

1.3.1 Similarity Queries

Given a dataset $\mathbb{S} \subseteq \mathbb{U}$ representing a document collection $\mathcal{C}$, the (simplified) similarity search scenario is set as follows:

1. The similarity search provides retrieval of objects similar to a single object, thus the similarity query predicate is represented by a *query object* $Q \in \mathbb{U}$.

2. The relevance score of an object $O_i \in \mathbb{S}$ to a similarity query predicate is determined as a similarity to the query object Q, i.e. the score is $s(O_i, Q)$.

3. Unlike for exact-match queries, where the binary relevance score qualifies whether an object has to appear in the query result or not, a similarity query predicate itself cannot qualify any object of $\mathbb{S}$ to be relevant or irrelevant to a query. Additionally, there must be a *similarity query extent* specified, determining which similarity scores (the objects having such a score respectively) have to appear in the query result.

4. The result of a similarity query is structured – all objects included in the query result are ordered by their similarity scores to the query predicate.

Among several forms of similarity queries, we consider two of them – range queries and k-NN queries:

Definition 1.2 (range query)

Given a query object Q (representing the similarity query predicate) and a minimum similarity threshold α (representing the similarity query extent), the *range* *query* `range(`Q`,`α`,`$\mathbb{S}$`,`s`)` determines all such objects $O_i \in \mathbb{S}$ that $s(Q, O_i) \geq \alpha$, i.e. all the objects, the similarity scores of which (with respect to Q) are not less than α. □

Example 1.4

Let us have a collection of fingerprints. A common range query could provide tasks like: *"Determine every person having his/her fingerprint similar to a pattern fingerprint more than 90%"*. A range query with a high α threshold could provide tasks like: *"Identify a person having his/her fingerprint almost identical to a pattern fingerprint"*.

Definition 1.3 (nearest neighbours query)

Given a query object Q (the query predicate) and an integer $k \geq 1$ (the query extent), the *k-nearest neighbours query (k-NN query)* `NN(`Q`,`k`,`$\mathbb{S}$`,`s`)` determines k such objects $O_i \in \mathbb{S}$ that similarity scores $s(Q, O_i)$ are the k highest. Possible ties are resolved arbitrarily. The special case of 1-NN query is called simply the *nearest neighbour query.* □

Example 1.5

Let us consider a collection of newspaper articles. A common k-NN query could provide tasks like: *"Return 20 articles thematically most similar to a pattern newspaper article"*.

In many applications the similarity search is often combined with a relevance/feedback functionality. For example, by a k-NN query the user retrieves k most similar objects, but the set of retrieved objects is not considered by the user as a convincing result. Therefore, an object from the query result is chosen by the user and the query is repeated with the new query object. The process of query refinement terminates as soon as the user decides the result is sufficient.

1.4 Similarity Search in Metric Spaces

Without a loss of generality, we can turn the similarity function into an analogous dissimilarity function as follows.

Definition 1.4 (dissimilarity function)

Given a universe $\mathbb{U}$ of objects and a similarity function s, a *dissimilarity function* d complementary to s is defined as $d : \mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}$ such that $s(O_i, Q) \leq s(O_j, Q) \Leftrightarrow d(O_i, Q) \geq d(O_j, Q), \forall O_i, O_j \in \mathbb{U}$ and a fixed $Q \in \mathbb{U}$. In other words, for arbitrary two objects $O_i, Q \in \mathbb{U}$ a higher value of $d(O_i, Q)$ implies a lower value of $s(O_i, Q)$ and vice versa. □

Moreover, we restrict to a non-negative dissimilarity function $d : \mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}_0^+$, where the zero dissimilarity score is interpreted as the maximum similarity score (i.e. two objects $O_i, O_j \in \mathbb{U}$ for which $d(O_i, O_j) = 0$ are identical). Obviously, such a restriction is possible for upper-bounded similarity functions only. Finally, we additionally restrict the non-negative dissimilarity function to be a metric (or distance function) as follows.

Definition 1.5 (metric)
A dissimilarity function d is called a *metric* (or a distance function) if the following metric axioms are satisfied $\forall O_i, O_j, O_k \in \mathbb{U}$:

$$\begin{array}{rcllr} d(O_i, O_i) & = & 0 & & \text{reflexivity} \\ d(O_i, O_j) & > & 0 & (O_i \neq O_j) & \text{positivity} \\ d(O_i, O_j) & = & d(O_j, O_i) & & \text{symmetry} \\ d(O_i, O_j) + d(O_j, O_k) & \geq & d(O_i, O_k) & & \text{triangular inequality} \end{array}$$

The couple $(\mathbb{U}, d)$ is called *metric space*. If the metric d is additionally upper-bounded by a maximum distance value d^+, i.e. $d : \mathbb{U} \times \mathbb{U} \mapsto \langle 0, d^+ \rangle$, then $(\mathbb{U}, d)$ is called a *bounded metric space.* □

Among many others, we name several popular metrics:

Example 1.6
The *Minkowski* L_p metrics, defined for D-dimensional vector spaces as

$$L_p(v_1, v_2) = (\sum_{i=1}^{D} |v_1[i] - v_2[i]|^p)^{\frac{1}{p}} \qquad (p \geq 1)$$

The most used cases for finite p are the *Euclidean* L_2 and the *Manhattan* (or city-block) L_1 distances. If $p = \infty$, we obtain the *"Max"* L_∞ metric defined as $L_\infty(v_1, v_2) = max_{i=1}^{n}\{|v_1[i] - v_2[i]|\}$.

Example 1.7
A generalization of Euclidean distance is the *weighed Euclidean distance*, defined as

$$wL_2(v_1, v_2) = \sqrt{\sum_{i=1}^{D} w_i(v_1[i] - v_2[i])^2}$$

where to i-th dimension a weight w_i is assigned, allowing to adjust the significance of each particular dimension. For the classic Euclidean distance all dimensions are equally important, i.e. $w_i = 1$.

Example 1.8
The *quadratic-form distance* function, defined as

$$d_{QF}(v_1, v_2) = \sqrt{(v_1 - v_2)M(v_1 - v_2)^T}$$

where M is a positive definite matrix (positive definite matrix M guarantees that $d_{QF}(v_1, v_2) \geq 0$). Each value M_{ij} quantifies a correlation rate between the i-th and the j-th dimension of the vectors.

In fact, the quadratic-form distance is a direct generalization of the weighed Euclidean distance, for which the matrix M is diagonal and the weights w_i are placed just on the diagonal (i.e. $M_{ii} = w_i$). For the classic Euclidean distance the weights w_i are equal to one, thus in such case M is a unitary diagonal matrix.

Example 1.9
The *Levenshtein metric* [61] (edit distance) over strings $d_E(s_1, s_2)$, which counts the minimum number of string edit operations (character insertions, deletions and replacements) needed to transform string s_1 into s_2. Another string metric, the *Hamming distance*, counts the number of non-matching characters on the corresponding positions in both strings. We can also view the Hamming distance as a special case of the Levenshtein metric, where only the replacement operation is supported (i.e. character insertions/deletions are not allowed).

Example 1.10
The *normed overlap distance* (also known as the *Jaccard coefficient*), measuring set similarity, defined as:

$$d_{NO}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Example 1.11
The *tree edit distance* [91] $d_{TE}(T_1, T_2)$ over trees T_1, T_2 (e.g. XML trees), which counts the minimum number of tree edit operations (node deletion, insertion and relabeling) needed for transformation of T_1 into T_2.

Time Complexity

Some metrics can be considered as computationally cheap, e.g. the L_p metrics, which are computed in linear time (according to the dimensionality of compared vectors). On the other side, we understand some metrics as expensive, e.g. the Levenshtein metric implemented by dynamic programming[3], being of quadratic complexity with the number of characters in strings. The quadratic-form distance is also of quadratic complexity (due to vector-by-matrix multiplication), and the tree edit distance is even of complexity $O(n^4)$.

[3]The edit distance can be also computed in linear time by a deterministic finite automaton [63, 53, 54]. Instead of quadratic time complexity, this method requires sub-exponential space (subexponential time for the automaton construction, respectively).

1.4.1 Metric Queries

Nowadays, the majority of similarity search applications is based on metric search, i.e. the similarity function is modelled by a metric. In such a scenario, the similarity queries have to be slightly modified in order to obtain *metric queries* providing the same semantics.

Definition 1.6 (metric range query)
Given a query object Q and a *query radius* r_Q, the *range query* `range(`Q`,`r_Q`,`$\mathbb{S}$`,`d`)`, or simply (Q, r_Q), determines all such objects $O_i \in \mathbb{S}$ for which $d(Q, O_i) \leq r_Q$, that is, all the objects the distances of which (with respect to Q) are not greater than r_Q. A special case for $r_Q = 0$ is called the *point query*. □

The geometric interpretation of range query is a hyper-spherical[4] metric region centered in Q and of radius r_Q. All the objects in the dataset $\mathbb{S}$ located inside the query hyper-sphere are returned as a query result.

Definition 1.7 (metric k-nearest neighbours query)
Given a query object Q and an integer $k \geq 1$, the *k-nearest neighbours query* `kNN(`Q`,`k`,`$\mathbb{S}$`,`d`)`, or simply (Q, k), determines k such objects $O_i \in \mathbb{S}$ that distances $d(Q, O_i)$ are the k smallest. Possible ties are resolved arbitrarily. □

The geometric interpretation of a k-NN query is similar to that of range query. The difference is that r_Q is not known in advance, it is dynamically adjusted during the query processing.

Example 1.12
In Figure 1.1 several points O_i in two-dimensional vector space and six range query regions (having the same query object and radius) are depicted for L_1, L_2, L_∞, L_5, weighed Euclidean and quadratic-form distances. For other L_p metrics ($p > 2$), the query regions look like squares with rounded corners. The higher p, the sharper corners and vice versa.

The geometric difference between regions for Euclidean, weighed Euclidean and quadratic-form distances is that for Euclidean distance it is a hyper-sphere, for weighed Euclidean distance it is an iso-oriented hyper-ellipsoid and for quadratic-form distance it is an arbitrarily oriented hyper-ellipsoid.

Note: For non-vector datasets, the geometric interpretation of metric regions is complicated, nevertheless, there are ways how to embed a metric dataset into a vector space or even into Euclidean vector space (see Section 2.5.2).

[4]The strict definition of term "hyper-sphere" is related to Euclidean vector spaces. Nevertheless, for the lack of terminology we relax the meaning of "hyper-sphere" to general metric spaces, representing a region defined by a center object $O_i \in \mathbb{U}$ and bounded by a radius r_{O_i} (e.g. for L_1 metric it is a "hyper-diamond", for L_2 it is a real hyper-sphere, and for L_∞ it is a hyper-cube).

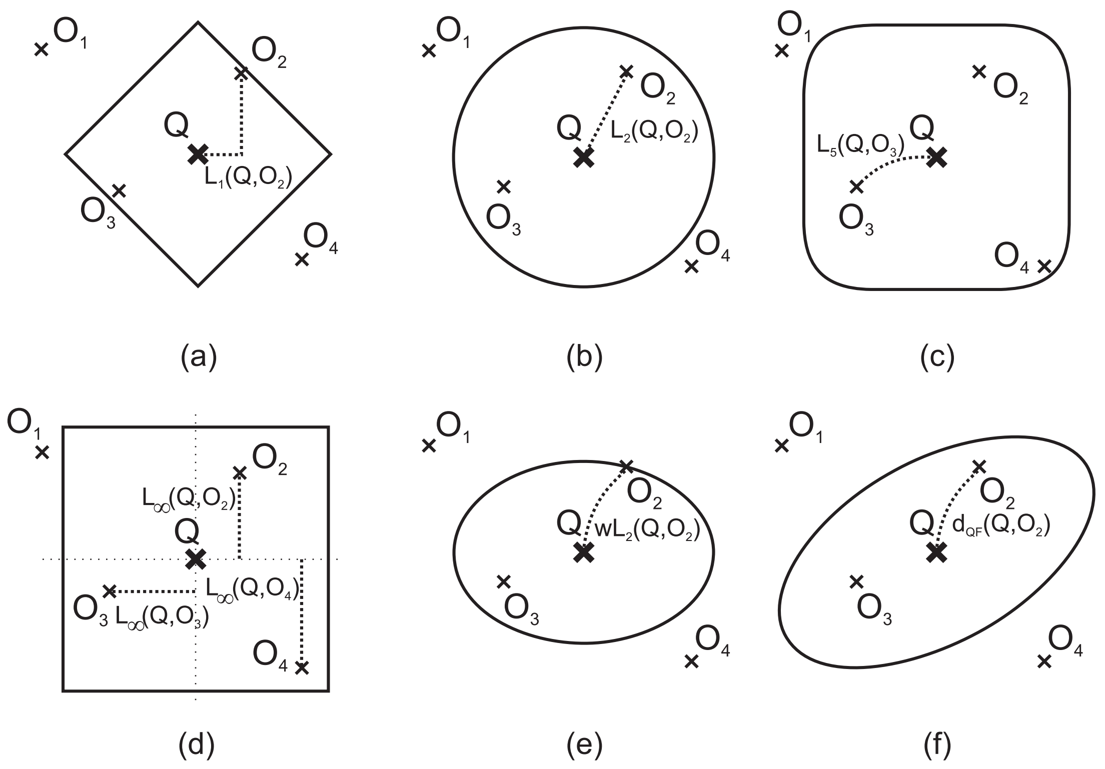


Figure 1.1 Range query regions for:
(a) Manhattan distance (L_1) (b) Euclidean distance (L_2)
(c) L_5 metric (d) L_∞ distance
(e) weighed Euclidean distance (f) quadratic-form distance

1.4.2 Limitations of the Metric Search

Since the similarity functions (dissimilarity functions actually) are modelled by metric functions, there arise several conceptual problems (mentioned in the works cited at the beginning of Section 1.3) related to the metric axioms, as follows.

Reflexivity

The reflexivity axiom has been refuted by similarity theories proposing that a self-similarity of an object (stimulus respectively) also depends on the density of objects in neighbourhood of the respective object.

Positivity

The positivity was questioned by theories of perception. The theories emphasize that sometimes an object is identified as another object more frequently than it is identified as itself.

Symmetry

A number of theories refute the symmetry axiom, showing asymmetries between objects where a less salient (less important in some sense) object is more similar to a more salient object than vice versa.

Triangular Inequality

The triangular inequality is the most attacked condition. It is, however, the fundamental property that allows us to organize the dataset $\mathbb{S}$ efficiently (as we will see in the following chapters).

There are two aspects of similarity search in Information Retrieval. The human aspect emphasizes that similarity search should be *effective* in sense that it should effectively mimic the human mind of assessing similarity between two objects (two stimuli respectively). On the other side, the computer performance aspect emphasizes the efficiency of similarity search. The search must be *efficient* in sense that the object dataset should be well-organized, in order to evaluate a query quickly.

While the more general similarity (or dissimilarity) functions support the human aspect, the more restrictive properties of metric axioms sustain the computer performance aspect. The computational model of similarity, however, should be both effective and efficient. The trade-off between effectiveness and efficiency is the main reason for considering currently the metric model as sufficient.

1.5 Applications of Metric Search

The similarity search in metric spaces has a plenty of applications in Information Retrieval, in this section we mention a few of them.

1.5.1 Text Retrieval

One of the original problems in Information Retrieval is an effective retrieval of full-text documents. Among many Text Retrieval models, the vector model (and its extensions) uses the cosine measure as a similarity function. Given a document (represented by a vector of term weights in the document) and a query (represented by a vector as well), the cosine of the two vectors' deviation determines a similarity score between the query and the document.

Recently, we have introduced a metric approach [96] to similarity search in the vector model, exploiting a metric analogy to the cosine measure – the deviation metric. For more details about the approach see Chapter 8.

1.5.2 Image Retrieval

Since there exist many interpretations of visual similarity, the subject of general content-based Image Retrieval is very rich. Many methods of feature extraction have been developed so far, and many similarity measures (more or less specific) have been used to model the human perception of similarity. Due to the lack of space, we outline only several approaches here, for a comprehensive survey we refer to [85].

Color histograms

The most frequent type of image feature extraction is based on usage of color histograms. In the simpler case, the histogram is extracted for the whole image, a more sophisticated way is to segment the image and create a histogram for each of the segments [26]. The feature vectors are classified usually according to the quadratic-form[5], Manhattan or Euclidean distance functions. Another type of color feature extraction is a color correlogram [55], a histogram representing spatial correlations of color changes considering the pixel distances. For similarity measuring of correlograms, the Manhattan, Euclidean and quadratic-form distances have been used.

Textures

A texture (i.e. a homogeneous pattern in image) can be represented by a histogram as well. For a texture histogram representation, the relative brightness of pixel pairs is computed such that degree of contrast, regularity, coarseness and directionality can be estimated [102]. The texture feature vectors are classified by the same metrics as the color histograms.

Shape

For a similar-shape retrieval the points of interest in an image are recognized, representing locations in the image where the pixel brightness is in contrast with the brightness of neighbouring pixels. Such points usually determine corners and edges present in the image. In [62] the shape boundary has been coded as an ordered sequence of interest points, while the Euclidean distance has been used on them as a dissimilarity measure.

Advanced Statistical Methods

Many sophisticated methods have appeared recently in Information Retrieval, trying to wisely reduce the dimensionality of feature vectors. A kind of dimen-

[5]The quadratic-form distance is widely used [50, 89], since (due to human perception) many colors are considered to mutually correlate.

sionality reduction can be achieved by means of statistical preprocessing of the dataset, so that a basis is formulated for lower-dimensional feature vector representation. In Image Retrieval these methods have been applied to many types of feature vectors, e.g. color histograms [101], or even vectors of all the pixels in image [83]. As a representative domain application, the Karhunen-Loeve Transformation (KLT) (or Principal Component Analysis) has been used for recognition of human faces [92, 57, 78].

1.5.3 Approximate String Matching

The databases of strings are used to collect either strings of natural language or strings modelling a more complex data instances. In the former case, a database can consist of e.g. sentences present in newspaper articles, while the latter case is interesting for modelling e.g. biological data, genomic databases, XML, etc. Besides the well-known Levenshtein (edit) [70, 10, 44] or Hamming distance, there are used domain-specific metrics, e.g. the local sequence alignment [110] in case of genomic databases.

1.5.4 XML Retrieval

With the evolution of native XML databases, there is a need for efficient similarity search in XML documents. In [73] the authors exploit a kind of tree edit distance to evaluate similarity between two XML trees. Recently, we have proposed another approach [58], decomposing the XML database into a set of paths. The distance between two XML paths is defined as a generalized string edit distance, where each string element (character respectively) is represented by an XML element tag.

Part I

Exact Search

Chapter 2

Metric Access Methods

In this chapter we overview several state-of-the-art methods providing *exact metric search*. Such methods search in a given metric dataset exactly, no *false drops* (i.e. objects which are understood as relevant to the query but do not appear in the query result) are tolerated.

Within the scope of Information Retrieval, we focus on searching in large document collections (hundreds of thousands documents and more), hence an efficient similarity search in such collections is necessary. It is obvious that even the most qualitatively *effective* similarity search method is useless, if it is not *efficient*. In order to keep the similarity search efficient as much as possible, there is a need for development of search methods minimizing the overall search costs. The most objective definition of search costs can be determined as the computer time needed to evaluate a query. In case of metric search, the methods should minimize two major components of the overall search costs – the *computation costs* (CC) and the *I/O costs* (or disk access costs). The computation costs represent the number of distance computations needed for a query evaluation. The I/O costs are related to the volume of data needed to be transfered from a secondary memory during a query evaluation.

There have been developed many (more or less efficient) *metric access methods* (MAMs)[1], providing search in general metric datasets. However, most of the current MAMs do not consider the influence of I/O costs on the overall search costs, assuming the time required for a distance computation is disproportionately larger to the time needed for an I/O operation. This may be true in case of a rather small dataset, however, in the context of searching in large collections the I/O costs often grow beyond the computation costs (especially when a cheap metric is used simultaneously). Hence, in order to develop a truly efficient metric access method, we must cope with both types of the search costs.

[1]For an overview on various metric access methods we refer to a comprehensive survey [33].

2.1 Background

The MAMs organize (e.g. partition and/or cluster) the dataset $\mathbb{S}$ among equivalence classes of objects (see Figure 2.1a), such that each class E_i includes objects sharing a common distance property (e.g. E_i consists of objects sufficiently close to each other). The resultant data structure describing the organization is called *metric index*. At this moment, we have to emphasize that the triangular inequality property of a metric d is the only and essential tool used (in various ways) by every MAM for efficient filtering of irrelevant objects.

Each equivalence class E_i is spatially bounded by a region in the metric space (e.g. by a hyper-sphere) spatially covering all the objects in the particular class. The distance function, as the only tool used for partitioning, must satisfy the triangular inequality, so that two metric regions are easy to check for an overlap. Consequently, at query time the metric index is searched by a MAM in order to find those "candidate" equivalence classes, the regions of which overlap the query region. Such candidate classes are exhaustively checked for the relevant objects (see Figure 2.1b). A non-overlapping class contains objects which are surely irrelevant, thus the class can be excluded (or discarded) from further processing.

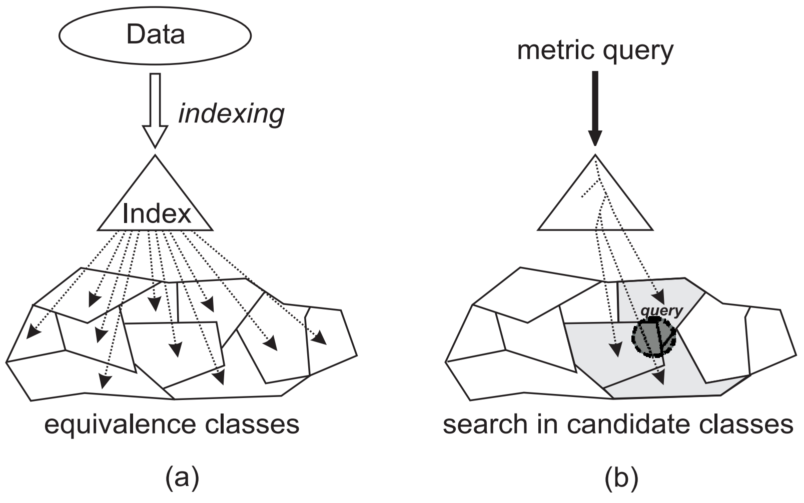


Figure 2.1 (a) Structure of metric index (b) Metric query processing

The region shape of an equivalence class is specific to each particular MAM. Furthermore, different MAMs organize the classes of objects either in a hierarchical or in a flat index structure (or even in a combination of both).

2.2 Spatial Access Methods

Since the dataset $\mathbb{S}$ is often a collection of vectors, the straightforward solution for searching in $\mathbb{S}$ is utilization of a *spatial access method* (SAM) [18] (e.g. R-tree [52], UB-tree [11], X-tree [13], Pyramid-tree [12], etc.). Spatial access methods have been originally developed for indexing vector datasets, in order to provide efficient processing of exact-match queries, including *window queries* (hyper-rectangular query specified by two boundary points), point queries, etc. However, there have been proposed many approaches later, extending the functionality of SAMs in order to provide also the similarity search (metric search respectively).

2.2.1 Metric Search using SAMs

Since most of the SAMs are based on similar principles, we have chosen the R-tree [52] to demonstrate metric search realized by SAMs. The R-tree indexes D-dimensional points by a way of height-balanced hierarchy of nested hyper-rectangles (see an example in Figure 2.2).

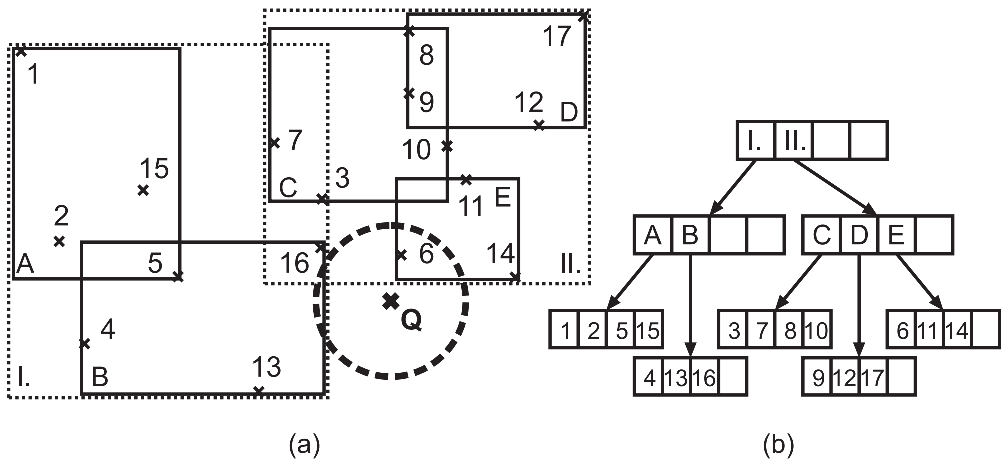


Figure 2.2 (a) Hierarchy of nested MBRs
(b) Appropriate R-tree

The leaf nodes of R-tree represent the equivalence classes of objects, while the inner nodes are used to recursively cluster all the classes. Each node is associated with a *minimum bounding rectangle* (MBR), in order to determine the region in which the objects stored in descendant leaves are located. The R-tree construction algorithms keep the MBRs to cover only those parts of space containing the data objects. Since MBRs at the same R-tree level may overlap, even a simple point query may lead to multiple search paths. This fact, however, negatively affects efficiency, introducing additional search costs. In order to minimize search costs, the R-tree construction algorithms keep the total volume of MBRs minimized, thus smaller MBRs have a lower probability to be accessed.

For a range query region (Q, r_Q), we have to retrieve such objects stored in leaves that overlap the query region. At each level of R-tree, only those nodes are accessed, the MBRs of which overlap the query region (see Figure 2.2).

2.2.2 Major Drawbacks

Most of the SAMs build their indices to primarily support exact-match queries (e.g. window queries or point queries). Concerning this, the usage of SAMs for similarity search is limited by several drawbacks:

1. The objects in an equivalence class (e.g. in an R-tree leaf) are not clustered according to a metric we would like to use for metric search. As an example, consider an R-tree leaf the MBR of which is stretched over the whole domain extent in one dimension, but in the other dimensions the MBR ranges are small or constant. Such a "linear" MBR is surely of small volume, however, two objects in the MBR, each on the opposite side of the range, can be very distant (e.g. considering any L_p metric).

2. Since SAMs generally do not cope with distance computations, they are designed to minimize only the I/O costs. An optimization is therefore needed, in order to minimize also the number of distance computations.

3. We showed that, in order to retrieve all the objects satisfying a range query, all of the class regions overlapping the query region have to be accessed. The overlap check of such two regions is an easy task if the used metric is a simple one (e.g. the L_2 metric), but this can be very difficult if the metric is complex (e.g. the quadratic-form distance).

4. Finally, SAMs can only index vector spaces, thus they are useless for straightforward indexing of non-vector datasets, i.e. datasets of strings, graphs, sets, etc. Nevertheless, an indirect usage of SAMs for search in general metric spaces is mentioned in Section 2.5.2.

2.3 Methods utilizing Global Pivots

Since in general metric spaces the distance function is the only tool for building the structure of a metric index, an important role in metric indexing play objects called *pivots* (or vantage points), i.e. some objects selected from the dataset to which the other objects are somehow referenced by distance.

In this section we discuss MAMs exploiting a *global* set of pivots, assigned to the entire metric index. We call them *distance matrix methods*, because there is created a matrix of distances between pivots and all of the objects in the dataset.

2.3.1 AESA

The *Approximating and Eliminating Search Algorithm* (AESA) [106, 107] uses an $n \times n$ matrix consisting of distances between all pairs of objects in the dataset. Every object plays the role of pivot. During a range query (Q, r_Q) processing, a pivot $P \in \mathbb{S}$ is randomly chosen. The distance between Q and P is computed and used for discarding such objects O_i, that $|d(O_i, P) - d(Q, P)| > r_Q$. Next, another pivot is chosen and all the remaining non-discarded objects are checked. This process is repeated until the set of non-discarded objects is small enough. The remaining non-discarded objects are compared directly against Q.

Example 2.1

In Figure 2.3 see how the filtering condition is used by processing a range query. Since distances between all the objects and a pivot P are known, such objects can be safely discarded, the distances of which to Q fall outside the interval $\langle d(Q, P) - r_Q, d(Q, P) + r_Q \rangle$ (objects located outside the grey area).

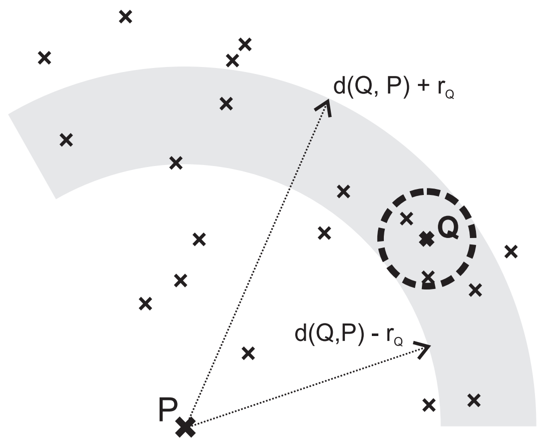


Figure 2.3 AESA/LAESA filtering

The distance computation search costs are reported to be an order of magnitude lower than by other competing MAMs. However, the construction costs and also the I/O search costs are of quadratic complexity $O(n^2)$, thus AESA is applicable only for search in small datasets of at most few thousands objects.

2.3.2 LAESA and Modifications

The drawback of AESA, being quadratic in space and I/O costs, is solved by *Linear AESA* (LAESA) [65, 66]. This method uses p fixed pivots, so that space costs and time needed for construction of the $p \times n$ distance matrix are $O(pn)$. The methods of selection an optimal set of pivots are well-known [90, 7, 28] while, in general, we can say that a set of pivots is sampled such that distances among pivots are maximal (for methods of optimal pivot selection see Section **??**).

In the first step, some irrelevant objects are discarded using all the pivots (the same way as AESA does it). The larger set of pivots is used, the greater proportion of irrelevant objects is discarded. The distance computation search costs are reported to be $p + O(1)$, while the I/O search costs remain $O(pn)$.

The I/O search costs are reduced by a modification called *TLAESA* [64], which (instead of a sequential index) builds a GH-tree-like structure (for GH-tree see the next section) using the same set of p pivots. The I/O search costs for TLAESA have been argued to be between $O(log\ n)$ and $O(pn)$. Another improvement of LAESA is *Spaghettis* [29], which maintains p arrays A_j, one for each pivot P_j. An array A_j contains n object-and-distance pairs (id(O_i), $d(P_j, O_i)$), sorted according to the distance. During a range query processing, each array A_j is queried for a distance interval $\langle d(Q, P_j) - r_Q, d(Q, P_j) + r_Q \rangle$. In order to obtain the candidate objects, an intersection of all the sets of objects retrieved from the arrays is performed. Finally, the remaining objects are filtered as usual. The authors of Spaghettis report I/O search costs reduction to $O(p\ log\ n)$.

2.4 Methods utilizing Local Pivots

Most of the MAMs exploit *local* pivots, which means that to local pivot (to a combination of several local pivots respectively) only a part of the index is referenced. For dynamic MAMs the local pivots are often dynamic as well, because they are selected when (a part of) the metric index is created or altered.

2.4.1 GH-tree

The *generalized-hyperplane tree* (GH-tree) [105] is a binary tree built recursively in top-down fashion as follows. Two local pivots O_i and O_j are selected, while objects closer to O_i are assigned to a "left" subset and those closer to O_j are assigned to a "right" subset (see an example in Figure 2.4). The partitioning of subsets continues recursively, so that we obtain an unbalanced binary tree.

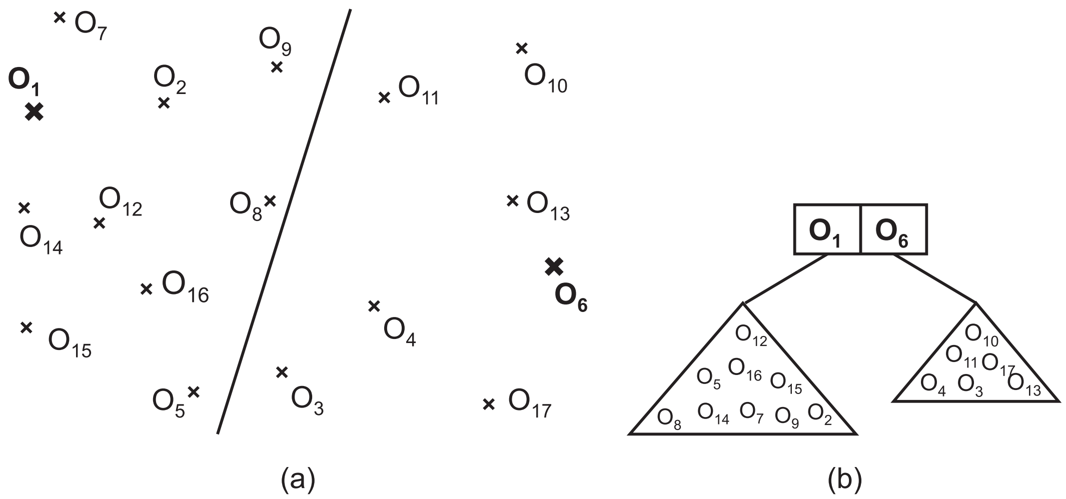


Figure 2.4 (a) Dataset partitioned by a generalized hyper-plane
(b) Root level of GH-tree

During a range query processing we compute $r_1 = d(Q, O_i)$ and $r_2 = d(Q, O_j)$ for each node being processed. The left subtree is processed if $r_1 - r_Q < r_2 + r_Q$ while the right subtree is processed if $r_2 - r_Q \leq r_1 + r_Q$. It is possible to process both subtrees. The costs analysis we present for the more general GNAT. Similar as GH-tree are the *Bisector Tree* [56, 76, 77] and the *Voronoi Tree* [42, 75].

2.4.2 GNAT

The *geometric near-neighbour access tree* (GNAT) [22] is a direct generalization of GH-tree to p-ary tree. Given a dataset $\mathbb{S}$, p local pivots $O_1, \ldots, O_j, \ldots O_p$ are selected. Objects closest to O_j are assigned to the j-th subset (see an example in Figure 2.5). The partitioning of subsets continues recursively, so that we obtain an unbalanced p-ary tree. Besides p-ary partitioning principle, GNAT stores also an information about distances between pivots and objects in subtrees. In each internal node a $p \times p$ matrix is stored, consisting of distance ranges.

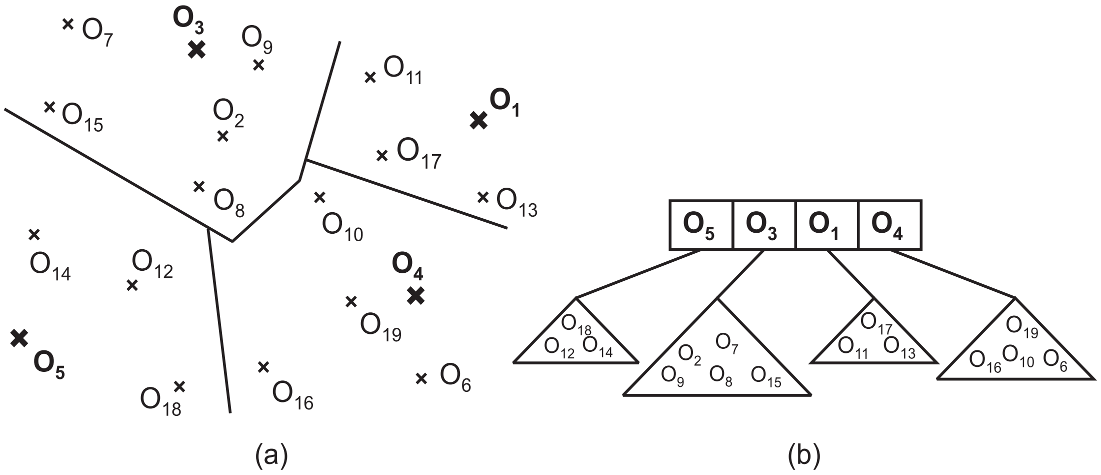


Figure 2.5 (a) Dataset partitioned by generalized hyper-planes
(b) Root level of GNAT using 4 pivots

The query processing in GNAT follows the idea of GH-tree, some additional computation costs are saved due to the stored distance ranges. The I/O construction costs (and also space costs) are $O(np^2)$, the distance computation construction costs are $O(np\ log_p\ n)$. The search costs have not been analyzed by authors, however, the experiments in [22] have shown that GNAT outperforms GH-tree and VP-tree.

2.4.3 VP-tree

The *vantage-point tree* (VP-tree) [112] is based on recursive top-down decomposition of the metric space among hyper-spherical cuts around a local pivot (see an example in Figure 2.6). In m-way VP-tree each internal node has a format $[O_i, [\mu_1, \ldots, \mu_{m-1}], [\mathrm{ptr}_1, \ldots \mathrm{ptr}_m]]$, where O_i is the pivot, and μ_j are *cut-off* distances used to partition the space into m hyper-spherical cuts, so that all the objects in dataset are distributed among m equivalence classes of (almost) equal cardinalities. The classes are furthermore recursively partitioned – ptr_j are pointers to the child nodes of VP-tree.

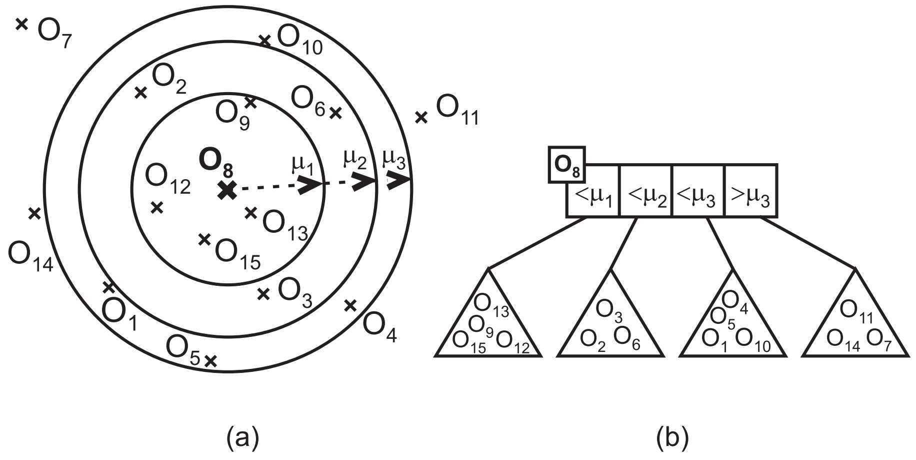


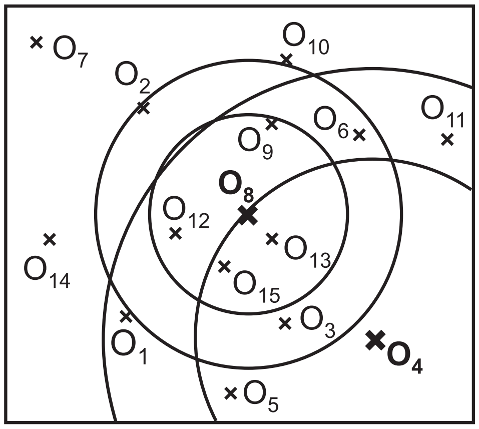
Figure 2.6 (a) Dataset partitioned by 4 spherical cuts
(b) Root level of 4-way VP-tree

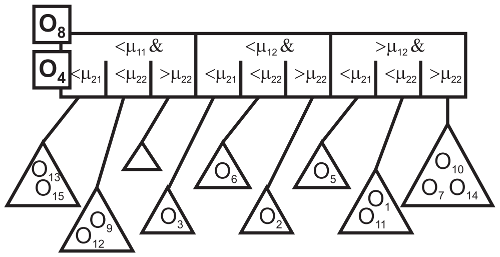
The algorithm for a range query recursively searches the overlapping nodes of VP-tree, i.e. those child nodes of a node $[O_i, [\mu_1, \ldots, \mu_{m-1}], [\mathrm{ptr}_1, \ldots \mathrm{ptr}_m]]$, pointed by ptr_{j+1}, for which $d(Q, O_i) + r_Q \geq \mu_j \wedge d(Q, O_i) - r_Q < \mu_{j+1}$ is satisfied. The search costs are argued to be $O(log\ n)$, but this is true only for very small query radii.

2.4.4 MVP-tree

A generalization of VP-tree is the *multi-vantage-point tree* (MVP-tree) [20, 21], where in each node the space is partitioned by more than one pivot. In a node of an m-way MVP-tree, each of the p pivots divides the space into m hyper-spherical cuts using $m-1$ cut-off values. The combination of all the cuts forms m^p equivalence classes in the space, while each class is assigned to a child node which is furthermore recursively partitioned (see an example in Figure 2.7). Moreover, an extra information is kept in the leaves for even more effective discarding of irrelevant objects.

As for the VP-tree, the search costs of MVP-tree are $O(p\ log_{m^p}\ n)$ but this is true only for very small query radii. The authors show [21] that MVP-tree outperforms the VP-tree, while a larger improvement is achieved when more pivots is used instead of increasing m.





(a) (b)

Figure 2.7 (a) Dataset partitioned by 2 pivots, each sub-partitioning among 3 hyper-spherical cuts
(b) Root level of 3-way 2-pivot MVP-tree (9-ary tree)

2.4.5 M-tree

The M-tree [37] is a dynamic and balanced metric tree, providing an efficient secondary memory management. The M-tree maintains a hierarchy of hyper-spherical clusters in a paged tree structure, conceptually introduced by B$^+$-tree. As a basis of our research, we describe the M-tree in the following chapter.

2.4.6 VP-forest

The *excluded-middle-vantage-point forest* (VP-forest) [111] is another member of the family of VP-trees. The VP-forest is based on a ball partitioning technique, where the middle-distant objects are excluded from partitioning. The dataset is partitioned among three sets S_1, S_2, X, such that each object O_i of the dataset is assigned to the appropriate set using a function:

$$bp(O_i, O_j, d_m, \rho) = \begin{cases} S_1 & \text{if } d(O_i, O_j) \leq d_m - \rho \\ S_2 & \text{if } d(O_i, O_j) > d_m + \rho \\ X & \text{otherwise} \end{cases}$$

where O_j is a fixed pivot, d_m is a medium distance, and ρ is a splitting parameter. The set X is called *exclusion set*, and it contains objects having the middle distances from the pivot. The sets S_1, S_2 are recursively repartitioned, so that we obtain a balanced binary tree. All the exclusion sets X_k created by the recursive partitioning are unified into a single exclusion set, and this set is repartitioned by another tree (using different parameters O_j, d_m, ρ of the partitioning function). The trees are constructed until the last exclusion set becomes empty (or sufficiently small). The trees are linked, forming a forest, see an example of VP-forest in Figure 2.8.

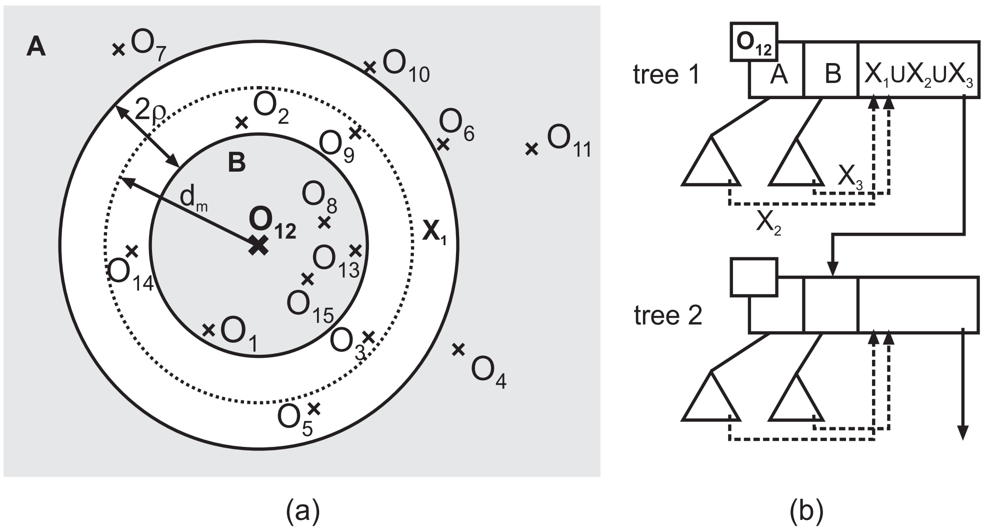


Figure 2.8 (a) Excluded middle partitioning (b) VP-forest

During a range query processing, the idea of excluding the middle distances eliminates the examination of more than one branch of a particular tree if the query radius is less than ρ. The next tree in VP-forest is searched only if the exclusion set of the preceding tree must be visited.

The I/O construction costs are $O(n)$ while the distance computation construction costs are $O(n^{2-\alpha})$, where $O(n^{1-\alpha})$ is the number of trees in the VP-forest. Queries are answered in $O(n^{1-\alpha}\log\, n)$ distance computations. The parameter $0 < \alpha < 1$ depends on ρ, the dataset, and the distance function. Unfortunately, to achieve a greater value of α the parameter ρ has to be quite small.

2.4.7 D-index

Recently, an approach based on *external metric hashing* was proposed, called *D-index* [46]. The dataset is partitioned (as in case of VP-forest) by excluded middle ball partition functions[2] (here called *ball partitioning ρ-split functions*) $bps^{1,\rho,j}$, defined as:

$$bps^{1,\rho,j}(O_i) = \begin{cases} 0 & \text{if } d(O_i, P_j) \leq d_m - \rho \\ 1 & \text{if } d(O_i, P_j) > d_m + \rho \\ 2 & \text{otherwise} \end{cases}$$

where P_j is a pivot assigned to the function $bps^{1,\rho,j}$, ρ is a splitting parameter, and d_m is a medium distance. When combined k such functions (and k pivots),

[2]Several different types of first-order separable ρ-split functions are proposed, analyzed, and evaluated in [45].

we obtain a complex hashing function $bps^{k,\rho}$ partitioning the dataset among 2^k partitions and one exclusion set. For each indexed object a hash key of its target partitions is computed as a combination of binary values $0, 1$ returned by particular $bps^{1,\rho,j}$ functions. In case that (at least) one 2 is returned by a $bps^{1,\rho,j}$ function, the object is assigned to the exclusion set. In simple words, the exclusion set stands for a "border territory" separating the partitions, so that objects in different partitions can be easily distinguished during search.

The idea of D-index is simple. A h-level hashing table is created, such that 2^{k_i} buckets are maintained at the i-th level (k_i is the number of pivots used at i-th level), where every bucket corresponds to one partition, and is accessible by the i-th-level hashing function $bps_i^{k_i,\rho}$. For the objects hashed into the i-th-level exclusion set, the $i+1$-th level of the table is created and the remaining objects are repartitioned by function $bps_{i+1}^{k_{i+1},\rho}$. The last level consists of a single bucket belonging to the exclusion set of the entire D-index. For different D-index levels, the hashing functions $bps_i^{k_i,\rho}$ can vary in the number of ρ-split functions and, consequently, in the number of pivots.

Example 2.2

See Figure 2.9a,b for an example of partitioning the dataset by D-index. The entire dataset is partitioned using a 2-pivot hashing function among buckets A_1, B_1, C_1 and an exclusion set X_1. At the second level, the exclusion set X_1 is repartitioned using a single-pivot hashing function between buckets A_2, B_2 and an exclusion set X_2. At the third level of D-index, the rest of objects is stored in an exclusion bucket X. The physical organization of buckets is presented in Figure 2.9c. Since the number of objects stored in various buckets is different, the buckets are aligned into linked blocks (disk pages respectively).

A substantial advantage of D-index is that for range queries having $r_Q \leq \rho$ only one bucket per level and, possibly, the exclusion bucket are accessed (see Figure 2.9a), reducing the average search costs to $O(1)$. Unfortunately, the ρ value applicable for effective partitioning is usually very small, thus range queries with reasonably high r_Q must search in multiple buckets. Another advantage is that D-index has low construction/storage requirements, it needs only a little larger space than a simple sequential file.

A particular drawback is that D-index is an unbalanced data structure, the buckets are of various sizes (each bucket is physically stored within a number of blocks of fixed size), and therefore, when inappropriate pivots (or parameters ρ, d_m) are chosen, the search in a single bucket can deteriorate to sequential scan over a large part of the dataset. The unbalance property is critical especially for the exclusion bucket where most of the queries must search.

The authors report the D-index beats other MAMs in I/O search costs, which is a consequence of its low storage requirements. In case of point queries and

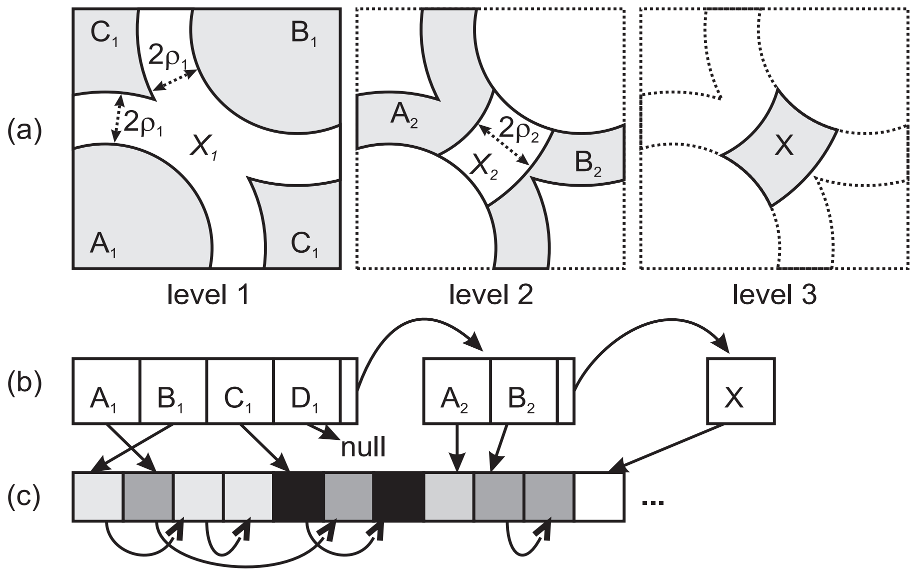


Figure 2.9 (a) Multi-level partitioning
(b) Buckets of D-index assigned to the partitions
(c) Physical organization of buckets of various sizes

range queries where $r_Q \leq \rho$, the overall search costs are very low, for $r_Q > \rho$ the distance computation search costs are comparable to those of M-tree.

In order to speedup similarity joins on D-index, the authors have proposed *eD-index* [47]. Two algorithms for similarity self-joins have been introduced, while significant efficiency improvements have been experimentally verified.

2.5 Other Techniques

At the end of our listing, we present two MAMs which do not exploit neither global pivots nor local pivots.

2.5.1 SAT

The *spatial approximation tree* (SAT) [69, 71] does not use pivots for the dataset partitioning, it is rather based on "spatial" approximation heuristics. An object X is selected as the root and connected to such objects $O_i \in \mathbb{S}$ that are closer to X more than to any other object connected to X (see an example in Figure 2.10). Each object connected to X is a root of an appropriate subtree. The SAT was originally designed as static, a dynamic version of SAT was proposed in [72].

The range query algorithm starts in the root and follows a neighbour (i.e. a connected object) that is closest to the query object Q. Since the query radius r_Q is nonzero, we consider that unknown objects are searched with tolerance r_Q, i.e.

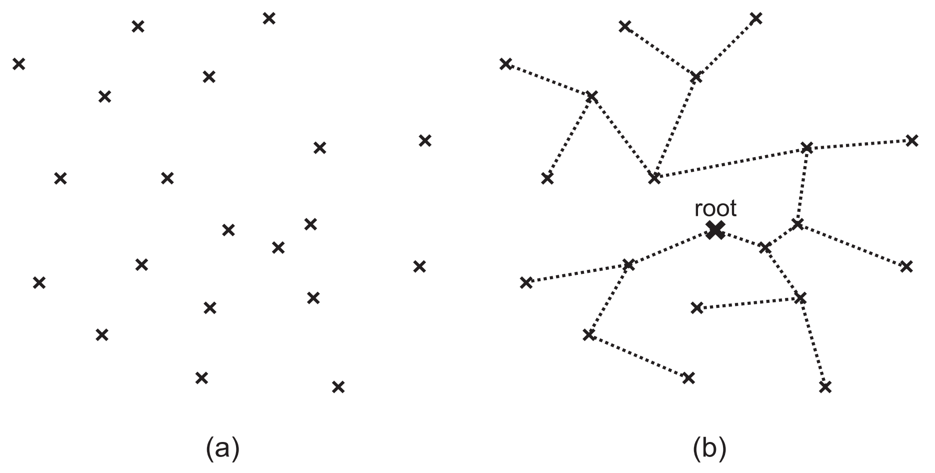


Figure 2.10 (a) Dataset to index
(b) Spatial Approximation Tree

we consider that any distance may have an "error" of at most r_Q. Hence, we may have to follow many branches of the tree, since measuring the "error" could lead to a different neighbour candidating to the closest one. The I/O construction and space costs for the SAT are $O(n)$, the distance computation construction costs are referred to be $O(n\ log\ n/log\ log\ n)$. The search requires to retrieve $\Theta(n^{1-\Theta(1/log\ log\ n)})$ objects.

2.5.2 Metric Mapping

The problem of searching in general metric spaces can be transformed to searching in vector spaces, that is, all objects of the metric space are represented as points in a vector space. Formally, a mapping

$$\Psi : (\mathbb{U}, d) \mapsto (\mathbb{R}^D, \delta)$$

is established, turning the original metric dataset into a D-dimensional vector dataset. In order to preserve distances at least partially, the mapping Ψ is required to be *contractive*, i.e. $\delta(\Psi(O_i), \Psi(O_j)) \le d(O_i, O_j)$, $\forall O_i, O_j \in \mathbb{U}$. In other words, the objects close in metric space $(\mathbb{U}, d)$ are required to be close (or even closer) in the target vector space $(\mathbb{R}^D, \delta)$.

As a consequence, a range query associated with the original metric space can be easily transformed into the target vector space by projecting the query object Q into $\Psi(Q)$. Due to the contractivity of Ψ, the query radius r_Q remains the same as for the original query. Since the original distances have been contracted, the range query $(\Psi(Q), r_Q)$ will always capture the objects qualified by the original query (Q, r_Q). Unfortunately, the query $(\Psi(Q), r_Q)$ can retrieve also *false hits*, i.e. some objects not qualified by the original query (Q, r_Q). Hence, all objects in the result of query $(\Psi(Q), r_Q)$ must be additionally filtered by the original query.

Note: If the metric space $(\mathbb{U}, d)$ is mapped into a vector space of form $(\mathbb{R}^D, L_\infty)$, the dataset $\Psi(\mathbb{S})$ can be organized by a spatial access method providing window queries, like R-tree, UB-tree, etc.

Example 2.3

The LAESA approach (see Section 2.3.2) exploits a mapping of form

$$\Psi : (\mathbb{U}, d) \mapsto (\mathbb{R}^p, L_\infty)$$

The mapping is defined by p pivots P_j such that $\delta(O_i, O_k) = L_\infty(\Psi(O_i), \Psi(O_k))$, where $\Psi(O_i)$ is a vector $[d(P_1, O_i), d(P_2, O_i), \ldots, d(P_p, O_i)]$. The vector $\Psi(O_k)$ is constructed similarly. In Figure 2.11 see an example of LAESA mapping. Note the mapping Ψ is strictly contractive, so that range query $(\Psi(Q), r_Q)$ retrieves also a false hit O_4.

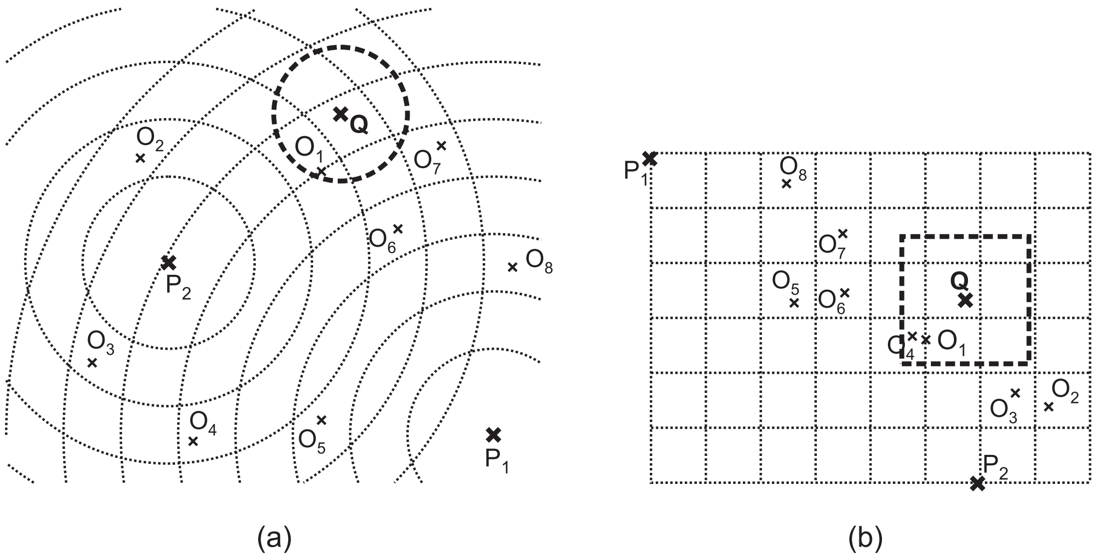


Figure 2.11 LAESA-like mapping $\Psi : (\mathbb{U}, L_2) \mapsto (\mathbb{R}^2, L_\infty)$ utilizing 2 pivots
(a) Dataset in the original metric space $(\mathbb{U}, L_2)$
(b) Dataset mapped into the vector space $(\mathbb{R}^2, L_\infty)$

Example 2.4

A usual reason for embedding metric datasets into vector spaces is a substitution of the original metric d, considered as computationally expensive, by a (much) simpler metric δ. An approximate embedding of text strings (according to the *edit distance with moves*) into a L_1 vector space is described in [40], reducing the complexity problem of string matching from $O(n^2)$ to sub-quadratic.

Metric Multidimensional Scaling

A statistical approach to metric mapping is the *metric multi-dimensional scaling* (MDS) [19]. Given an object-to-object distance matrix, the aim is to construct a map in Euclidean space that corresponds to the original distances[3] (i.e. Ψ^{-1} must be contractive as well), so that no false hits appear. In general, the distance matrix is decomposed (e.g. using the principal component analysis (PCA)), such that eigenvalues and eigenvectors are obtained. The original objects of metric space are represented by linear combinations of the eigenvectors. The number of nonzero eigenvalues determines the (smallest) dimensionality of a vector space, in which the original distances are preserved by Euclidean distance. Unfortunately, the MDS is computationally expensive and static, which limits its applicability.

2.6 Methods for Discrete Distance Functions

All of the presented MAMs suppose that a general continuous[4] distance function is used. However, there have been proposed also methods assuming that a given distance function is discrete, or more precisely, assuming the function returns only a small set of different values.

Example 2.5
The Levenshtein metric can be considered as a discrete distance function. Suppose a collection of strings, where the length of each string is limited by 20 characters. Then the maximum distance between two strings is 20 (20 character replacements), thus the set of all possible distances is of cardinality 20.

Since we are interested in MAMs supporting general continuous distance functions, we only name the most known "discrete" MAMs: Burkhard-Keller Tree [23], Fixed-Queries Tree [7], Fixed-Height Fixed-Queries Tree [6, 7, 8], Fixed-Queries Array [30]. For a detailed overview we refer to [33].

2.7 Summary

All the presented MAMs have been (more or less) successfully utilized in various applications of metric search, as well as for many similarity search scenarios. However, we are concerned by MAMs applicable in Information Retrieval, thus

[3]The object-to-object distances can be even dissimilarities, so that MDS provides a general way how to "metricate" a non-metric dataset.

[4]Actually, any distance function cannot be represented as a really continuous function in computer. Since the set of distances is always of finite cardinality, every distance function represented in computer is discrete. As a continuous function we rather consider a metric with huge number of possible distance values, say at least 2^{32} (e.g. cardinality of four-byte float).

we require the following properties (well-known in the area of database indexing) to be supported:

1. **Efficiency**
   Since the typical size of a large document collection is in order of millions of objects, a MAM should perform efficiently also when the size of dataset grows. In particular, depending on the dataset size $n = |\mathbb{S}|$, we require the index construction costs to be *sub-quadratic* (i.e. lower than $O(n^2)$) and the search costs to be *sub-linear* (i.e. lower than $O(n)$). Naturally, we must take into account both the I/O costs as well as the computation costs.

2. **Dynamicity**
   A MAM has to support dynamic insertions/deletions (of sub-linear construction costs) of objects to/from dataset.

3. **Secondary Memory Management**
   Due to large volumes of data, it is impossible to store the entire dataset in main (or primary) memory. The method, therefore, should be able to efficiently exploit secondary storage devices.

4. **Data Independence**
   The method should behave efficiently for all reasonable data distributions. Furthermore, the order in which the data objects are consecutively inserted into the index (if dynamic MAM) should not have a significant influence on the overall search efficiency. In particular, a metric index should be maintained balanced, and the user should not have to specify any internal parameters of the MAM explicitly.

In Table 2.1 the overviewed MAMs are classified according to the above mentioned properties. If a particular MAM supports a particular property, it is denoted with the ★ symbol in the table (the ⋆ symbol denotes the property is satisfied only partially). We do not consider SAMs and the M$^+$-tree, because they can index only vector datasets. Moreover, we do not consider the idea of metric mapping – actually, the methods of metric mapping are not standalone MAMs, they rather provide a formal framework for projection of a metric space into a vector space. As the table shows, the only MAM supporting all the properties is the M-tree, followed by the D-index. As a basis of our research, the M-tree is described in the following chapter.

MAM	Dyna-	Constr.costs $< O(n^2)$		Search costs $< O(n)$		Sec.mem.	Data
	micity	I/O	CC	I/O	CC	managem.	indep.
AESA				★	★	★	★
LAESA	★	★	★		★	★	★
TLAESA		★	★	★	★		★
Spaghettis		★	★	★	★	★	★
GH-tree	★	★	★	★	★		
GNAT	★	★	★	★	★		
VP-tree		★	★	★	★		★
MVP-tree		★	★	★	★		★
VP-forest		★	★	★	★		⋆
D-index	⋆	★	★	★	★	★	⋆
M-tree	★	★	★	★	★	★	★
SAT	★	★	★	★	★		

Table 2.1 Applicability of MAMs to IR

2.8 Open Problems

So far, there has been given only a marginal attention to development of metric access methods searching according to either a *special metric* (satisfying an additional restrictive condition, besides the basic metric axioms) or, on the other hand, a *non-metric* (relaxing one or more of the metric axioms).

2.8.1 Special Metrics

All of the existing kinds of metrics satisfy the triangular inequality property. However, there exist special types of metrics[5], conditioned by an additional restrictive inequality describable by a first-order characterization, like:

- the widely used *ultrametrics*, satisfying the *ultrametric inequality*

$$d(O_i, O_j) \leq max\{d(O_i, O_k), d(O_j, O_k)\}$$

- the *four-point metrics*, satisfying the *four-point inequality*

$$d(O_i, O_j) + d(O_k, O_l) \leq max\{d(O_j, O_k) + d(O_i, O_l), d(O_i, O_k) + d(O_j, O_l)\}$$

- the *hypermetrics*, satisfying the *hypermetric inequality* defined for each $b : \mathbb{U} \mapsto \mathbb{Z}$ such that $\sum\{b(O_i) : O_i \in \mathbb{U}\} = 1$ as

$$\sum\{b(O_i)b(O_j)d(O_i, O_j) : O_i, O_j \in \mathbb{U}\} \leq 0$$

[5]For a comprehensive survey on various kinds of metrics we refer to [108].

- negative-type metrics, pentagon metrics, spherical metrics, etc.

The additional restrictive properties of such metric types might be applied to the existing (or completely new) MAMs, allowing to provide even more efficient metric search.

2.8.2 Non-Metrics

From another point of view, the metric axioms are sometimes too restrictive in order to model the dissimilarity function adequately to human needs. In particular, in various areas the triangular inequality seems to be a major obstacle for modelling a similarity measure by metric.

For similarity search requirements, there would be useful to develop access methods for indexing *semi-metric* datasets[6] – i.e. relaxing the triangular inequality to a weaker condition or even omitting it – or *quasi-metric* datasets – i.e. neglecting the symmetry property. Naturally, the less restrictive properties a dissimilarity function satisfies, the less information is available for the indexing and, consequently, the less efficient search in such a dataset is possible. Nevertheless, "good" (in some sense) dissimilarity distributions of non-metric functions could compensate the lack of information provided by the metric axioms, thus an efficient dissimilarity organization may be possible – but this is an open question.

Nowadays, general semi-metric and/or quasi-metric datasets can be searched just by sequential scanning the entire dataset, combined with various heuristic techniques (e.g. the usage of inverted file for vector query processing in Text Retrieval).

[6]We partially address the problem of approximate semi-metric search in Chapter 7.

Chapter 3

M-tree

The M-tree [37, 80] organizes objects of a metric dataset $\mathbb{S}$ in a balanced, paged, and dynamic tree, exploiting the idea originally established by the well-known B$^+$-tree. The B$^+$-tree has been already used as a basis for development of many spatial indexing structures, including R-tree, UB-tree, X-tree, etc. The M-tree is a similar modification of B$^+$-tree for purposes of indexing metric datasets. We can even say, that organization of a metric dataset by M-tree is, in principle, very similar to organization of a vector dataset by R-tree.

3.1 Structure

The structure of M-tree is based on hierarchical organization of data objects $O_i \in \mathbb{S}$ according to a given metric d. Like other B-tree-based trees, the M-tree structure consists of balanced hierarchy of nodes. The nodes have a fixed capacity and a minimum utilization threshold. The indexed data objects are recursively clustered in hyper-spherical metric regions associated with M-tree nodes.

The inner nodes contain *routing entries* describing the metric regions, while the *ground entries* (stored in leaves) represent the indexed data objects. A routing entry, stored in an inner node, is denoted as[1]:

$$rout(O_i) = [O_i, ptr(T(O_i)), r_{O_i}, d(O_i, \mathrm{Par}(O_i))]$$

where $O_i \in \mathbb{S}$ is a *routing object* (a local pivot actually), $ptr(T(O_i))$ is a pointer to the subtree $T(O_i)$ of $rout(O_i)$ (called *covering subtree*), and r_{O_i} is a *covering radius*. The routing entry $rout(O_i)$ defines a hyper-spherical metric region (O_i, r_{O_i}) bounding all the objects indexed by $T(O_i)$. The precomputed distance $d(O_i, \mathrm{Par}(O_i))$ (where $\mathrm{Par}(O_i)$ is the parent routing object of N) is used for optimizing most of the M-tree algorithms. A ground entry, stored in a leaf, is denoted as:

$$grnd(O_i) = [O_i, oid(O_i), d(O_i, \mathrm{Par}(O_i))]$$

[1]We use the original notation of node entries, introduced in [37, 80].

where $O_i \in \mathbb{S}$ is an indexed data object, $oid(O_i)$ is an identifier of the original DB object (stored externally), and, again, $d(O_i, \mathrm{Par}(O_i))$ is a precomputed distance.

In situations where the type of node entry is not important, we also denote routing entry $rout(O_i)$ or ground entry $grnd(O_i)$ simply as $entry(O_i)$. Wherever needed, $rout_l(O_i)$ denotes a routing entry stored in a node at the l-th level of M-tree.

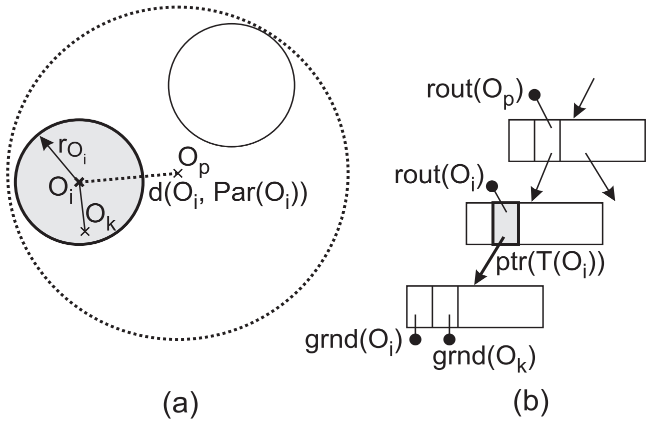


Figure 3.1 (a) Metric region (O_i, r_{O_i})
(b) Routing entry $rout(O_i)$ in the M-tree structure

In Figure 3.1 a metric region and its appropriate routing entry $rout(O_i)$ in an inner node are presented. In order to provide correct discarding of irrelevant subtrees, the M-tree hierarchy is limited by the following *nesting condition*.

Condition 3.1 (nesting condition)
Given a routing entry $rout(O_i)$, all objects stored in the covering subtree $T(O_i)$ must be located inside the region (O_i, r_{O_i}), i.e. $\forall O_j \in T(O_i)$, $d(O_i, O_j) \leq r_{O_i}$. $\square$

If we realize, the nesting condition is very weak, because there can be constructed many M-trees of the same object content but of different hierarchies. The most important consequence is that many regions at the same M-tree level may overlap.

Note: The nesting condition for metric regions in M-tree is similar to that for minimum bounding rectangles in R-tree, see Section 2.2.1.

Example 3.1
In Figure 3.2 a correct M-tree hierarchy is presented for several data objects partitioned among 3 metric regions. Note that region of $rout_1(O_1)$ is "sticking out" the parent region $rout_0(O_2)$ but, nevertheless, the nesting condition is still preserved.

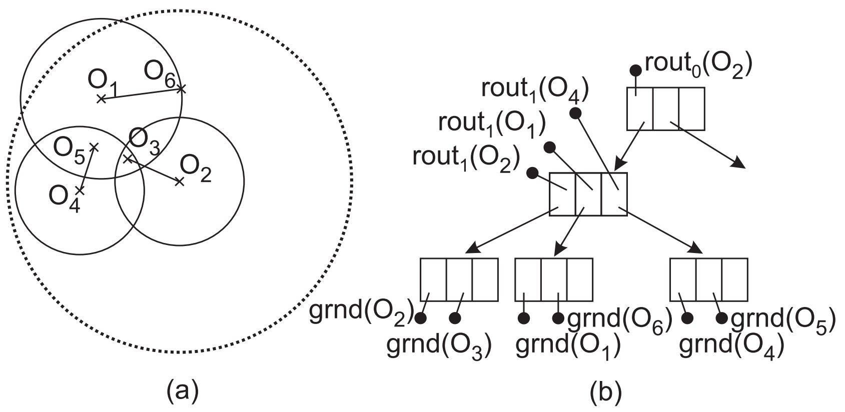


Figure 3.2 (a) Hierarchy of M-tree regions (b) Appropriate M-tree

3.2 Queries

Before we present specific algorithms for building the M-tree, we show how the information stored in the M-tree nodes (routing entries respectively) is used for processing of metric queries. In order to minimize both, the I/O costs and the distance computation costs, the information concerning (precomputed) distances which are stored in M-tree routing entries (i.e. $d(O_i, \mathrm{Par}(O_i))$ and r_{O_i}) is used to apply the triangular inequality property. In order to discard the irrelevant subtrees during a query processing, the two following lemmas are applied.

Lemma 3.1
Let (Q, r_Q) be a hyper-spherical query region, and (O_i, r_{O_i}) be a metric region described by a routing entry $rout(O_i)$. If $d(O_i, Q) > r_{O_i} + r_Q$, then for each object O_j in $T(O_i)$, it is $d(O_j, Q) > r_Q$. Thus, the regions do not overlap, and the subtree $T(O_i)$ can be safely excluded from the search.

Proof: Since $d(O_j, O_i) \leq r_{O_i}$ holds for $\forall O_j \in T(O_i)$, it is

$$\begin{array}{rcll}
d(O_j, O_i) + d(O_j, Q) & \geq & d(O_i, Q) & \text{(by triangular inequality)} \\
d(O_j, Q) & \geq & d(O_i, Q) - d(O_j, O_i) & \\
d(O_j, Q) & \geq & d(O_i, Q) - r_{O_i} & \text{(covering radius upper bound)} \\
d(O_j, Q) & > & r_Q & \text{(by hypothesis)}
\end{array}$$

■

Example 3.2
See the idea of Lemma 3.1 in Figure 3.3. The L_1-spherical region (O_i, r_{O_i}) does not overlap the query region (Q, r_Q) and, therefore, the appropriate $T(O_i)$ surely cannot contain any relevant objects. On the other side, the L_2-spherical region (O_i, r_{O_i}) overlaps the query region, thus the appropriate $T(O_i)$ is relevant to the query, and cannot be excluded from the search.

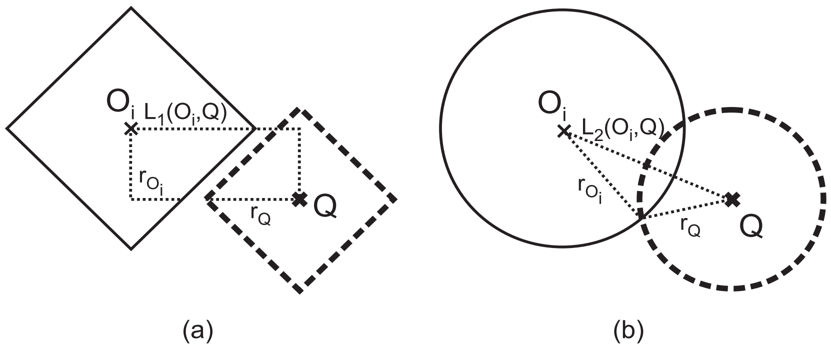


Figure 3.3 (a) Non-overlapped L_1-spherical metric regions
(b) Overlapped L_2-spherical metric regions

In order to apply Lemma 3.1, the distance $d(O_i, Q)$ has to be computed. However, in many cases this can be avoided using the precomputed distances $d(O_i, O_p)$[2] (stored in a routing entry $rout(O_i)$) and $d(Q, O_p)$ (already computed at the previous M-tree level).

Lemma 3.2

If $|d(O_p, Q) - d(O_i, O_p)| > r_{O_i} + r_Q$, then $d(O_i, Q) > r_{O_i} + r_Q$, thus the subtree $T(O_i)$ can be safely excluded from the search.

Proof: This is a direct consequence of triangular inequality, which guarantees that both $d(O_i, Q) \geq d(O_p, Q) - d(O_i, O_p)$ and $d(O_i, Q) \geq d(O_i, O_p) - d(O_p, Q)$ hold. ■

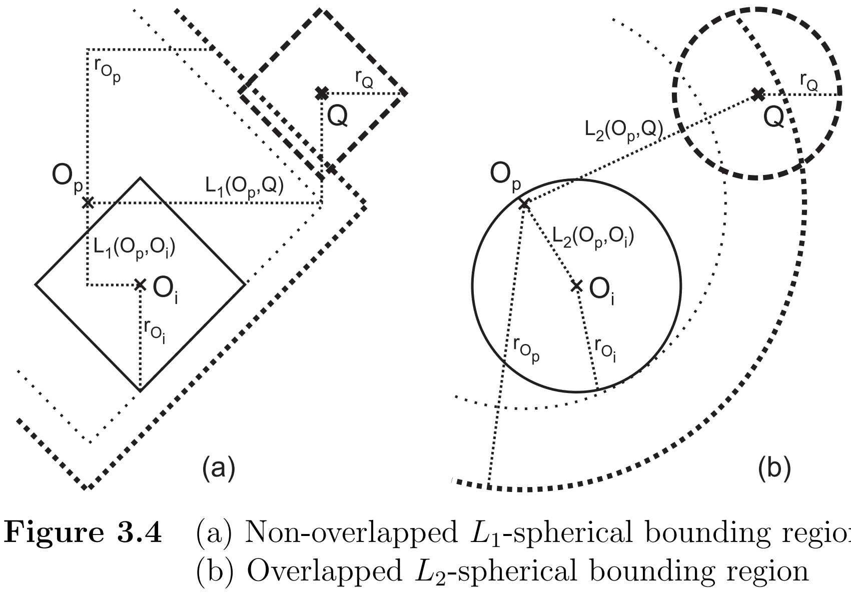


Figure 3.4 (a) Non-overlapped L_1-spherical bounding region
(b) Overlapped L_2-spherical bounding region

[2]Here we denote the parent object $\mathrm{Par}(O_i)$ as O_p.

Example 3.3

The idea of Lemma 3.2 (in case that $d(O_p, Q) - d(O_i, O_p) \geq r_{O_i} + r_Q$) is presented in Figure 3.4. The region (O_i, r_{O_i}) is entirely embedded inside a bounding region $(O_p, d(O_p, O_i) + r_{O_i})$. If the regions (Q, r_Q) and $(O_p, d(O_p, O_i) + r_{O_i})$ do not overlap, we can safely exclude the subtree $T(O_i)$ from search, because also the (O_i, r_{O_i}) region cannot be overlapped (computation of $d(O_i, Q)$ is not needed).

3.2.1 Range Queries

The range query algorithm has to follow all paths in the M-tree leading to objects O_j overlapping the query region (Q, r_Q), i.e. to objects satisfying $d(Q, O_j) \leq r_Q$.

In Listing 3.1 a recursive algorithm of range query is presented. The algorithm is optimal in I/O costs, because only those nodes are accessed, the metric regions of which (defined by parent routing entries) overlap the query region. Application of the above mentioned lemmas is marked in the pseudo-code. Initially, the range query algorithm is executed for the root node.

Listing 3.1 (range query algorithm)

```
QueryResult RangeQuery(Node $N$, RQuery $(Q, r_Q)$)
{
  let $O_p$ be the parent routing object of $N$ /* if $N$ is root then $d(O_i, O_p)$=$d(O_p, Q)$=0 */
  if $N$ is not a leaf then {
    for each $rout(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q + r_{O_i}$ then { /* application of Lemma 3.2 */
        compute $d(O_i, Q)$
        if $d(O_i, Q) \leq r_Q + r_{O_i}$ then /* application of Lemma 3.1 */
          RangeQuery($ptr(T(O_i))$, $(Q, r_Q)$)
      }
    } /* for each ... */
  } else {
    for each $grnd(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q$ then { /* application of Lemma 3.2 */
        compute $d(O_i, Q)$
        if $d(O_i, Q) \leq r_Q$ then
          add $O_i$ to the query result
      }
    } /* for each ... */
  }
} /* RangeQuery */
```

3.2.2 Nearest Neighbours Queries

Realization of the k-NN query algorithm in M-tree is a bit more complicated modification of the range query algorithm. Since the query radius r_Q, i.e. the distance between Q and the k-th nearest neighbour, is not known in advance, it must be determined dynamically during the query processing. For this purpose a *branch-and-bound* heuristic algorithm has been introduced [37], quite similar to that one for R-trees [84]. The k-NN query algorithm utilizes a priority queue PR of pending requests, and a k-elements array NN used to keep the candidates of the nearest neighbours and which, at the end of the execution, contains the result. At the beginning of k-NN query processing, the *dynamic radius* r_Q is set to ∞, while during query processing the radius r_Q is consecutively reduced down to the "true" distance between Q and the k-th nearest neighbour.

PR queue. The priority queue PR of pending requests $[ptr(T(O_i)), d_{min}(T(O_i))]$ is used to keep (pointers to) such subtrees $T(O_i)$, which (still) cannot be excluded from the search due to overlap of their metric regions (O_i, r_{O_i}) with the *dynamic query region* (Q, r_Q). The priority order of each such request is given by $d_{min}(T(O_i))$, which is the smallest possible distance between an object stored in $T(O_i)$ and the query object Q. The smallest distance is defined as the lower-bound distance between Q and the metric region (O_i, r_{O_i}):

$$d_{min}(T(O_i)) = max\{0, d(O_i, Q) - r_{O_i}\}$$

During k-NN query execution, the requests from PR are being processed in priority order, i.e. the request with the smallest lower-bound distance goes first.

NN array. The NN array contains k entries either of form $[oid(O_i), d(Q, O_i)]$ or $[-, d_{max}(T(O_i))]$. The NN array is sorted according to the distance values in ascending order. An entry of form $[oid(O_i), d(Q, O_i)]$ on the j-th position in NN represents a candidate object for the j-th nearest neighbour. On the other hand, the value $d_{max}(T(O_i))$ in an entry $[-, d_{max}(T(O_i))]$ represents an upper-bound distance between Q and objects in subtree $T(O_i)$ (in which some k-NN candidates could be stored). The upper-bound distance $d_{max}(T(O_i))$ is defined as:

$$d_{max}(T(O_i)) = d(O_i, Q) + r_{O_i}$$

Since NN is a sorted array containing the k nearest neighbours candidates (or at least upper-bound distances of the still relevant subtrees), the dynamic query radius r_Q can be determined as the current distance stored in the last entry NN[k]. During query evaluation, only the closer candidates (or smaller upper-bound distances) are inserted into NN array, i.e. those candidates, which are currently located inside the dynamic query region (Q, r_Q).

After an insertion into NN is performed, the query radius r_Q is decreased (because NN[k] entry has been replaced). The priority queue PR must contain only the (still) relevant subtrees, i.e. such subtrees the regions of which overlap the dynamic query region (Q, r_Q). Hence, after the dynamic radius r_Q is decreased, all irrelevant requests (such that $d_{min}(T(O_i)) > r_Q$) must be deleted from PR.

At the beginning of k-NN query processing, the nearest neighbours candidates are unknown, thus all the entries in the NN array are set to $[-, \infty]$. The query processing begins at the root level, so that $[ptr(root), \infty]$ is the first and the only request in PR. In Listing 3.2 the k-NN query algorithm is described in detail.

Listing 3.2 (k-NN query algorithm)

```
NodeSearch(Node $N$, kNNQuery $(Q, k)$)
{
  let $O_p$ be the parent routing object of $N$ /* if $N$ is root then $d(O_i, O_p)$=$d(O_p, Q)$=0 */
  if $N$ is an internal node then {
    for each $rout(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q + r_{O_i}$ then { /* application of Lemma 3.2 */
        compute $d(O_i, Q)$
        if $d_{min}(T(O_i)) \leq r_Q$ then {
          insert $[ptr(T(O_i)), d_{min}(T(O_i))]$ to PR
          if $d_{max}(T(O_i)) < r_Q$ then {
            $r_Q$ = NNUpdate($[-, d_{max}(T(O_i))]$)
            remove from PR all requests for which $d_{min}(T(O_i)) > r_Q$
          }
        } /* if $d_{min}$( ...*/
      }
    } /* for each ...*/
  } else { /* $N$ is a leaf */
    for each $grnd(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q$ then { /* application of Lemma 3.2 */
        compute $d(O_i, Q)$
        if $d(O_i, Q) \leq r_Q$ then {
          $r_Q$ = NNUpdate($[oid(O_i), d(O_i, Q)]$)
          remove from PR all requests for which $d_{min}(T(O_i)) > r_Q$
        }
      }
    } /* for each ...*/
  }
}

Node ChooseNode(PRQueue PR)
{
  let $d_{min}(T(O_i^*)) = min\{d_{min}(T(O_i))\}$, considering all the entries in PR
  remove entry $[ptr(T(O_i^*)), d_{min}(T(O_i^*))]$ from PR
  return *$ptr(T(O_i^*))$
}
```

```
QueryResult **kNNQuery**(kNNQuery $(Q, k)$)
{
  PR = $\{[ptr(root), \infty]\}$
  **for** $i = 1$ **to** $k$ **do**
    NN[$i$] = $[-, \infty]$    /* $r_Q$ = NN[$k$].$d_{max} = \infty$ */
  **while** PR is not empty **do** {
    NextNode = **ChooseNode**(PR)
    **NodeSearch**(NextNode, $(Q, k)$)
  }
  **return** NN
}
```

The **NNUpdate** operation performs an ordered insertion into the NN array, receiving back the current value of dynamic query radius r_Q =NN[k].d_{max}.

Theorem 3.1

The k-NN query algorithm, as presented in Listing 3.2, is optimal in I/O costs, because it only accesses those nodes, the metric regions of which overlap the query region $(Q, d(Q,\text{NN}[k].d_{max}))$. In other words, the I/O costs of a k-NN query (Q, k) and costs of the equivalent range query $(Q, d(Q,\text{NN}[k].d_{max}))$ are the same.

Proof: Suppose the algorithm accesses a node N (having parent routing entry $rout(O_i)$), the metric region of which does not overlap the range query hypersphere $(Q, d(Q,\text{NN}[k].d_{max}))$, i.e. $d_{min}(T(O_i)) > r_Q$. Let N_0 be a leaf containing the k-th nearest neighbour of Q, N_1 be the parent node of N_0, …, and N_h be the root of M-tree. Let N_0's parent routing entry be $rout(O_{i_0})$, N_1's be $rout(O_{i_1})$, and so on. From the definition of d_{min} and of nesting condition, it follows that

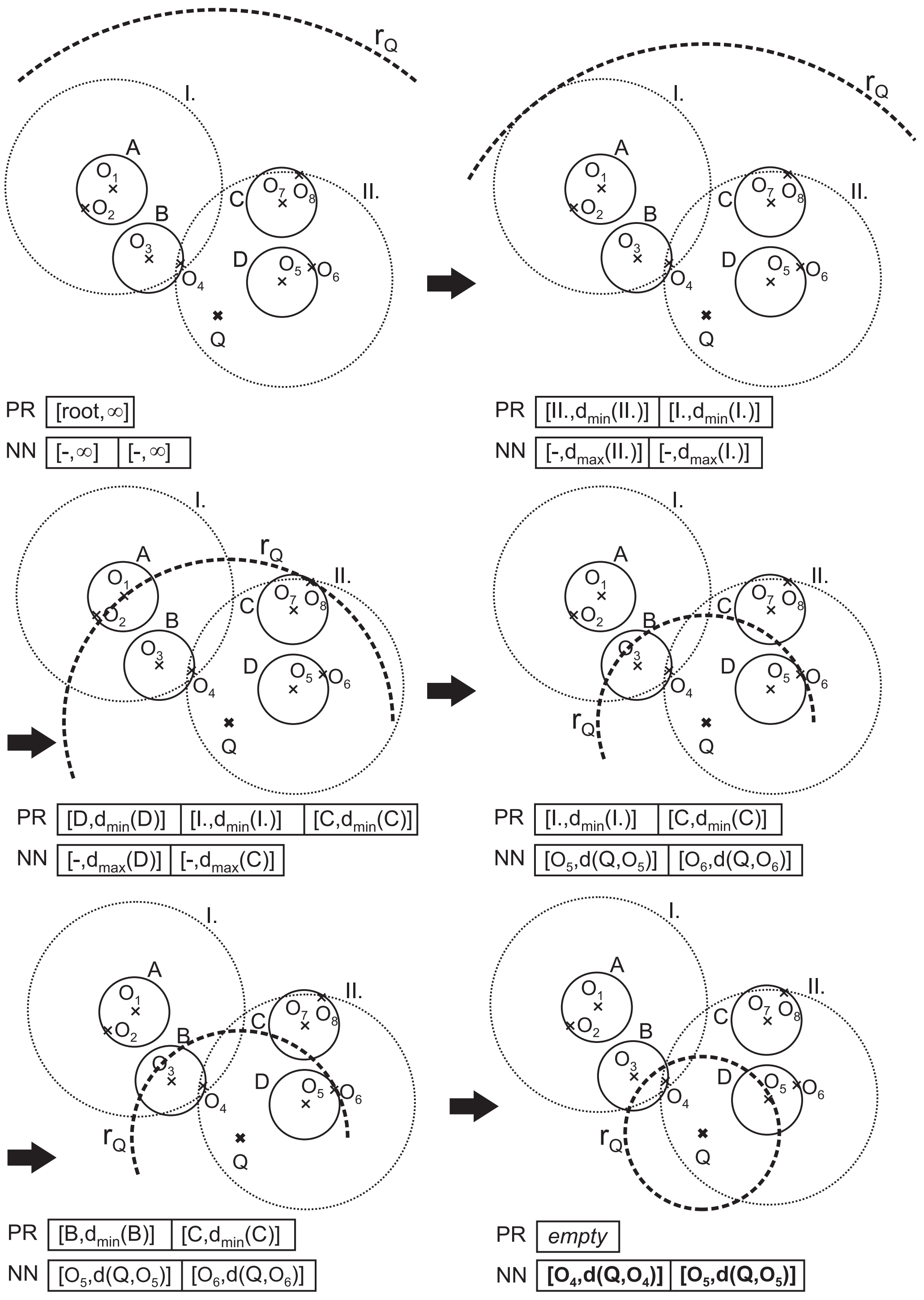
$$r_Q \geq d_{min}(T(O_{i_0})) \geq \ldots \geq d_{min}(T(O_{i_{h-1}}))$$

and thus (by the assumption)

$$d_{min}(T(O_i)) > r_Q \geq d_{min}(T(O_{i_0})) \geq \ldots \geq d_{min}(T(O_{i_{h-1}}))$$

During query processing, the root N_h is replaced in the PR queue by its children N_{h-1}, and so on, until the leaf N_0 is loaded. Since $d_{min}(T(O_i)) > d_{min}(T(O_{i_0}))$, N cannot be accessed until N_0 has been loaded. If N_0 is loaded, however, all the requests having d_{min} greater than r_Q are removed from the PR queue. Hence, N is not accessed and this contradicts the assumption. ■

Note: Although the k-NN query algorithm is optimal in I/O costs, the computation costs are not guaranteed to be optimal. Nevertheless, the computation costs are fairly correlated to the I/O costs, so that computation costs of k-NN processing are only a little higher than computation costs of the equivalent range query.



r_Q
I.
II.
A
B
C
D
O1 O2 O3 O4 O5 O6 O7 O8
Q
PR [root,∞]
NN [-,∞] [-,∞]
PR [II.,d_min(II.)] [I.,d_min(I.)]
NN [-,d_max(II.)] [-,d_max(I.)]
PR [D,d_min(D)] [I.,d_min(I.)] [C,d_min(C)]
NN [-,d_max(D)] [-,d_max(C)]
PR [I.,d_min(I.)] [C,d_min(C)]
NN [O5,d(Q,O5)] [O6,d(Q,O6)]
PR [B,d_min(B)] [C,d_min(C)]
NN [O5,d(Q,O5)] [O6,d(Q,O6)]
PR empty
NN [O4,d(Q,O4)] [O5,d(Q,O5)]

Figure 3.5 Example of a 2-NN query processing

Example 3.4
In Figure 3.5 see an example of a 2-NN query processing. Each of the depicted phases shows the content of PR queue and NN array right before getting and processing a request from PR. The algorithm consecutively eliminates the irrelevant subtrees using the PR and NN structures. Due to the decreasing query radius r_Q, the dynamic query region (Q, r_Q) is reduced down to $(Q, d(Q, O_5))$. Note that the algorithm accesses 4 nodes (because processing each request from PR involves a single node access), while the equivalent range query would also take 4 node accesses.

3.3 M-tree Construction

The first method of M-tree construction is the dynamic insertion of each single object of the dataset into an existing M-tree. The second method, called *bulk loading*, is applicable in case that the entire dataset $\mathbb{S}$ is available before indexing.

3.3.1 Dynamic Object Insertion

First, we will discuss the dynamic insertion of a single object. The insertion of an object into the M-tree follows two general steps (see also Listing 3.3):

1. Find the optimal target leaf into which the object O_i will be inserted as a ground entry. Insert the object O_i into that leaf. Such a target leaf should be chosen, the parent routing object of which is close to O_i. Simultaneously, after insertion of object O_i the enlargements of (grand)parent regions' radii should be minimal (possibly zero).

2. If a node overflows, split the node (partition its content between two new nodes), select two new routing objects and promote them into the parent node. If now the parent node overflows, repeat step 2 for the parent node. If a root is split, the M-tree grows by one level.

Listing 3.3 (dynamic object insertion)

```
Insert(Object $O_i$)
{
  let $N$ be the root node
  TargetLeaf = FindLeaf($N$,$O_i$)
  store ground entry $grnd(O_i)$ in the TargetLeaf
  if TargetLeaf is overfull then
    Split(TargetLeaf)
}
```

3.3.2 Choosing the Target Leaf

An optimal choice of the target leaf is tightly related to the heuristic criterion that suggests to keep the volume[3] of metric regions as small as possible. The larger overall volume of metric regions, the higher probability of an overlap with the query region (which leads to less efficient searching), and vice versa. We further discuss the quality of M-tree hierarchy in the following chapter.

Considering the original approach [37], the motivation used to find the optimal target leaf is to follow a single path in the M-tree (we denote it as *single-way leaf choice*), which would avoid any enlargements of covering radii. At each level of M-tree, the covering subtree of a $rout(O_j)$ is chosen, for which $d(O_j, O_i) \leq r_{O_j}$. If multiple paths with this property exist, the one for which object O_i is closest to the routing object $rout(O_j)$ is chosen. If no routing object exists for which $d(O_j, O_i) \leq r_{O_j}$, an enlargement of a covering radius is necessary. In that case, a node is chosen such that enlargement of its parent region's covering radius is minimal.

The single-way leaf choice will access only h nodes (where h is the height of M-tree), one node at each M-tree level, thus its time complexity is $O(log\ n)$. In Listing 3.4 see the single-way leaf choice algorithm which can be used as a particular implementation of the **FindLeaf** operation, as presented in Listing 3.3. We denote the whole procedure of inserting an object using the single-way leaf choice as *single-way insertion*.

Listing 3.4 (single-way leaf choice)

```
Node FindLeafSingleWay(Node $N$, Object $O_i$)
{
  if $N$ is a leaf node then
    return $N$
  let $\mathcal{N}_{in}$ be the set of entries $rout(O_j)$ from $N$ for which $d(O_j, O_i) \leq r_{O_j}$
  if $\mathcal{N}_{in}$ is not empty then
    let $rout(O_j^*)$ be an entry in $\mathcal{N}_{in}$ such that $d(O_j^*, O_i)$ is minimum
  else {
    let $rout(O_j^*)$ be an entry in $N$ such that $d(O_j^*, O_i) - r_{O_j^*}$ is minimum
    let $r_{O_j^*} = d(O_i^*, O_i)$
  }
  FindLeafSingleWay(*$ptr(T(O_j^*))$)
}
```

[3]We consider only an imaginary volume, since there exists no universal notion of volume in general metric spaces. However, without loss of generality, we can say that volume of a hyper-spherical metric region grows if its covering radius increases.

Example 3.5

In Figure 3.6 see an example of the single-way leaf choice. At the root level, the child node of entry $rout_0(O_1)$ is chosen, having the routing object closest to the inserted object O_i. At the second level, there exist no routing entry the metric region of which bounds O_i and, therefore, the child node of entry $rout_1(O_1)$ is chosen, because the enlargement of its covering radius is smaller than for $rout_1(O_3)$. The child leaf of entry $rout_1(O_1)$ is returned as the target leaf.

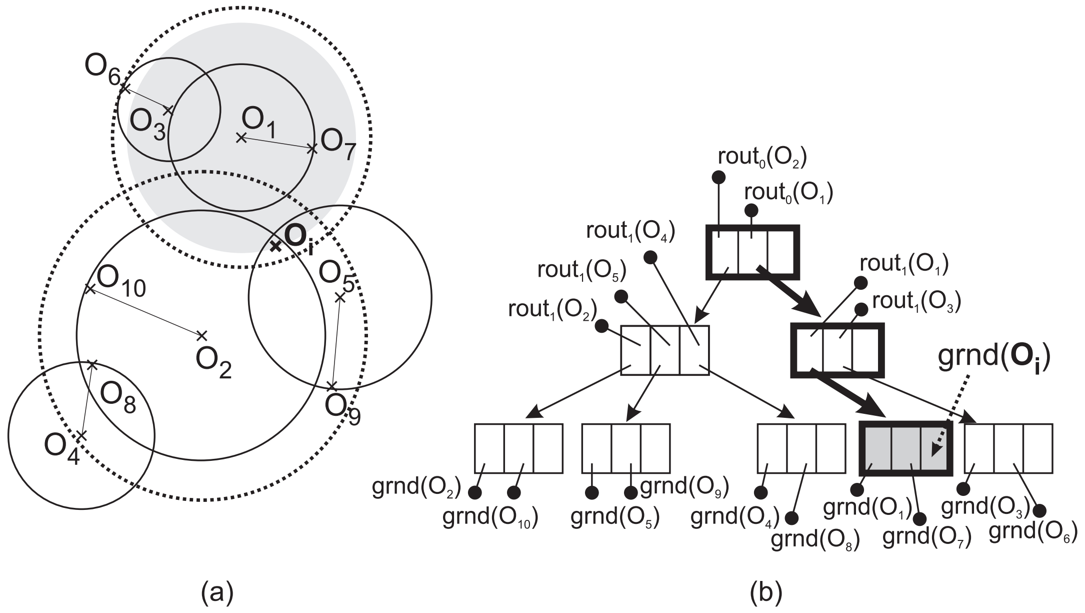


Figure 3.6 Single-way leaf choice:
(a) The target enlarged region (grayed) represented by $rout_1(O_1)$
(b) Single path leading to the chosen leaf

3.3.3 Node Splitting

After a node overflows, it must be split. In order to keep the overlap between regions of the two new nodes minimal , a suitable *splitting policy* must be chosen, determining how to split a given node. In particular, the splitting algorithm must decide which objects to select as the new routing objects (a *promotion* policy), and how to partition the objects between the new nodes (a *partition* policy). In Listing 3.5 see the algorithm of node splitting.

As the original experiments [80] have shown, the `minMAX_RAD` promotion policy (in the split algorithm denoted as **Promote**) causes the best search efficiency of the M-tree. The `minMAX_RAD` method examines all of the $\frac{m(m-1)}{2}$ (where m is the number of entries in node) pairs of objects candidating to the two new routing objects. For every such pair, the remaining entries in the node are temporarily partitioned between two groups, belonging to the objects of the pair. For each pair of candidate routing objects a maximal covering radius is determined. Finally, such a pair $(rout(O_i), rout(O_j))$ of new routing objects is chosen (or promoted), for which the maximal radius (the greater of the two radii r_{O_i}, r_{O_j}) is minimal.

For the object partition policy (in the split algorithm denoted as **Partition**), a distribution according to *general hyperplane* is used as the most beneficial method. An object is simply assigned to the closer routing object. In order to preserve the minimal node utilization threshold, a fixed number of objects is distributed according to the *balanced distribution*.

A comprehensive description of various M-tree construction details as well as the discussion about the splitting policies is included in [80].

Listing 3.5 (node splitting)

```
Split(Node $N$)
{
  let $\mathcal{N}$ be the set of entries in node $N$
  if $N$ is not the root then
    let $rout(O_p)$ be the parent entry of $N$, stored in a node $N_p$
  allocate a new node $N'$
  Promote($\mathcal{N}$,$O_{r_1}$,$O_{r_2}$)
  Partition($\mathcal{N}$,$O_{r_1}$,$O_{r_2}$,$\mathcal{N}_{r_1}$,$\mathcal{N}_{r_2}$)
  store $\mathcal{N}_{r_1}$'s entries in $N$ and $\mathcal{N}_{r_2}$'s entries in $N'$
  if $N$ is the current root then {
    allocate a new root node $N_p$
    store $rout(O_{r_1})$ and $rout(O_{r_2})$ in $N_p$
  } else {
    replace entry $rout(O_p)$ with entry $rout(O_{r_1})$ in $N_p$
    store $rout(O_{r_2})$ in $N_p$
    if $N_p$ is overfull then
      Split($N_p$)
  }
}
```

3.3.4 Bulk Loading

The bulk loading algorithm has been proposed in [34], allowing to construct M-tree for the entire dataset $\mathbb{S}$ in a batch. On a given dataset, a hierarchy is recursively built, resulting into a complete M-tree. Basically, the bulk loading algorithm follows the GNAT partitioning idea, and can be described as follows:

1. First, given a dataset of objects $\mathbb{S}$, we perform a k-way clustering by sampling k objects $O_1, \ldots, O_k$ from $\mathbb{S}$.

2. Then, for each object in $\mathbb{S}$ we determine its nearest sample O_i, and insert the object into a set $\mathcal{F}_i$. In this way we obtain k clusters of objects.
3. Now, we invoke the bulk loading algorithm recursively on each of the k clusters $\mathcal{F}_1, \ldots, \mathcal{F}_k$, obtaining k sub-trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$.
4. Finally, for the k sub-trees a root is created, resulting into a single k-ary tree.

The bulk loading algorithm, as presented, would produce a non-balanced tree (GNAT actually). In order to obtain an M-tree, two different techniques are used:

- The objects in an underfull set $\mathcal{F}_i$ are reinserted into the other sets, and the corresponding sample object is deleted.
- The taller sub-trees are split, obtaining shorter sub-trees. The roots of the new sub-trees are inserted into the sample set, replacing the original sample objects.

For a more precise description of the bulk loading algorithm we refer to [34] or [80].

3.4 M-tree Modifications

Recently, several modifications of M-tree have appeared. The Slim-trees [103] extend the M-tree by a new splitting technique based on *minimum spanning tree* (MST), thus decreasing the distance computation construction costs. Moreover, the slim-down algorithm is introduced, a post-processing method decreasing the search costs.

Besides the original distance function d (used for indexing), an extension of M-tree, called *Query-Index-Comparison* (QIC) [36], allows to exploit a *query distance function* d_Q and a *comparison distance function* d_C. The query distance functions provide a more flexible querying in M-tree, because the user can formulate a query exploiting any metric d_Q for which d is lower-bounding. Moreover, any cheap comparison distance function d_C which lower-bounds d can be used to quickly discard irrelevant subtrees of the M-tree during search. The d_Q as well as d_C distance functions even need not to be full metrics. The authors point out that the Query-Index-Comparison approach can be extended to other MAMs, namely the VP-tree, GNAT, and LAESA.

Very recently, the M$^+$-tree was introduced [114], a modification of M-tree designed to index datasets embedded within Euclidean vector spaces. The M$^+$-tree exploits a *key dimension* concept, according to which a single hyper-spherical region is divided (in a generalized hyper-plane way) in two parts, each belonging to one half of so-called *twin-node*. The authors of M$^+$-tree report a slightly better search efficiency in average, when compared to the original M-tree.

Chapter 4

Quality of M-tree Hierarchy

Since the M-tree's nesting condition is very weak, the efficiency of search in a given dataset is significantly influenced by particular M-tree hierarchy, even though the correctness and the logic of search are guaranteed for all M-tree hierarchies satisfying the nesting condition. The key problem of M-tree search efficiency resides in:

1. the overall volume of M-tree regions defined by routing entries. The larger volume, the higher probability of an overlap with query region and, consequently, the higher search costs.
2. a quantity of overlaps among metric regions. If we realize, the query processing must access all nodes, the parent metric regions of which overlap the query region. If the query region lies (even partially) in an overlap of two or more regions, all the appropriate nodes must be accessed, thus the search costs grow.

Originally, the algorithms on M-tree have been developed to achieve a trade-off, an efficient construction and a (relatively) efficient searching. Consequently, the M-tree construction techniques incorporate decision moments, that regard only a partial knowledge about the distance distribution in a given dataset. Using single-way dynamic insertion, the M-tree hierarchy is constructed locally – at a moment when the nodes are about to split. On the other side, the bulk loading algorithm works with the entire dataset, however, it also works locally – according to several sample objects. The local construction methods cause the M-tree hierarchies are not compact enough, which increases the overall volume of metric regions as well as the quantity of overlaps among them.

In our approach, we wanted to utilize also global techniques of (re)building M-tree, providing a reasonable optimization of the M-tree hierarchy. In order to improve the search efficiency at the expense of construction costs, in this chapter we propose two global methods of constructing more compact M-tree hierarchies [98] – the *generalized slim-down algorithm* and the *multi-way object* *insertion*. The motivation for such efforts has been well-founded by a common DBMS scenario, in which the database (dataset $\mathbb{S}$) is updated only occasionally (the dynamic insertions/deletions are not frequent) but, on the other hand, there are many queries issued at a moment. In such scenario, we rather favour to speedup the search process, while the costs of index updating are not so important. Following this idea, the two proposed methods decrease both the overlaps among metric regions as well as the overall volume, which leads to a higher search efficiency.

4.1 Slim-Down Algorithm

Recently, a post-construction method has been proposed for the Slim-tree [103] (which is, in fact, the same structure as M-tree), called the *slim-down algorithm*. The slim-down algorithm has been used for improvement of Slim-tree hierarchy already built by dynamic object insertions.

The basic idea of the slim-down algorithm is an assumption, that a more suitable leaf exists for a ground entry stored in a leaf. Given a ground entry $grnd(O_i)$ in a leaf N (having parent routing entry $rout(O_p)$) where $d(O_i, O_p) = r_{O_p}$, the task is to find a non-full leaf N^* (having parent routing entry $rout(O_p^*)$) such that $d(O_i, O_p^*) < r_{O_p^*}$. If such a leaf exists, the entry $grnd(O_i)$ is inserted into them (without a need of $r_{O_p^*}$ enlargement), and deleted from N together with a decrease of r_{O_p} (if O_i was the most distant object from O_p in N). The algorithm is repeated for all ground entries in all leaves as long as the movements of entries occur. The experiments [103] have shown, that the original version of slim-down algorithm improves the search efficiency by 35% (in average).

4.1.1 Generalized Slim-Down Algorithm

We have generalized [98] the slim-down algorithm also for the inner nodes as follows:

1. The algorithm traverses each M-tree level, starting at the leaf level $k = h-1$. For each node N at the k-th level ($h > k > 0$), a better location for each of its entries $entry(O_i)$ is tried to find:

   (a) For a routing entry $rout(O_i)$ in an inner node N, a set of relevant nodes $\mathcal{N}$ is retrieved. This is achieved by the *node sieve* operation (see Listing 4.1), which returns such nodes N_j (at the k-th level of M-tree), the parent metric regions (O_j, r_{O_j}) of which *entirely* contain the region (O_i, r_{O_i}), i.e. $\forall O_j, d(O_j, O_i) + r_{O_i} < r_{O_j}$. In other words, the M-tree is used as a hierarchical sieve having hyper-spherical holes, through which the "region ball" (O_i, r_{O_i}) may fall onto the k-th level.

   (b) For a ground entry $grnd(O_i)$ in a leaf N, a set of relevant leaves $\mathcal{N}$ is retrieved using a leaf sieve, which is a special case of the node sieve where the "region ball" is just a point $(O_i, 0)$. In fact, the same leaves are retrieved as those accessed by a point query $(O_i, 0)$.

2. Among the nodes in $\mathcal{N}$, a non-full node N^* is chosen:
   (a) In case of routing entry $rout(O_i)$, such node N^* is chosen, the parent routing object O_p^* of which is closest to the farthest possible object in $T(O_i)$, i.e. N^* is chosen such that $d(O_i, O_p^*) + r_{O_i}$ is minimal.
   (b) In case of ground entry $grnd(O_i)$, such leaf N^* is chosen, the parent routing object O_p^* of which is closest to O_i, i.e. such that $d(O_i, O_p^*)$ is minimal.

3. The entry $entry(O_i)$ is moved from N to the node N^*. Note that, due to correct node sieve filtering, the movements of entries do not violate the nesting condition.

4. Let $rout(O_p)$ be the parent routing entry of N. If $d(O_i, O_p) + r_{O_i} = r_{O_p}$, the covering radius r_{O_p} of the N's parent routing entry $rout(O_p)$ (as well as radii of all grandparent entries) could be decreased.

Processing a given level could be repeated as long as any movements of entries occur. After a level is finished, the algorithm starts for the $(k-1)$-th level. During the algorithm processing, the number of nodes at each M-tree level is preserved, since only redistribution of entries at the same level is performed. In Listing 4.2 see a more precise description of the generalized slim-down algorithm.

Listing 4.1 (node sieve algorithm)

```
Node[] **NodeSieve**(Node $N$, RQuery ($Q$, $r_Q$), int CurrentLevel, int TargetLevel)
{
  let $O_p$ be the parent routing object of $N$
  **for each** $entry(O_j)$ in $N$ **do** {
    **if** $|d(O_p, Q) - d(O_j, O_p)| \leq r_Q + r_{O_j}$ **then** { /* application of Lemma 3.2 */
      compute $d(O_j, Q)$
      **if** $d(O_j, Q) + r_Q < r_{O_j}$ **then** /* the "sieve" condition */
        **if** CurrentLevel < TargetLevel then
          **NodeSieve**($ptr(T(O_j))$, $(Q, r_Q)$, CurrentLevel+1, TargetLevel)
        **else**
          add $N$ to the query result
    }
  } /* for each ... */
}
```

Listing 4.2 (generalized slim-down algorithm)

```
SlimDown(MTree $T$)
{
  for $k = h - 1$ to $1$ do /* the root level (k=0) cannot be slimmed */
    for each node $N$ at the $k$-th level of $T$ do
      for each $entry(O_i)$ in $N$ do { /* either routing or ground entry */
        if $N$ is leaf then
          let $\mathcal{N}$ = NodeSieve($root(T)$, ($O_i$, 0), 0, $k$)
        else
          let $\mathcal{N}$ = NodeSieve($root(T)$, ($O_i$, $r_{O_i}$), 0, $k$)
        let $rout(O_j^*)$ be parent routing entry of a non-full node $N^*$ in $\mathcal{N}$,
          such that $d(O_j^*, O_i) + r_{O_i}$ is minimal
        move $rout(O_j^*)$ from $N$ to $N^*$ (but only if $N$ does not underflow)
        adequately decrease the covering radii of all the $N$'s (grand)parent routing entries
      }
}
```

Example 4.1

In Figure 4.1 see an example of the generalized slim-down algorithm processing. In Figure 4.1a the original M-tree is represented. The M-tree hierarchy after slimming-down the leaf level is shown in Figure 4.1b. The ground entries 4 and 6 have been moved from leaf B (D respectively) into leaf A, while the appropriate covering radii of the (grand)parent routing entries have been decreased. The M-tree hierarchy after slimming-down the middle level is shown in Figure 4.1c. The routing entries B and D have been moved from node II. (I. respectively) into node I. (II. respectively) and, again, appropriate covering radii of the parent routing entries have been decreased. Note that the M-tree metric region volumes have been significantly decreased, and the overlaps among regions have become smaller.

4.1.2 Construction Costs

The generalized slim-down algorithm is quite expensive, it consumes $O(n)$ node sieve operations (one for each node entry). If we realize, the node sieve operation is similar to point query processing, it takes from $O(n)$ (in the worst case) to $O(log\ n)$ (in the best case) of search costs. Hence, the overall costs of slim-down algorithm are ranged between $O(n\ log\ n)$ and $O(n^2)$. However, for well-structured M-tree hierarchies, the average costs of point query (the node sieve operation respectively) approach to $O(log\ n)$. Moreover, search costs of the node sieve operation can be reduced to e.g. $O(c\ log\ n)$ (where c is a constant) by selection of only sub-optimal nodes used for the object redistribution.

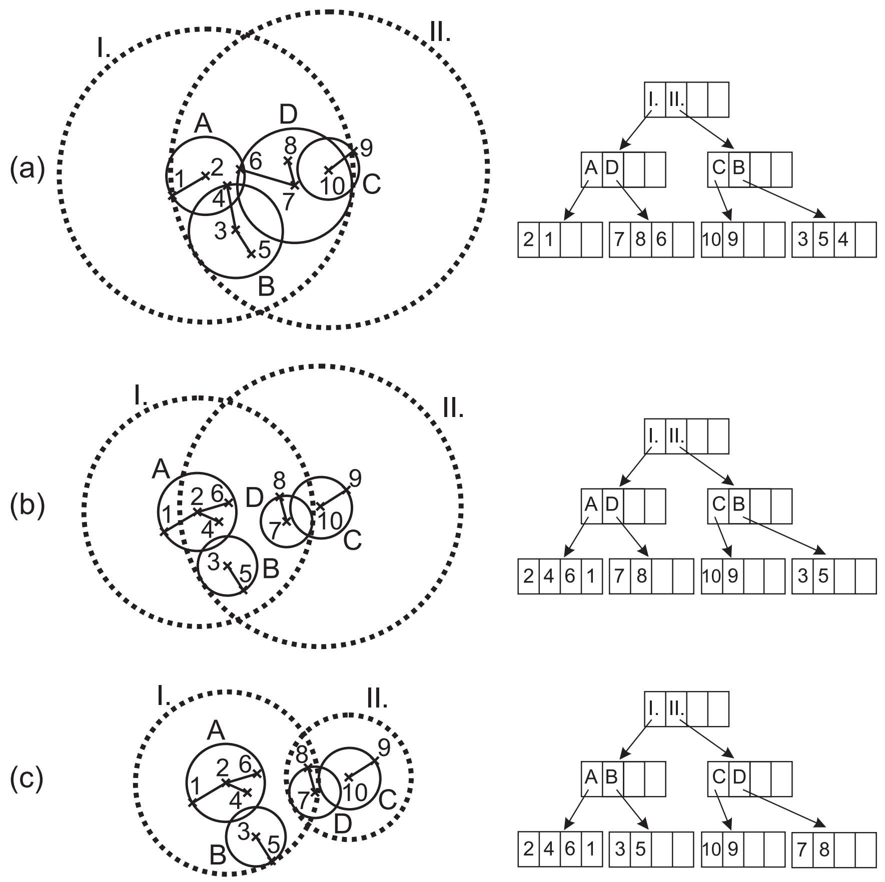


Figure 4.1 (a) The original (not slimmed) M-tree
(b) The M-tree after slimming-down the leaf level
(c) The M-tree after slimming-down the middle level

From the DMBS point of view, the slim-down algorithm is not a transaction, it can be whenever interrupted and resumed. This capability is suitable for running the slim-down algorithm in processor idle mode, thus construction costs can be spread over the time.

4.2 Multi-Way Leaf Choice

The original (single-way) leaf choice heuristic used by dynamic object insertion keeps the construction costs as low as possible and, simultaneously, tries to find a leaf node for which the insertion of object O_i does not enlarge the metric regions much. However, this heuristic behaves very locally (only one path in the M-tree is processed), thus the optimal leaf is rarely chosen.

In our approach, the priority has been focused on choice of the most optimal leaf node at all, by means of *multi-way leaf choice* [98]. In principle, a point query is executed (leaf sieve operation respectively, see Listing 4.1), defined by the inserted object O_i. For all the accessed relevant leaves (their parent routing objects $rout(O_j)$ respectively), the distances $d(O_j, O_i)$ are computed, and the leaf is chosen for which the distance is minimal (see Listing 4.3). If no such a leaf is found, i.e. no region (O_j, r_{O_j}) bounding O_i exists, the single-way leaf choice is performed.

Listing 4.3 (multi-way leaf choice)

```
Node FindLeafMultiWay(Node $N$, Object $O_i$)
{
  let $\mathcal{N}$ = NodeSieve($N$, ($O_i$,0), 0, $h-1$)
  if $\mathcal{N}$ is empty then
    return FindLeafSingleWay($N$, $O_i$)
  let $rout(O_j^*)$ be parent routing entry of a (non-full) node $N^*$ in $\mathcal{N}$, such that $d(O_j^*, O_i)$
    is minimum
  return $N^*$
}
```

The multi-way leaf choice behaves more globally, since multiple paths in the M-tree are examined. In fact, all the leaves the regions of which spatially contain the object O_i are candidates for the optimal leaf. Furthermore, the multi-way leaf choice can optionally consider only the non-full leaves for an object insertion, which leads to a higher node utilization (see experiments in Section 4.4). Considering only the non-full leaves, a leaf is split in case it is overfull, but this can happen only for the single-way leaf choice (i.e. after an unsuccessful attempt of the multi-way leaf choice). The **FindLeafMultiWay** operation is incorporated into the insertion algorithm presented in Listing 3.3, replacing the **FindLeaf** operation. We denote the whole procedure of inserting an object using the multi-way leaf choice as *multi-way insertion*.

Example 4.2
In Figure 4.2 see an example of the multi-way leaf choice (the same situation for the single-way choice has been presented in Example 3.5). The leaf sieve returns two candidate leaves, the metric regions of which spatially contain the inserted object O_i. Between these leaves the one is chosen, the parent routing object of which is closer to O_i. Note that no enlargement of a covering radius is needed.

4.2.1 Construction Costs

Unlike the single-way leaf choice, the multi-way leaf choice is not of logarithmic complexity yet, the point query (leaf sieve operation respectively) search costs

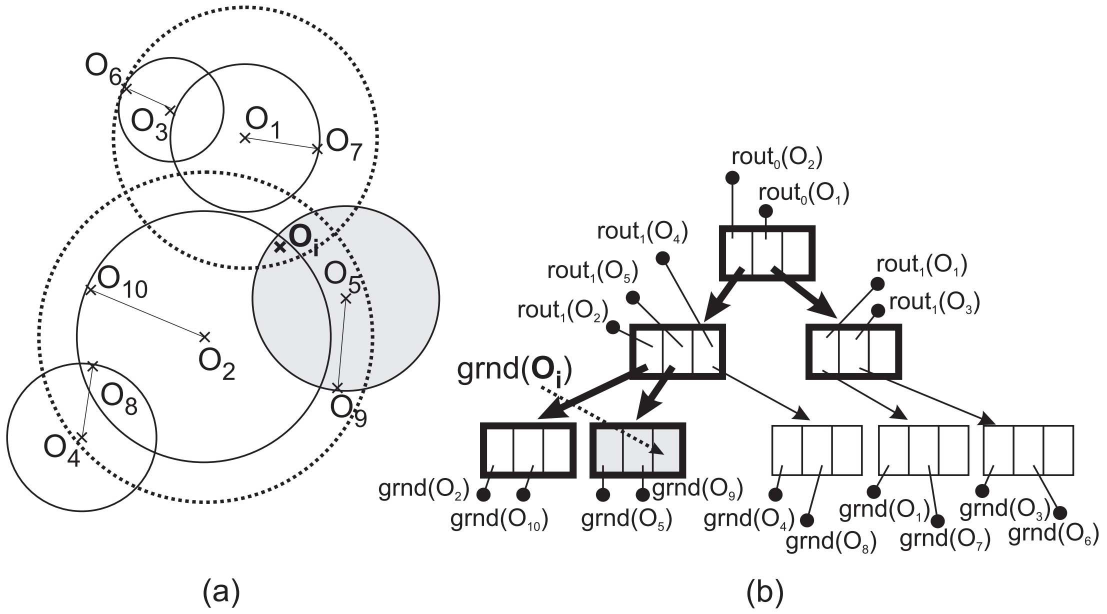


Figure 4.2 Multi-way leaf choice:
(a) The target leaf region (grayed) represented by $rout_1(O_5)$
(b) Multiple paths leading to candidate leaves

take from $O(n)$, in the worst case, to $O(log\ n)$, in the best case. Nevertheless, the costs of leaf sieve operation could be reduced, similarly like for the slim-down algorithm, to e.g. $O(c\ log\ n)$ (where c is a constant) by selection of only sub-optimal leaves used for the object insertion.

4.3 Fat-Factor

In order to evaluate the quality of an M-tree hierarchy built either by the generalized slim-down algorithm or by the multi-way object insertion, we need a measure to quantify the amount of metric region overlaps. In general metric spaces we cannot quantify the volume of two metric regions overlap, and we cannot even compute the volume of a whole metric region. Thus, we cannot measure the quality of an M-tree hierarchy as a sum of overlap volumes.

With the introduction of Slim-tree [103], the *fat-factor* has been introduced as a way how to quantify the number of region overlaps in Slim-tree. We can adopt the fat-factor for usage in M-tree as well. The fat-factor is tightly related to the M-tree search efficiency, because it gives an information about the objects located in region overlaps by means of sequence of point queries.

Definition 4.1 (fat-factor)
For the fat-factor computation, a point query for each ground entry in the M-tree

is performed. Let h be the height of an M-tree T, n be the number of ground entries in T, m be the number of nodes, and I_c be the total I/Os spent by all the n point queries. Then,

$$ff(T) = \frac{I_c - h \cdot n}{n} \cdot \frac{1}{(m - h)}$$

is the *fat-factor* of T, a number in interval $\langle 0, 1 \rangle$. □

For an optimal M-tree hierarchy, the $ff(T)$ is zero. On the other side, for the worst possible M-tree, the $ff(T)$ is equal to one. For an M-tree with $ff(T) = 0$, every performed point query costs h I/Os, while for $ff(T) = 1$ every performed point query costs m I/Os, i.e. the entire M-tree index is traversed.

4.4 Experimental Results

We made several experiments on synthetic vector datasets of multi-dimensional tuples. The datasets were of various dimensionalities, from $D = 2$ to $D = 50$, while the size of a dataset was increasing with the dimensionality, from 20,000 2D tuples to 1 million 50D tuples.

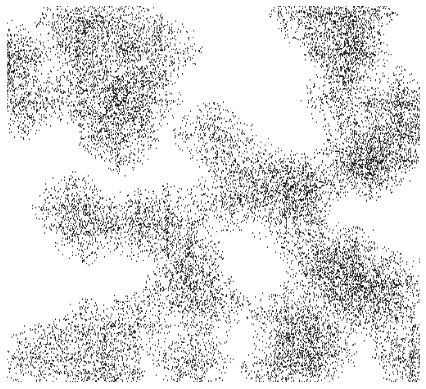


Figure 4.3 Two-dimensional dataset distribution

The tuples were distributed uniformly, inside L_2-spherical clusters with radii increasing from 10% of the domain extent (for 2D tuples) to 80% of the domain extent (for 50D tuples). The number of clusters was increasing with the increasing dimensionality – from 50 to 1,000 clusters. In such distributed datasets, the hyper-spherical clusters became highly overlapping, because of their quantity and large radii. See an example of 2D dataset distribution in Figure 4.3.

4.4.1 Building the M-tree

The datasets were indexed in five ways. The single-way insertion method and the bulk loading algorithm (in the graphs denoted as SingleWay and Bulk Loading) represent the original methods of M-tree construction. In addition, the multi-way insertion method (denoted as MultiWay) and the generalized slim-down algorithm represent the new building techniques introduced in this chapter. The slim-down algorithm, as a post-processing technique, was applied on both SingleWay and MultiWay indices, which resulted into indices SingleWay+SlimDown and MultiWay+SlimDown. Some general M-tree statistics are presented in Table 4.2.

Metric:	L_2 (Euclidean)	Node capacity:	20 objs.	Dimensions:	2 – 50
Objects(n):	20,000 – 1,000,000	Tree height(h):	3 – 5	Index size:	1 – 400 MB
		Minimal node utilization:	30% (i.e. 6 objects)		

Table 4.2 M-tree statistics

The first experiment evaluated the M-tree construction costs. The I/O construction costs are presented in Figure 4.4a. We can see that the SingleWay and Bulk Loading indices were built much cheaply than the others, but the construction costs have not been the primary objective in our approach. Figure 4.4b illustrates the average realtime costs per one inserted object.

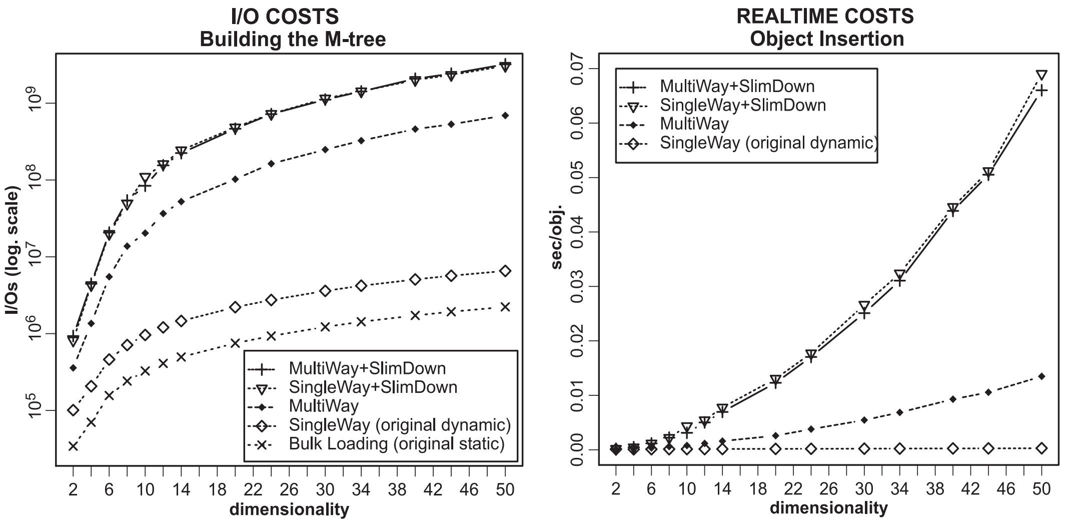


Figure 4.4 Building the M-tree: (a) I/O costs
(b) Realtimes per one object

In Figure 4.5a the fat-factor characteristics of the indices are depicted. The fat-factor of SingleWay+SlimDown and MultiWay+SlimDown indices was very low, which indicates that these indices contained (relatively) few overlapping regions.

An interesting fact can be observed in Figure 4.5b, showing the average node utilization. Thanks to the multi-way selection favouring the non-full leaves, the MultiWay index utilization was by more than 10% higher than utilization of the SingleWay index. This value is not relevant for the SingleWay+SlimDown and MultiWay+SlimDown indices, since the "slimming-down" did not change the average node utilization, i.e. the results were the same as those achieved for SingleWay and MultiWay indices.

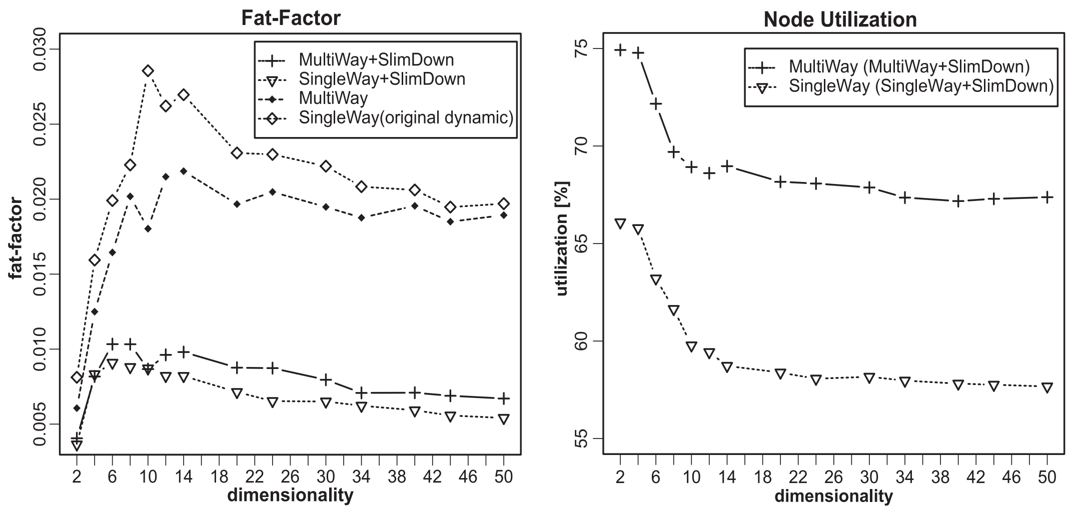


Figure 4.5 Building the M-tree: (a) Fat-factor (b) Node utilization

4.4.2 Range Queries

The objective of our approach was to increase the search efficiency of M-tree. For the querying experiments, many query objects were randomly selected from the datasets. Each query test consisted from 100 to 750 queries (according to the dimensionality and dataset size). The results are averaged.

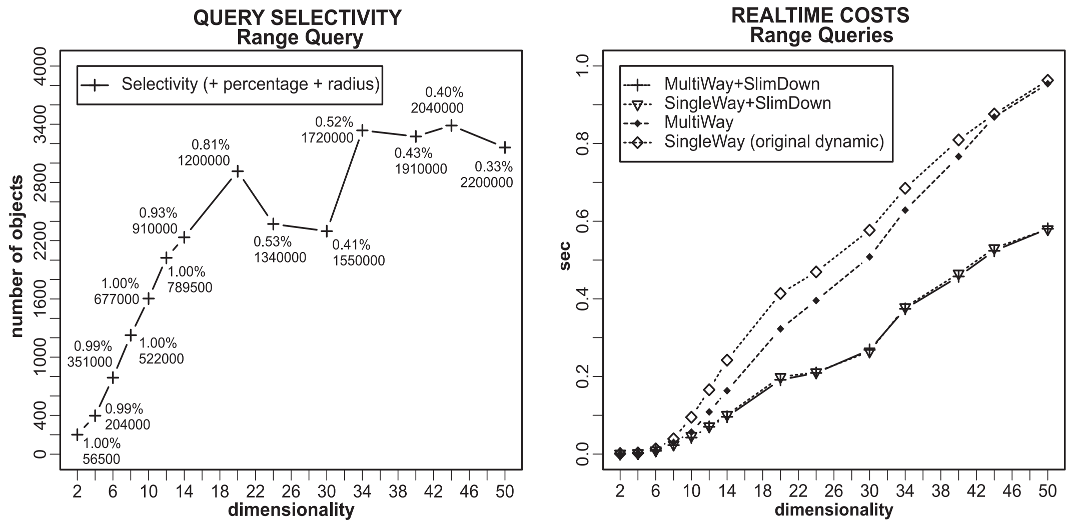


Figure 4.6 Range queries: (a) Query selectivity (b) Realtimes

In Figure 4.6a the average range query selectivity (i.e. the average number of objects in query result) is presented for each dataset. For every range query, the selectivity was set below 1% of all the objects in the respective dataset. The real-time costs for range queries are presented in Figure 4.6b. We can see that query evaluation using SingleWay+SlimDown and MultiWay+SlimDown indices was almost twice as efficient, when compared to the SingleWay index.

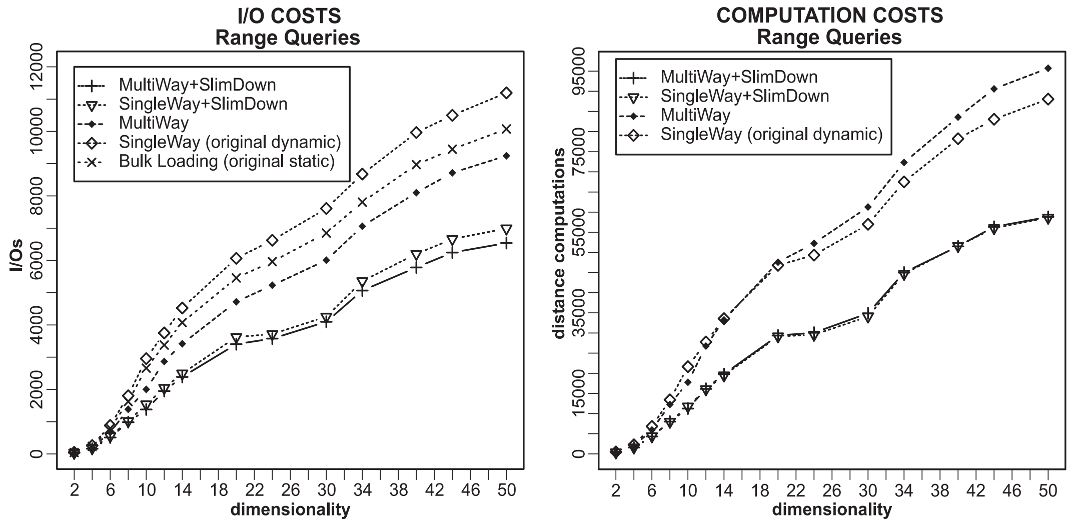


Figure 4.7 Range queries: (a) I/O costs (b) Computation costs

The I/O costs and the computation costs for range queries are presented in Figure 4.7. The computation costs represent the total number of the distance computations needed for a query evaluation.

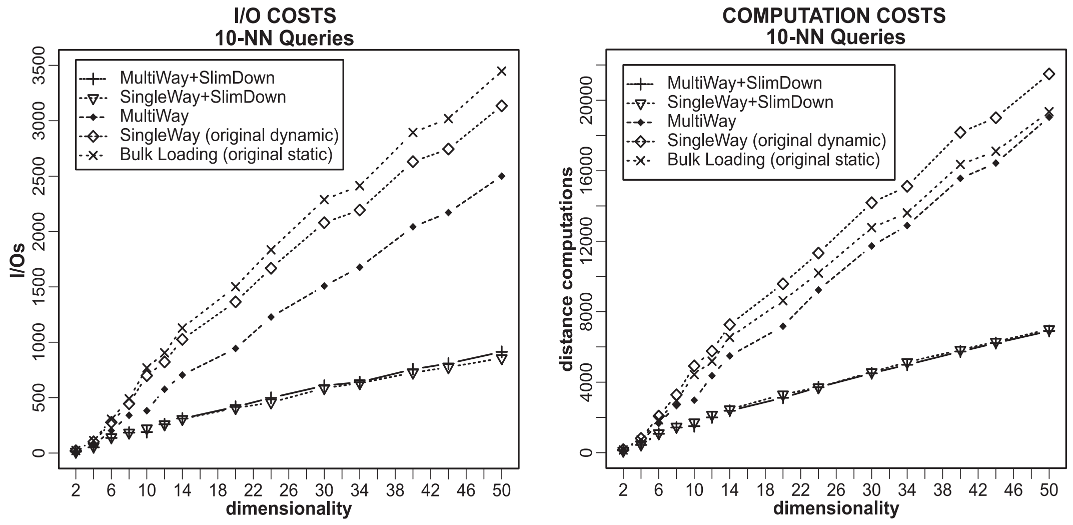
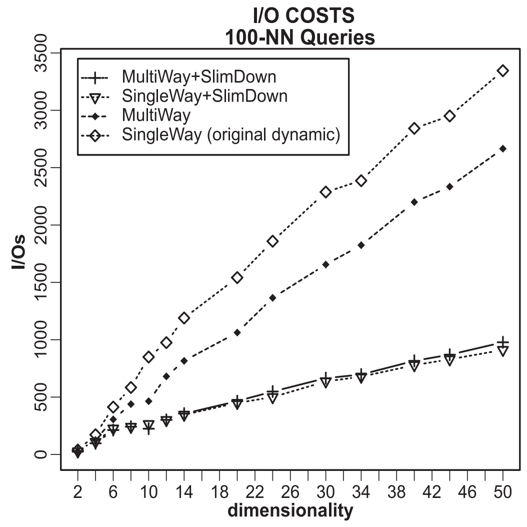


Figure 4.8 10-NN queries: (a) I/O costs (b) Computation costs

4.4.3 Nearest Neighbours Queries

The search efficiency growth is even more obvious for k-NN queries processing. In Figure 4.8a the I/O costs are presented for 10-NN queries. As the results show, querying the SingleWay+SlimDown index consumed 3.5-times less I/Os than querying the SingleWay index. A similar behaviour can be observed also for the computation costs, presented in Figure 4.8b.



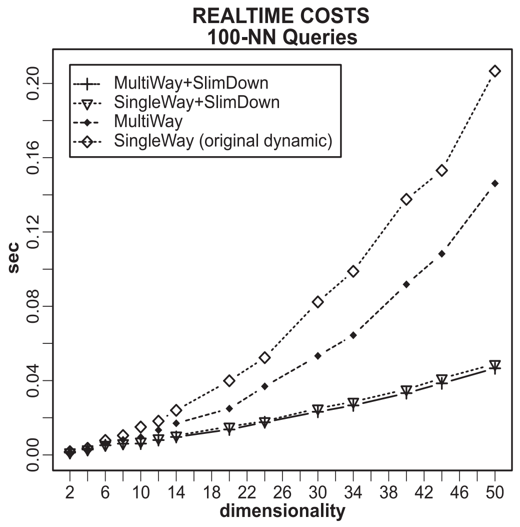


Figure 4.9 100-NN queries: (a) I/O costs (b) Realtimes

The most promising results are presented in Figure 4.9, for the 100-NN queries testing. The search efficiency of the SingleWay+SlimDown index is here by more than 300% higher than search efficiency of the SingleWay index.

An additional experimental evaluation of the generalized slim-down algorithm on real-world datasets is included in Chapter 6 (in the context of PM-tree) and Chapter 8 (in the context of semi-metric search in Text Retrieval).

Chapter 5

PM-tree

As we have discussed previously, the efficiency of search in M-tree is dependent on the overall volume of metric regions. The higher volume, the lower search efficiency. In the previous chapter, we have presented two ways of reducing the overall volume using *object redistribution* but, however, the redistribution alone is not an ultimate solution and, moreover, it is computationally expensive.

In order to achieve even higher volume reduction and to keep the construction costs low, we consider also another region reduction in this chapter, a modification of metric region *shape*.

5.1 Motivation

Each metric region of M-tree is described by a bounding hyper-sphere (defined by a local pivot and a covering radius). However, the shape of hyper-spherical region is far from optimal, because it does not bound the data objects tightly together, thus the region volume is too large. In other words, relatively to the hyper-sphere volume, there is only ”few” objects spread inside the hyper-sphere, and a huge proportion of empty space[1] is covered. Consequently, for hyper-spherical regions of large volumes the probability of overlap with the query region grows, thus query processing becomes less efficient.

On the other side, the tightest possible boundary for a set of objects (i.e. boundary for which the proportion of dead space is zero) is the set of objects itself. Unfortunately, the simple description of such a ”grain region” is useless, since storage of all the objects is too large, and an overlap check with a query region would take many distance computations. In fact, checking a ”grain region” for an overlap is equivalent to sequential search over all the objects stored in the appropriate covering subtree.

Keeping the previous observations in mind (properties of the hyper-sphere region and the ”grain region”), we can formulate four requirements on a compact

[1]The uselessly indexed empty space is often referred as the ”dead space” [18].

metric region shape (a trade-off between region volume and storage/computation costs), bounding a given set of objects:

- The representation of a region stored in a routing entry should be as small as possible, so that storage of inner nodes will be (by far) smaller than storage of the leaves.
- The shape of region should be easy to check for an overlap with the query region (query hyper-sphere respectively).
- The shape should be *compact*, it should bound the objects tightly together, so that probability of an empty intersection with the query region (i.e. a case that no indexed objects are located in the intersection) will be minimal.
- Given a set of regions, there should be easy to create a super-region the shape of which bounds each of the regions. This requirement is tree-specific – it ensures that creating a super-region (when splitting an inner node) can be automatically handled. Moreover, the requirement guarantees the nesting condition (introduced for M-tree) is still preserved.

5.1.1 Hyper-Ring Region

In the following, we will introduce the concept of *hyper-ring* metric region, a combination of two nested hyper-spheres, the first one acting as usual while the second one acting as a "hole" inside the first hyper-sphere.

Definition 5.1 (complementary-spherical metric region)
We denote $\neg(O_i, r_{O_i}) \subset \mathbb{U}$ a *complementary-spherical metric region*, such that an object $O_j \in \mathbb{U}$ is covered by the region, i.e. $O_j \in \neg(O_i, r_{O_i})$, just in case that $d(O_j, O_i) \geq r_{O_i}$. □

Lemma 5.1
Let $\neg(O_i, r_{O_i})$ be a complementary-spherical metric region, and (Q, r_Q) be a hyper-spherical query region. If $d(O_i, Q) + r_Q < r_{O_i}$, then for each object $O_j \in \neg(O_i, r_{O_i})$, it is $d(O_j, Q) > r_Q$. Thus, the regions do not overlap.
Proof: Since $d(O_j, O_i) \geq r_{O_i}$ holds for $\forall O_j \in \neg(O_i, r_{O_i})$, it is

$$\begin{array}{rcll} d(O_i, Q) + r_Q & < & d(O_j, O_i) & \text{(by hypothesis and Definition 5.1)} \\ r_Q & < & d(O_j, O_i) - d(O_i, Q) & \\ d(O_j, Q) & \geq & d(O_j, O_i) - d(O_i, Q) & \text{(by triangular inequality)} \\ d(O_j, Q) & > & r_Q & \blacksquare \end{array}$$

Example 5.1
See the idea of Lemma 5.1 in Figure 5.1. While the L_1-complementary-spherical region $\neg(O_i, r_{O_i})$ overlaps the query region, the L_2-complementary-spherical region $\neg(O_i, r_{O_i})$ does not overlap the query region.

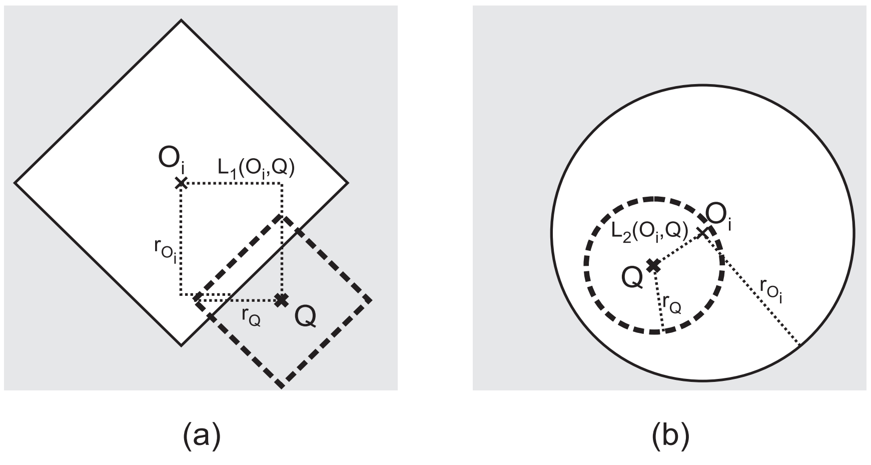


Figure 5.1 (a) Overlapped L_1-complementary-spherical metric region
(b) Non-overlapped L_2-complementary-spherical metric region

Definition 5.2 (hyper-ring metric region)
We call $(O_i, r^{low}_{O_i}, r^{up}_{O_i}) \subset \mathbb{U}$ (where $0 \leq r^{low}_{O_i} \leq r^{up}_{O_i}$) *hyper-ring metric region*, such that an object $O_j \in \mathbb{U}$ is covered by the region, i.e. $O_j \in (O_i, r^{low}_{O_i}, r^{up}_{O_i})$, just in case that $r^{low}_{O_i} \leq d(O_j, O_i) \leq r^{up}_{O_i}$.We call $r^{low}_{O_i}$ a *hyper-ring lower-bound distance* and $r^{up}_{O_i}$ a *hyper-ring upper-bound distance.* □

Lemma 5.2
Let $(O_i, r^{low}_{O_i}, r^{up}_{O_i})$ be a hyper-ring metric region, and (Q, r_Q) be a hyper-spherical query region. If $d(O_i, Q) + r_Q < r^{low}_{O_i} \vee d(O_i, Q) > r_Q + r^{up}_{O_i}$, then for each object $O_j \in (O_i, r^{low}_{O_i}, r^{up}_{O_i})$, it is $d(O_j, Q) > r_Q$. Thus, the regions do not overlap.

Proof: Follows immediately from Definition 5.2 and Lemmas 3.1 and 5.1. ■

Example 5.2
See the idea of Lemma 5.2 in Figure 5.2. While the L_1-hyper-ring region $(O_i, r^{low}_{O_i}, r^{up}_{O_i})$ does not overlap the query region, the L_2-hyper-ring region $(O_i, r^{low}_{O_i}, r^{up}_{O_i})$ overlaps the query region.

5.2 Structure of PM-tree

Recently [93, 100], we have proposed the *Pivoting M-tree* (PM-tree), an extension of M-tree exploiting a combination of hyper-ring regions for M-tree region volume reduction. The idea of PM-tree has been motivated by the previously mentioned observations on metric region representation, that is, the PM-tree structure has been designed in order to satisfy the four requirements on metric region shape.

Since PM-tree is an extension of M-tree, we just describe the new facts instead of a comprehensive definition. First of all, a set of p global pivots $P_t \in \mathbb{S}$ must be

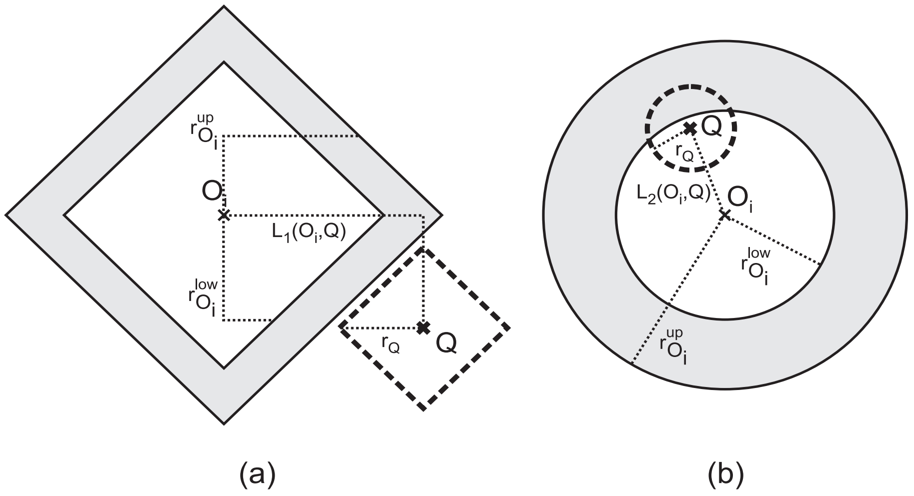


Figure 5.2 (a) Non-overlapped L_1-hyper-ring metric region
(b) Overlapped L_2-hyper-ring metric region

selected, similarly like by LAESA method. The set of pivots is stored externally in a *pivot file* – the size of pivot file is fixed for all the lifetime of a particular PM-tree index. A *routing entry*, in a PM-tree inner node, is defined as:

$$rout_{PM}(O_i) = [O_i, ptr(T(O_i)), r_{O_i}, d(O_i, \mathrm{Par}(O_i)), \mathrm{HR}]$$

The new HR attribute stands for an array of p_{hr} ($p_{hr} \leq p$) intervals, where the t-th interval $\mathrm{HR}[t] = \langle \mathrm{HR}[t].\mathrm{min}, \mathrm{HR}[t].\mathrm{max}\rangle$ is the smallest interval covering distances between pivot P_t and each of the objects stored in leaves of $T(O_i)$, i.e. $\forall O_j \in T(O_i)$:

$$\mathrm{HR}[t].\mathrm{min} = min \bigcup_j \{d(P_t, O_j)\}$$

$$\mathrm{HR}[t].\mathrm{max} = max \bigcup_j \{d(P_t, O_j)\}$$

The set of global pivots and the array HR define a set of p_{hr} hyper-ring regions $(P_t, \mathrm{HR}[t].\mathrm{min}, \mathrm{HR}[t].\mathrm{max})$, or simply $(P_t, \mathrm{HR}[t])$, such that *all* objects stored in the covering subtree $T(O_i)$ are located inside *each* of the hyper-rings, i.e. $\forall O_j \in T(O_i), \forall t \leq p_{hr} \Rightarrow O_j \in (P_t, \mathrm{HR}[t])$.

Since each hyper-ring region (P_t, HR[t]) defines a metric region containing *all* the objects stored in $T(O_i)$, an intersection of all the hyper-rings and the hyper-sphere (O_i, r_{O_i}) forms a metric region bounding all the objects in $T(O_i)$ as well. Due to the intersection with hyper-sphere, the PM-tree metric region is always smaller than the original M-tree region defined just by the hyper-sphere. For a comparison of an M-tree region and an equivalent PM-tree region, see Figure 5.3.

Example 5.3
In Figure 5.4 see a hierarchy of PM-tree regions and the appropriate PM-tree. Two global pivots P_1 and P_2 are used. Each PM-tree metric region is represented by an intersection of the hyper-sphere (O_i, r_{O_i}) and two hyper-rings $(P_1, \mathrm{HR}[1])$ and $(P_2, \mathrm{HR}[2])$.

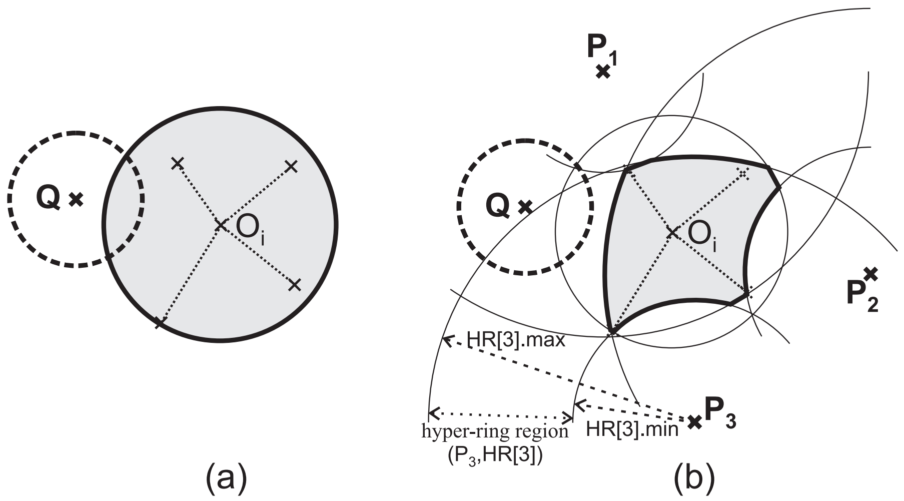


Figure 5.3 (a) Region of M-tree
(b) Region of PM-tree (reduced by three hyper-rings)

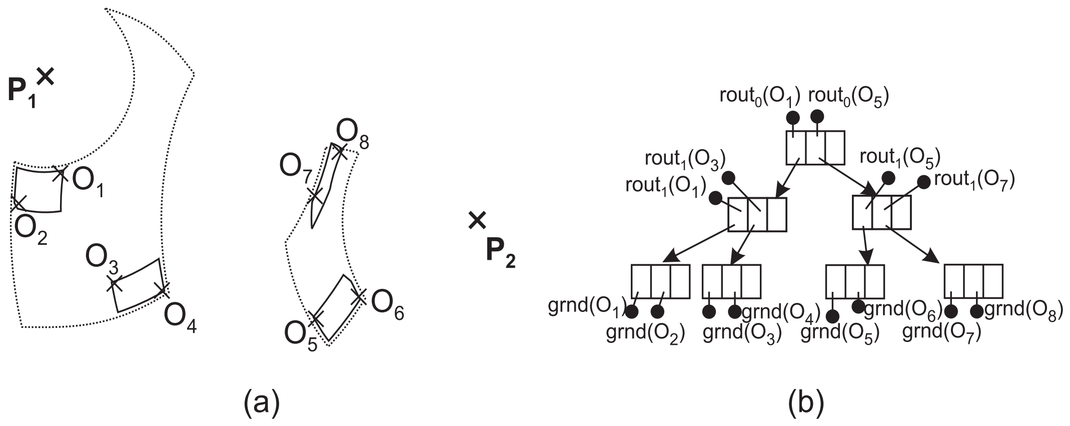


Figure 5.4 (a) Hierarchy of PM-tree regions (b) Appropriate PM-tree

Note: The nesting condition (see Condition 3.1), satisfied by the M-tree, is preserved by the PM-tree as well, because each hyper-ring interval $\mathrm{HR}[t]$ is nested inside a larger (or equal) interval defined by the parent routing entry, thus also each hyper-ring region $(P_t, \mathrm{HR}[t])$ is nested inside its parent hyper-ring.
For a PM-tree leaf, we define a ground entry as:

$$grnd_{PM}(O_i) = [O_i, oid(O_i), d(O_i, \mathrm{Par}(O_i)), \mathrm{PD}]$$

The new PD attribute stands for an array of p_{pd} *pivot distances* ($p_{pd} \leq p$), where the t-th distance $\text{PD}[t] = d(O_i, P_t)$. Using PD arrays, the leaves of PM-tree can be filtered the same way, as are filtered objects organized by LAESA-like methods.

The numbers p_{hr} and p_{pd} (both fixed for a PM-tree index lifetime) allow us to specify the "amount of pivoting". Obviously, specifying a suitable $p_{hr} > 0$ and $p_{pd} > 0$, the PM-tree can be tuned to achieve an optimal search efficiency.

5.2.1 Object-to-pivot Distance Representation

In order to minimize the storage volume of HR and PD arrays in PM-tree nodes, a short representation of object-to-pivot distances is necessary. We can easily represent each interval $\text{HR}[t]$ by two 4-byte reals, and a pivot distance $\text{PD}[t]$ by a single 4-byte real. However, when (a part of) the dataset $\mathbb{S}$ is known in advance, we can approximate the 4-byte representation by a single byte. For this reason, a distance distribution histogram for each pivot is created, by random sampling objects from the dataset and computing distances from each sample object to the pivot. Then a distance interval $\langle d^{low}, d^{up} \rangle$ is created, so that majority of the histogram distances falls into that interval, see an example in Figure 5.5.

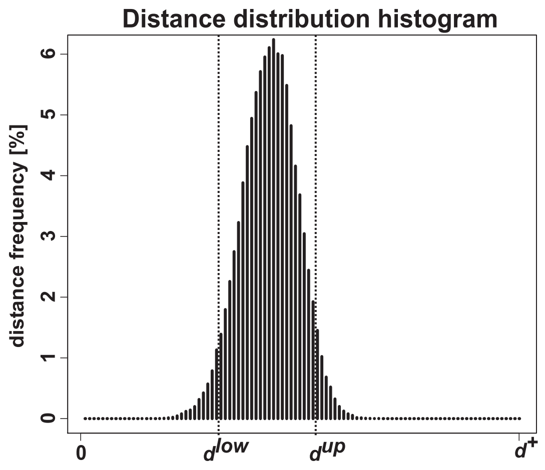


Figure 5.5 Distance distribution histogram for a pivot P_t, 90% of distances $d(P_t, O_i), \forall O_i \in \mathbb{S}$ are in interval $\langle d_t^{low}, d_t^{up} \rangle$

Distance values in HR and PD are scaled into the interval $\langle d_t^{low}, d_t^{up} \rangle$ as 1-byte approximations. Using 1-byte approximations, the storage savings are considerable. As an example, for $p_{hr} = 50$ together with using 4-byte distances, the HR arrays stored in an inner node having capacity 30 entries will consume $30 \cdot 50 \cdot 2 \cdot 4 = 12000$ bytes, while by using 1-byte approximations the HR arrays will take only $30 \cdot 50 \cdot 2 \cdot 1 = 3000$ bytes.

The intervals $\langle d_t^{low}, d_t^{up} \rangle$ assigned to the pivots P_t are stored, together with the pivots, in the pivot file.

5.3 Query Processing

Before processing a similarity query, the distances $d(Q, P_t)$, $\forall t \leq max(p_{hr}, p_{pd})$, have to be computed. During query processing, the PM-tree hierarchy is traversed down. Only if the metric region, described by a routing entry $rout(O_i)$, is overlapping the query region (Q, r_Q), the covering subtree $T(O_i)$ is relevant to the query, and thus it has to be further processed. A routing entry is relevant to the query just in case, that the query region overlaps *all* the hyper-rings represented by HR. Hence, prior to the standard hyper-sphere overlap check (used by M-tree), the overlap of hyper-rings HR$[t]$ against the query region is checked[2] (considering Lemma 5.2 and the definition of HR array) as follows:

$$\bigwedge_{t=1}^{p_{hr}} d(Q, P_t) - r_Q \leq \mathrm{HR}[t].\mathrm{max} \ \wedge \ d(Q, P_t) + r_Q \geq \mathrm{HR}[t].\mathrm{min} \tag{5.1}$$

If the above condition is false, the subtree $T(O_i)$ is not relevant to the query, thus can be discarded from further processing. In Figure 5.4 a range query situation is illustrated. Although the M-tree metric region cannot be discarded (see Figure 5.4a), the PM-tree region can be safely discarded, because the hyper-ring HR[2] is not overlapped (see Figure 5.4b).

Note: The condition (5.1) is only a *necessary* condition, it does not guarantee that a query region overlapping all the hyper-rings and the hyper-sphere is overlapping also the PM-tree region. Actually, the "geometric" shape of PM-tree region cannot be determined, since it is dependent on a particular metric d. The condition (5.1) is a *sufficient* condition just for a point query $(Q, 0)$.

Example 5.4

In Figure 5.6a see an intersection of two L_2-hyper-rings and a L_2-hyper-sphere. Even though the query region overlaps both the hyper-rings and the hyper-sphere, it does not overlap the intersection. On the other side, in Figure 5.6b see an intersection of two L_∞-hyper-rings and a L_∞-hyper-sphere, having the same properties as in the former case (the same values in HR, r_{O_i}, r_Q). Now, the query region overlaps also the intersection.

Nevertheless, the negation of overlap condition (5.1) can be used as a sufficient condition, that a query region *does not overlap* the PM-tree region.

At the leaf level, an irrelevant ground entry is determined such that the following condition is not satisfied:

$$\bigwedge_{t=1}^{p_{pd}} |d(Q, P_t) - \mathrm{PD}[t]| \leq r_Q \tag{5.2}$$

[2]Note that no additional distance computation is needed.

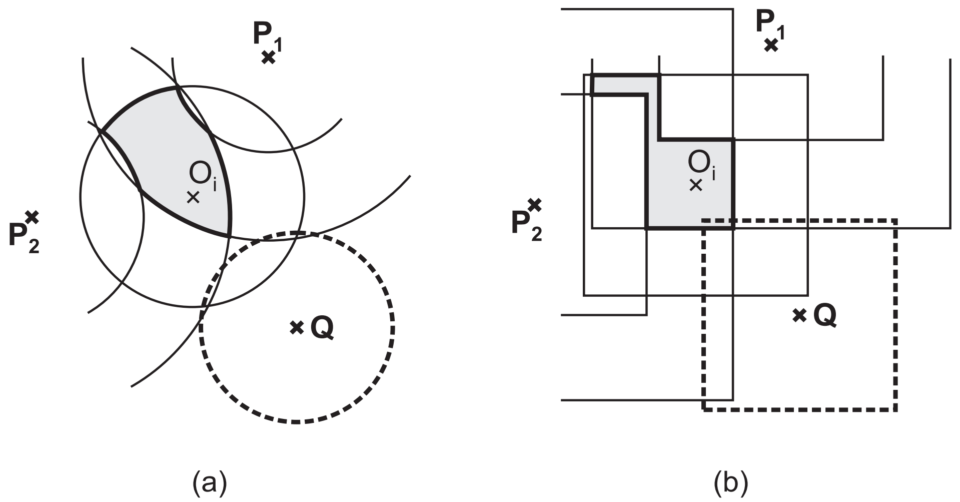


Figure 5.6 (a) Query region not overlapping the PM-tree region
(b) Query region overlapping the PM-tree region

The condition (5.2) provides the same functionality for the ground entries in PM-tree leaves, as is provided by LAESA-like filtering for the entire dataset $\mathbb{S}$.

5.3.1 Range Query Processing

The conditions (5.1) and (5.2) can be integrated into the original M-tree's range query as well as k-NN query algorithms. In case of range query, the adjustment is straightforward, see Listing 5.1.

The range query algorithm is executed by **RangeQuery** method, which must initially precompute the distances between Q and all of the pivots P_t. The recursive range query algorithm itself is subsequently executed by method **RangeQueryRec**. The operations **HROverlap** and **PDOverlap** provide the functionality of condition (5.1) and (5.2), respectively.

Listing 5.1 (PM-tree range query algorithm)

```
QueryResult **RangeQuery**(RQuery $(Q, r_Q)$)
{
  let dist be an array of $max(p_{hr}, p_{pd})$ floats
  **for** t=1 **to** $max(p_{hr}, p_{pd})$ **do** /* compute distances of $Q$ to all pivots $P_t$ */
    dist[t] = $d(P_t, Q)$
  **return RangeQueryRec**(root, $(Q, r_Q)$, dist)
}
```

```
QueryResult **RangeQueryRec**(Node $N$, RQuery $(Q, r_Q)$, float[] dist)
{
  let $O_p$ be the parent routing object of $N$
  if $N$ is not a leaf then {
    for each $rout(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q + r_{O_i}$ then { /* application of Lemma 3.2 */
        if HROverlap(HR, dist, $r_Q$) then { /* application of condition (5.1) */
          compute $d(O_i, Q)$
          if $d(O_i, Q) \leq r_Q + r_{O_i}$ then /* application of Lemma 3.1 */
            RangeQueryRec($ptr(T(O_i))$, $(Q, r_Q)$)
        }
      }
    } /* for each ... */
  } else {
    for each $grnd(O_i)$ in $N$ do {
      if $|d(O_p, Q) - d(O_i, O_p)| \leq r_Q$ then { /* application of Lemma 3.2 */
        if PDOverlap(PD, dist, $r_Q$) then { /* application of condition (5.2) */
          compute $d(O_i, Q)$
          if $d(O_i, Q) \leq r_Q$ then
            add $O_i$ to the query result
        }
      }
    } /* for each ... */
  }
} /* RangeQueryRec */

boolean HROverlap(HRarray HR, float[] dist, float $r_Q$)
{
  for t=1 to $p_{hr}$ do
    if dist[t] $- r_Q >$ HR[t].max OR dist[t] $+ r_Q <$ HR[t].min then /* by Lemma 5.2 */
      return false
  return true
}

boolean PDOverlap(float[] PD, float[] dist, float $r_Q$)
{
  for t=1 to $p_{pd}$ do
    if |dist[t] $-$ PD[t]| $> r_Q$ then
      return false
  return true
}
```

5.3.2 Nearest Neighbours Query Processing

In [99] we have proposed an optimal k-NN query algorithm for the PM-tree. We have modified the M-tree's k-NN query algorithm by revisiting the lower-bound distance d_{min} and the upper-bound distance d_{max}.

The requests $[ptr(T(O_i)), d_{min}(T(O_i))]$ in PR queue represent the still relevant subtrees $T(O_i)$, i.e. such subtrees, the parent metric regions of which overlap the dynamic query region (Q, r_Q). In the case of a PM-tree region, the lower-bound distance is further increased using the hyper-rings HR$[t]$ as follows:

$$d_{min}(T(O_i)) = max\{0, d(O_i, Q) - r_{O_i}, d^{low}_{HRmax}, d^{low}_{HRmin}\}$$

$$d^{low}_{HRmax} = max \bigcup_{t=1}^{p_{hr}} \{d(P_t, Q) - \mathrm{HR}[t].\mathrm{max}\}$$

$$d^{low}_{HRmin} = max \bigcup_{t=1}^{p_{hr}} \{\mathrm{HR}[t].\mathrm{min} - d(P_t, Q)\}$$

where $d^{low}_{HR} = max\{d^{low}_{HRmax}, d^{low}_{HRmin}\}$ determines the lower-bound distance between the query object Q and objects located in the farthest hyper-ring. In other words, when compared to M-tree's k-NN algorithm, the lower-bound distance $d_{min}(T(O_i))$ for a PM-tree region could be correctly increased due to the hyper-rings. The adjusted lower-bound distance d_{min} guarantees the property, that metric region of each request in PR always overlaps the dynamic query region (Q, r_Q).

The entries $[oid(O_i), d(Q, O_i)]$ or $[-, d_{max}(T(O_i))]$ in NN array represent the current k candidates for nearest neighbours (or at least upper-bound distances of the still relevant subtrees). In the case of a subtree $T(O_i)$, the upper-bound distance $d_{max}(T(O_i))$ can be further decreased by means of the hyper-rings HR$[t]$ as follows:

$$d_{max}(T(O_i)) = min\{d(O_i, Q) + r_{O_i}, d^{up}_{HR}\}$$

$$d^{up}_{HR} = min \bigcup_{t=1}^{p_{hr}} \{d(P_t, Q) + \mathrm{HR}[t].\mathrm{max}\}$$

where d^{up}_{HR} determines the upper-bound distance between the query object Q and objects located in the nearest hyper-ring.

The PM-tree's k-NN query algorithm, presented in Listing 5.2, is similar to that one presented for M-tree in Listing 3.2. The PM-tree version of the k-NN algorithm is adjusted similarly like the PM-tree range query algorithm:

- Initially, the distances between Q and all the pivots P_t have to be precomputed.

- Prior to each hyper-sphere overlap check, a hyper-rings overlap check (i.e. application of condition (5.1)) is performed.
- Another difference between the M-tree's and PM-tree's k-NN algorithms is in the construction of $d_{max}(T(O_i))$ and $d_{min}(T(O_i))$ bounds, but the construction is not specified in the listing explicitly.

Listing 5.2 (PM-tree k-NN query algorithm)

```
NodeSearch(Node $N$, kNNQuery $(Q,k)$)
{
  let $O_p$ be the parent routing object of node $N$, let dist be array of $max(p_{hr},p_{pd})$ floats
  for t=1 to $max(p_{hr},p_{pd})$ do /* compute distances of $Q$ to all pivots $P_t$ */
    dist[t] = $d(P_t,Q)$
  if $N$ is an internal node then {
    for each $rout(O_i)$ in $N$ do {
      if $|d(O_p,Q) - d(O_i,O_p)| \leq r_Q + r_{O_i}$ then { /* application of Lemma 3.2 */
        if HROverlap(HR, dist, $r_Q$) then { /* application of condition (5.1) */
          compute $d(O_i,Q)$
          if $d_{min}(T(O_i)) \leq r_Q$ then {
            insert $[ptr(T(O_i)), d_{min}(T(O_i))]$ to PR
            if $d_{max}(T(O_i)) < r_Q$ then {
              $r_Q$ = NNUpdate($[-, d_{max}(T(O_i))]$)
              remove from PR all requests for which $d_{min}(T(O_i)) > r_Q$
            }
          } /* if $d_{min}$( ...*/
        }
      }
    } /* for each ...*/
  } else { /* $N$ is a leaf */
    for each $grnd(O_i)$ in $N$ do {
      if $|d(O_p,Q) - d(O_i,O_p)| \leq r_Q$ then { /* application of Lemma 3.2 */
        if PDOverlap(PD, dist, $r_Q$) then { /* application of condition (5.2) */
          compute $d(O_i,Q)$
          if $d(O_i,Q) \leq r_Q$ then {
            $r_Q$ = NNUpdate($[oid(O_i), d(O_i,Q)]$)
            remove from PR all requests for which $d_{min}(T(O_i)) > r_Q$
          }
        }
      }
    } /* for each ...*/
  }
}
```

Theorem 5.1

The PM-tree's k-NN query algorithm **kNNQuery** is optimal in I/O costs, because it only accesses those PM-tree nodes, the metric regions of which overlap the query region $(Q, d(Q,\text{NN}[k].d_{max}))$. In other words, the I/O costs of a PM-tree k-NN query (Q, k) and I/O costs of the equivalent PM-tree range query $(Q, d(Q,\text{NN}[k].d_{max}))$ are the same.

Proof: The proof is identical to that of Theorem 3.1. The only difference is the way, in which $d_{max}(T(O_i))$ and $d_{min}(T(O_i))$ distance bounds are constructed. ■

Example 5.5

In Figure 5.7 see an example of a 2-NN query processing. The PM-tree hierarchy is the same as the M-tree hierarchy presented in Example 3.4, but the query processing behaves a bit differently. Although in this example both the M-tree's and the PM-tree's k-NN query algorithms access 4 nodes, searching the PM-tree saves one insertion into the PR queue.

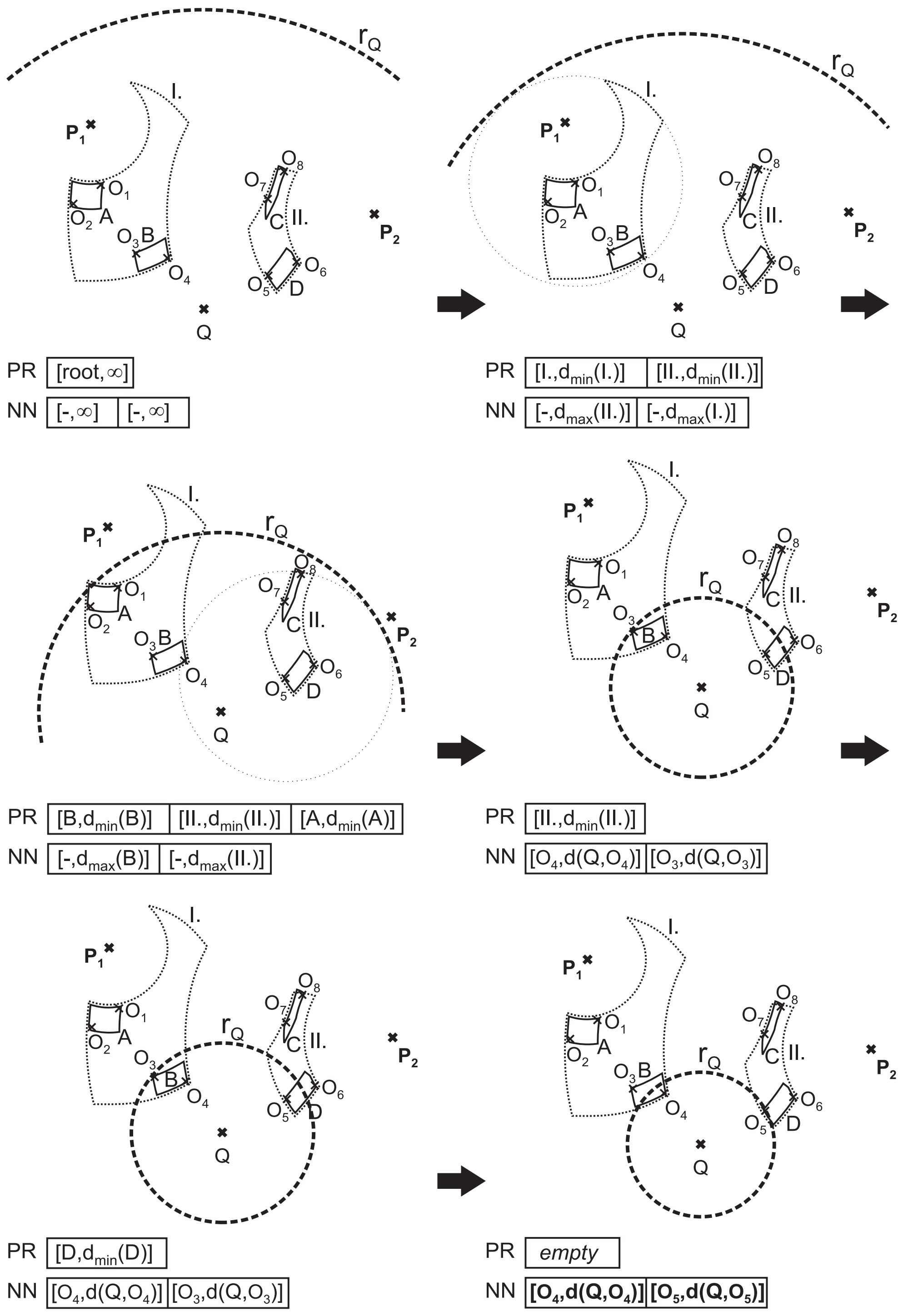

rQ
I.
P1
O1
O2 A
O3 B
O4
O7
O8
C II.
O6
O5 D
P2
Q
PR [root,∞]
NN [-,∞] [-,∞]
PR [I.,dmin(I.)] [II.,dmin(II.)]
NN [-,dmax(II.)] [-,dmax(I.)]
PR [B,dmin(B)] [II.,dmin(II.)] [A,dmin(A)]
NN [-,dmax(B)] [-,dmax(II.)]
PR [II.,dmin(II.)]
NN [O4,d(Q,O4)] [O3,d(Q,O3)]
PR [D,dmin(D)]
NN [O4,d(Q,O4)] [O3,d(Q,O3)]
PR empty
NN [O4,d(Q,O4)] [O5,d(Q,O5)]

Figure 5.7 Example of a 2-NN query processing in PM-tree

5.4 Building the PM-tree

The PM-tree is constructed the same way as is constructed the M-tree, i.e. the hyper-ring information stored in nodes is not used for an optimization of PM-tree hierarchy. Although, involvement of such an information into the construction algorithms is a subject of our future work.

In order to keep the HR and PD arrays up-to-date, the original M-tree construction algorithms [80, 98] have to be adjusted. We have to mention that the adjusted algorithms still preserve logarithmic time complexity.

5.4.1 Object Insertion

After a data object O_i is inserted into a leaf, the HR arrays of all routing entries in the insertion path are updated by values $d(O_i, P_t)$, $\forall t \leq p_{hr}$. In case of the leaf in the insertion path, the PD array stored in the new ground entry is populated by values $d(O_i, P_t), \forall t \leq p_{pd}$.

Listing 5.3 (dynamic object insertion into PM-tree)

```
Insert(Object $O_i$)
{
  let $N$ be the root node, let dist be an array of $max(p_{hr}, p_{pd})$ floats
  for t=1 to $max(p_{hr}, p_{pd})$ do /* compute distances of $O_i$ to all pivots $P_t$ */
    dist[t] = $d(P_t, Q)$
  TargetLeaf = FindLeaf($N$,$O_i$)
  store ground entry $grnd(O_i)$ in the TargetLeaf where PD = dist
  update HR arrays of all the parent routing entries of TargetLeaf by values stored in dist
  if TargetLeaf is overfull then
    Split(TargetLeaf)
}
```

A particular **FindLeaf** method (searching for an optimal leaf whereto insert the object O_i) is the same as for insertion into the M-tree (see Chapters 3,4).

5.4.2 Node Splitting

After a node is split, a new HR array for the left new routing entry is created by merging all appropriate intervals HR[t] (or computing HR in case of a leaf split) stored in routing entries (ground entries respectively) of the left new node. A new HR array for the right new routing entry is created similarly. These particular algorithmic modifications are to be integrated into the **Partition** operation used by the **Split** method, as presented in Listing 3.5.

5.5 Selecting the Pivots

The methods of selecting an optimal set of pivots have been intensively studied [90, 7, 28, 25] while, in general, we can say that a set of pivots is optimal, such that distances among pivots are maximal (close pivots give almost the same information), and the pivots are located outside the data clusters.

In the context of PM-tree, the optimal set of pivots causes that the hyper-spherical region is effectively "chopped off" by hyper-rings, such that the smallest overall volume of PM-tree regions (considering the volume of intersection of hyper-rings and the hyper-sphere) is obtained.

Example 5.6

In Figure 5.8 see two selections of pivots. In the first situation (Figure 5.8a), the pivots are close to each other, and both are located near the data clusters, thus appropriate hyper-rings do not reduce the hyper-spheres effectively. In the second situation (Figure 5.8b), the pivots are distant and outside the data clusters. In the second case, the hyper-spheres are reduced by hyper-rings more effectively.

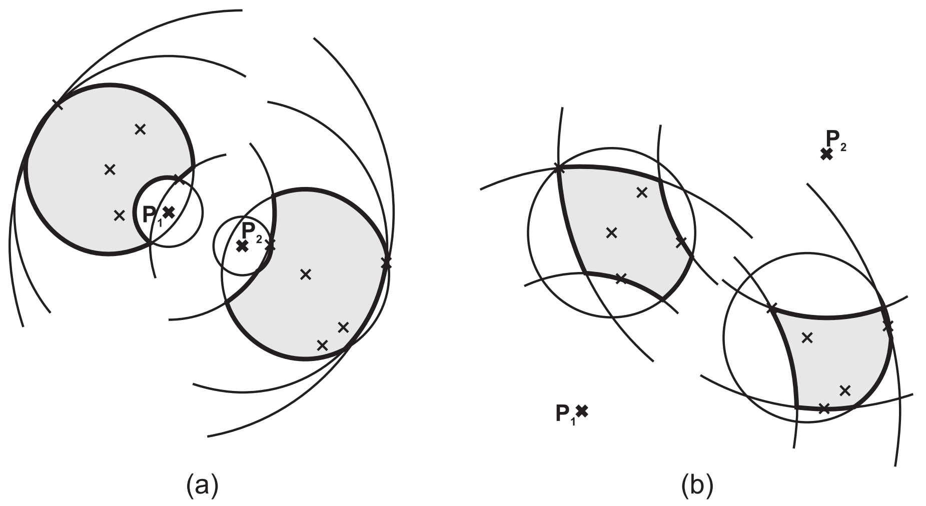


Figure 5.8 Pivot selection: (a) Not optimal (b) Optimal

5.5.1 Pivot Selection Methods

We briefly describe four heuristics of pivot selection. For two of them (proposed in [25]), an efficiency criterion μ_D has been introduced, measuring a quality of the selected pivots $P_t, t \leq p$. The criterion μ_D is estimated as the variance of distance distribution, considering distances $d(P_t, O_i), \forall O_i \in \mathbb{S}, t \leq p$, where the set of pivots P_t is fixed. In other words, when a high μ_D is achieved, each of the pivots P_t views the dataset from another "side". In context of PM-tree, a set of pivots with high μ_D allows to chop off the hyper-spherical regions effectively.

Random Selection

A simple method, denoted as `Random`, selects p pivots from the dataset $\mathbb{S}$ at random. This cheap method is suitable especially when searching in intrinsically high-dimensional datasets (for definition of intrinsic dimensionality see Chapter 6), where it practically does not matter which pivots are selected.

A more sophisticated method `RandomNgroups` chooses N groups of p pivots at random. For each of the groups, the criterion μ_D is calculated so that the group is selected, for which the μ_D value is maximal.

A cheaper variant of the previous method, denoted as `RandomNmax`, chooses N groups of p pivots at random. The group is selected, for which the sum of distances among pivots is maximal.

Incremental Selection

A pivot P_1 is selected from a sample of N objects of the dataset, such that the pivot alone has the maximum μ_D value. Then, a second pivot P_2 is chosen from another sample of N objects, such that set $\{P_1, P_2\}$ has the maximum μ_D value, considering P_1 fixed. The third pivot P_3 is chosen from another sample of N objects, such that $\{P_1, P_2, P_3\}$ has the maximum μ_D value, considering P_1 and P_2 fixed. The process is repeated until p pivots have been selected.

5.5.2 Selecting Pivots for the PM-tree

We can select the set of pivots in two ways, either statically or dynamically:

1. **Static selection**
   The pivots are sampled from the dataset $\mathbb{S}$ right before the PM-tree construction is started. However, such a selection requires the dataset to be available (at least partially) in advance.

2. **Dynamic selection**
   If the dataset is not available in advance, the pivots can be collected dynamically, being selected from the first k objects inserted into the PM-tree.

   The HR and PD arrays are not utilized (keeping them empty) until the k-th object is inserted into the PM-tree. Then, p pivots are selected among the k already indexed objects, and the HR and PD arrays are recomputed. Similarly, the pivot histograms used for the 1-byte approximation of object-to-pivot distances (see Section 5.2.1) can be recomputed after insertion of a sufficiently large number of objects. The usage of dynamic selection of pivots makes the PM-tree a completely dynamic MAM.

Note: Suppose the objects of $\mathbb{S}$ are inserted into PM-tree in a random order. Then pairwise distances among dynamically selected pivots will be similar to the distances among pivots selected statically using the `Random` method.

5.5.3 Re-pivoting

Since the structure of PM-tree hierarchy is not dependent on the particular selection of global pivots, the set of pivots can be replaced by a more optimal one (the PM-tree is ”re-pivoted”), achieving smaller region volumes and thus a higher retrieval efficiency. Then, since the pivots P_t have been replaced, the leaves of PM-tree have to be traversed, so that for each ground entry its PD array is recomputed as well as the HR arrays of all the parent routing entries. The re-pivoting can be suitable:

- when a more optimal set of pivots can be determined, e.g. when a larger part of the dataset is available
- after an application of the M-tree's slim-down algorithm (see Section 4.1), which does not keep the HR and PD arrays up-to-date

Example 5.7
In Figure 5.9a see PM-tree regions reduced by two hyper-rings centered in pivots P_1 and P_2. Since a more optimal pair of pivots is available, the PM-tree is re-pivoted, achieving smaller volumes of regions A,B,D,II., see Figure 5.9b.

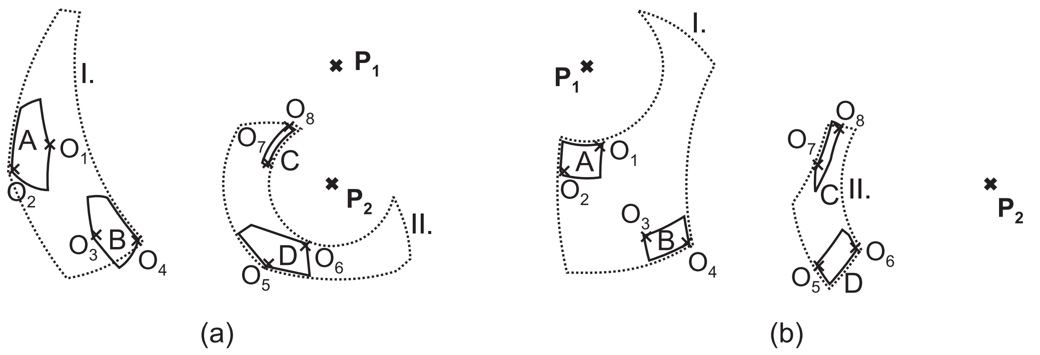


Figure 5.9 PM-tree regions: (a) Before re-pivoting (b) After re-pivoting

5.6 Cost Models

In this section, we present node-based and level-based cost models for query processing in PM-tree, allowing to predict the search costs. Since PM-tree is an extension of M-tree, we have extended the original cost models developed for M-tree [38]. Like M-tree cost models, also the PM-tree cost models are conditioned by the following assumptions:

- The only information used is the *distance distribution* of objects in a given dataset, because no information about *data distribution* is known.

- A *biased* query model is considered, i.e. the distribution of query objects is equal to that of data objects.
- The dataset $\mathbb{S}$ is supposed to have high "homogeneity of viewpoints" (for details we refer to [38, 80]).

The basic tool used in the cost models is a probability, that two hyper-spheres overlap, i.e.

$$\mathbf{Pr}\{\text{spheres } (O_1, r_{O_1}) \text{ and } (O_2, r_{O_2}) \text{ overlap}\} = \mathbf{Pr}\{d(O_1, O_2) \leq r_{O_1} + r_{O_2}\}$$

where O_1, O_2 are center objects and r_{O_1}, r_{O_2} are covering radii of the hyper-spheres. For this purpose, the *overall distance distribution* function is used, defined as:

$$F(x) = \mathbf{Pr}\{d(O_i, O_j) \leq x\}, \forall O_i, O_j \in \mathbb{U}$$

and also the *relative distance distribution* function is used, defined as:

$$F_{O_k}(x) = \mathbf{Pr}\{d(O_k, O_i) \leq x\}, O_k \in \mathbb{U}, \forall O_i \in \mathbb{U}$$

For an approximate F or F_{O_k} evaluation, a set $\mathcal{O}$ of s objects $\mathcal{O}_i \in \mathbb{S}$ is sampled. A value of F is computed using the $s \times s$ matrix of pairwise distances between objects in $\mathcal{O}$. For evaluation of function F_{O_k}, only the vector of s distances $d(\mathcal{O}_i, O_k)$ is needed.

5.6.1 Node-based Cost Model

In the node-based cost model (NB-CM), a probability of access to each PM-tree node is predicted. Basically, a node N is accessed if its metric region (described by the parent routing entry of N) overlaps the query hyper-sphere (Q, r_Q):

$$\mathbf{Pr}\{\text{node N is accessed}\} = \mathbf{Pr}\{\text{metric region of N is overlapped by } (Q, r_Q)\}$$

Specifically, a PM-tree node N is accessed if all the components of its metric region (i.e. the hyper-sphere and p_{hr} hyper-rings) overlap the query hyper-sphere:

$$\mathbf{Pr}\{\text{N is accessed}\} =$$

$$= \mathbf{Pr}\{\text{hyper-sphere is overlapped}\} \cdot \prod_{t=1}^{p_{hr}} \mathbf{Pr}\{t\text{-th hyper-ring is overlapped}\}$$

and finally (for a query radius r_Q and the parent routing entry $rout(O_i)$ of N)

$$\mathbf{Pr}\{\text{N accessed}\} \approx F(r_{O_i}+r_Q) \cdot \prod_{t=1}^{p_{hr}} F_{P_t}(\text{HR}_{O_i}[t].\text{max}+r_Q) \cdot (1-F_{P_t}(r_Q-\text{HR}_{O_i}[t].\text{min}))$$

In order to determine the estimated I/O costs for a range query, it is sufficient to sum the above probabilities over all the m nodes N_i in the PM-tree:

$nodes_{NB}(RQ(Q, r_Q)) = \sum_{i=1}^{m} \mathbf{Pr}\{\text{node } N_i \text{ is accessed}\} =$

$$= \sum_{i=1}^{m} F(r_{O_i} + r_Q) \cdot \prod_{t=1}^{p_{hr}} F_{P_t}(\mathrm{HR}_{O_i}[t].\max + r_Q) \cdot (1 - F_{P_t}(r_Q - \mathrm{HR}_{O_i}[t].\min))$$

The computation costs[3] are estimated considering the probability that a node is accessed, multiplied by the number of its entries, $e(N_i)$, thus obtaining

$dists_{NB}(RQ(Q, r_Q)) =$

$$= \sum_{i=1}^{m} e(N_i) \cdot F(r_{O_i} + r_Q) \cdot \prod_{t=1}^{p_{hr}} F_{P_t}(\mathrm{HR}_{O_i}[t].\max + r_Q) \cdot (1 - F_{P_t}(r_Q - \mathrm{HR}_{O_i}[t].\min))$$

In order to formulate cost models for k-NN queries, it is sufficient to estimate the distance between Q and the k-th nearest neighbour, and apply the distance as a query radius to the range query cost models.

Let $\mathbf{nn}_{Q,k}$ be a random variable, standing for the distance between the k-th nearest neighbour and Q. The probability that $\mathbf{nn}_{Q,k}$ is at most r, is equal to the probability that at least k objects are inside the hyper-sphere (Q, r), that is

$$\begin{aligned} P_{Q,k}(r) &= \mathbf{Pr}\{\mathbf{nn}_{Q,k} \le r\} = \\ &= \sum_{i=k}^{n} \binom{n}{i} \cdot \mathbf{Pr}\{d(Q, O_j) \le r\}^i \cdot \mathbf{Pr}\{d(Q, O_j) > r\}^{n-i} \approx \\ &\approx 1 - \sum_{i=0}^{k-1} \binom{n}{i} \cdot F(r)^i \cdot (1 - F(r))^{n-i} \end{aligned}$$

The expected k-th nearest neighbour distance $E[\mathbf{nn}_{Q,k}]$ is computed by integration of $P_{Q,k}(r)$ over all r values, and subtracting it from the upper-bound distance d^+:

$$E[\mathbf{nn}_{Q,k}] = d^+ - \int_0^{d^+} P_{Q,k}(r)\ \mathbf{dr}$$

In order to determine the estimated I/O costs for a k-NN query, the estimated k-NN distance $E[\mathbf{nn}_{Q,k}]$ is applied to the range query cost model as the query radius:

$$nodes_{NB}(NN(Q, k)) = nodes_{NB}(RQ(Q, E[\mathbf{nn}_{Q,k}]))$$

The estimated computation costs for a k-NN query are determined similarly:

$$dists_{NB}(NN(Q, k)) = dists_{NB}(RQ(Q, E[\mathbf{nn}_{Q,k}]))$$

[3]Optimizations utilizing the precomputed distances (see Section 3.1) are not considered here.

5.6.2 Level-based Cost Model

The problem with NB-CM is that maintaining statistics for every node is very time consuming when the PM-tree index is large. To overcome this, we consider a simplified level-based cost model (LB-CM), which uses only average information collected for each level of the PM-tree. For each level l of the tree ($l = 0$ for the root level, $l = h - 1$ for the leaf level), LB-CM uses this information: m_l (the number of nodes at level l), $\overline{r_l}$ (the average value of covering radius considering all the nodes at level l), $\overline{\mathrm{HR}_l[i].\mathrm{min}}$ and $\overline{\mathrm{HR}_l[i].\mathrm{max}}$ (the average information about hyper-rings considering all the nodes at level l). Given these statistics, the number of nodes accessed by a range query can be estimated as

$$nodes_{LB}(RQ(Q, r_Q)) \approx$$

$$\approx \sum_{l=0}^{h-1} m_l \cdot F(\overline{r_l} + r_Q) \cdot \prod_{t=1}^{p_{hr}} F_{P_t}(\overline{\mathrm{HR}_l[t].\mathrm{max}} + r_Q) \cdot (1 - F_{P_t}(r_Q - \overline{\mathrm{HR}_l[t].\mathrm{min}}))$$

Similarly, we can estimate computation costs as

$$dists_{LB}(RQ(Q, r_Q)) \approx$$

$$\approx \sum_{l=0}^{h-1} m_{l+1} \cdot F(\overline{r_l} + r_Q) \cdot \prod_{t=1}^{p_{hr}} F_{P_t}(\overline{\mathrm{HR}_l[t].\mathrm{max}} + r_Q) \cdot (1 - F_{P_t}(r_Q - \overline{\mathrm{HR}_l[t].\mathrm{min}}))$$

where $m_h \overset{\mathrm{def}}{=} n$ is the number of indexed objects.
The number of nodes accessed by a k-NN query can be estimated using LB-CM as

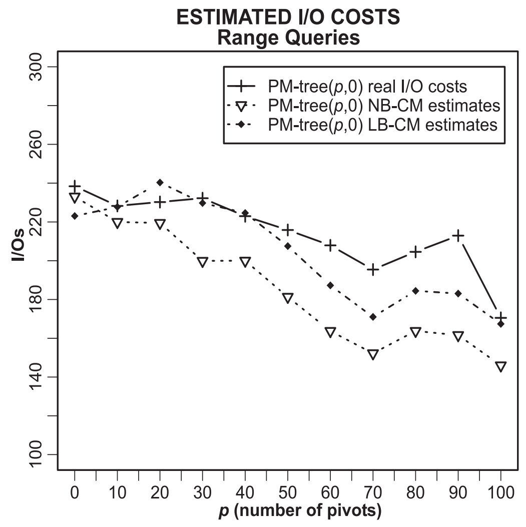
$$nodes_{LB}(NN(Q, k)) = nodes_{LB}(RQ(Q, E[\mathbf{nn}_{Q,k}]))$$

The estimated computation costs for a k-NN query can be estimated as

$$dists_{LB}(NN(Q, k)) = dists_{LB}(RQ(Q, E[\mathbf{nn}_{Q,k}]))$$

5.6.3 Experimental Evaluation

In order to evaluate accuracy of the presented cost models, we performed several experiments on a synthetic dataset. The dataset consisted of 10,000 10-dimensional tuples (embedded inside the unitary hyper-cube), uniformly distributed among 100 L_2-spherical clusters of diameter $\frac{d^+}{10}$ ($d^+ = \sqrt{10}$). The labels "`PM-tree(`x`,`y`)`" in the graphs below are described in Section 5.7.



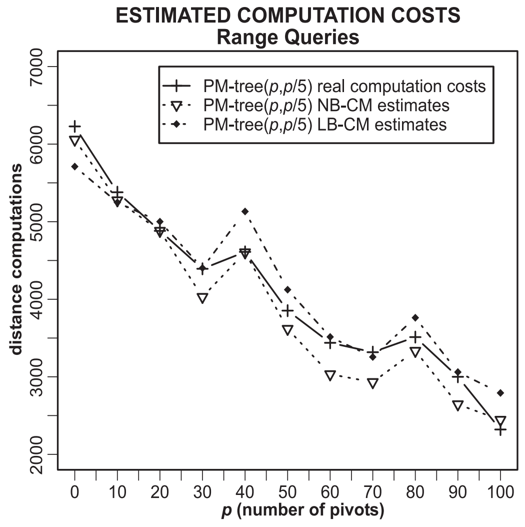
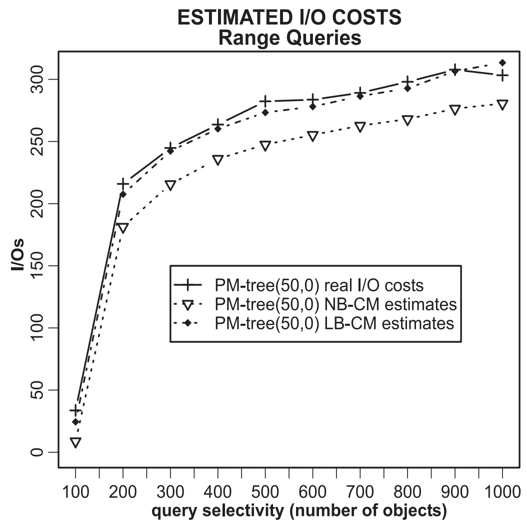


Figure 5.10 Range queries: (a) Estimated I/O costs
(b) Estimated computation costs

Range Queries

The first set of experiments investigated the accuracy of estimated costs for range queries. The range query selectivity (the average number of objects in the query result) was set to 200. In Figure 5.10a we present the estimated I/O costs as well as the real I/O costs, related to the increasing number of pivots used by the PM-tree. The relative error (i.e. $err = 1 - \frac{min(\text{estimated costs,real costs})}{max(\text{estimated costs,real costs})}$) of NB-CM estimates was below 0.2. Surprisingly, the relative error of LB-CM estimates was smaller than for NB-CM, below 0.15.



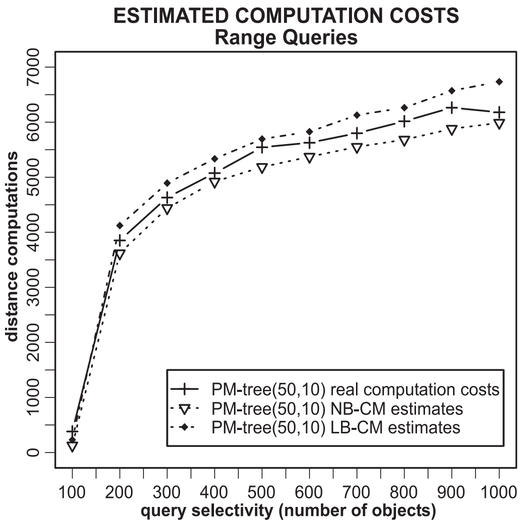


Figure 5.11 Range queries: (a) Estimated I/O costs
(b) Estimated computation costs

The estimates of computation costs (see Figure 5.10b) were even more accurate than for the I/O costs estimates, below 0.05 (for NB-CM) and 0.04 (for LB-CM).

In Figure 5.11 see the estimated and the real costs, according to the increasing query selectivity. The relative error of NB-CM I/O costs estimates (see Figure 5.11a) was below 0.1. Again, the relative error of LB-CM estimates was very small, below 0.02. The error of computation costs (see Figure 5.11b) was below 0.07 (for NB-CM) and 0.05 (for LB-CM).

k-NN Queries

The second set of experiments was focused on the accuracy of estimated costs for *k*-NN queries. In Figure 5.12 the estimated I/O costs as well as the computation costs are presented, according to the increasing number of neighbours.

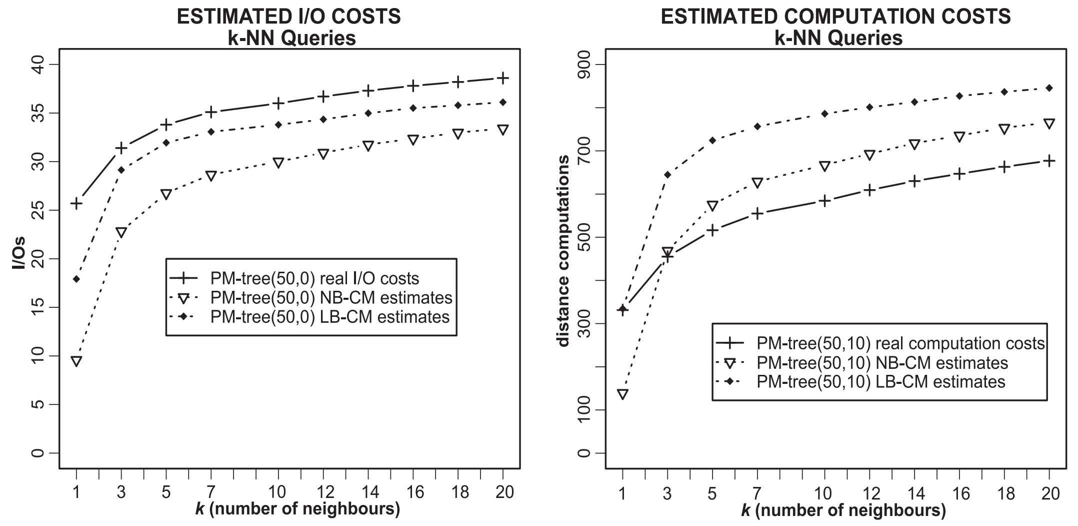


Figure 5.12 *k*-NN queries: (a) Estimated I/O costs
(b) Estimated computation costs

We can observe that the estimated costs for *k*-NN queries were less accurate than those for range queries. This was caused by the estimated distance $E[\mathbf{nn}_{Q,k}]$ to the *k*-th nearest neighbour, which was slightly underestimated. Hence, the I/O costs were underestimated as well. The computation costs were overestimated as a consequence of the fact, that *k*-NN query is not optimal in distance computations, while the *k*-NN query cost models are based on range query cost models.

5.7 Experimental Results

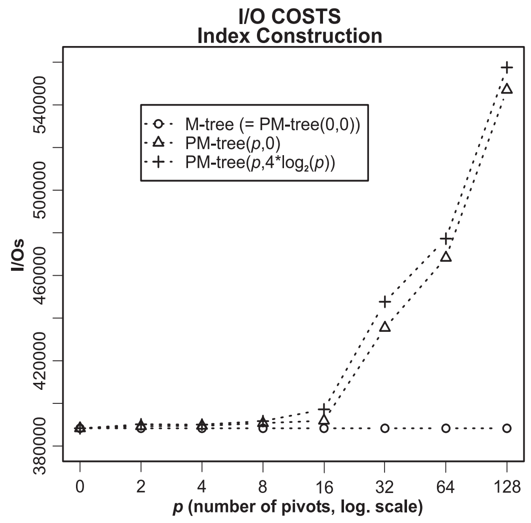
In order to evaluate the overall PM-tree performance, we present some results of experiments made on large synthetic as well as real-world vector datasets. In most of the experiments, the search efficiency of range queries and *k*-NN queries processing was examined. The query objects were randomly selected from the respective dataset, while each particular query test consisted of 1000 queries of the same query selectivity. In all experiments the Euclidean distance was used. For selection of pivots P_t the static `RandomNmax` method ($N = 10,000$) was used.

Abbreviations in Figures

Each label of form "`PM-tree(`x`,`y`)`" stands for a PM-tree index, where $p_{hr} = x$ and $p_{pd} = y$. A label "$\langle$*index*$\rangle$ `+ SlimDown`" denotes an index subsequently post-processed using the generalized slim-down algorithm (see Section 4.1).

5.7.1 Synthetic Datasets

For the first set of experiments, a collection of 8 synthetic vector datasets of increasing dimensionality (from $D = 4$ to $D = 60$) was generated. Each dataset (embedded inside unitary hyper-cube) consisted of 100,000 D-dimensional tuples, uniformly distributed among 1000 L_2-spherical uniformly distributed clusters. The diameter of each cluster was set to $\frac{d^+}{10}$, where $d^+ = \sqrt{D}$. These datasets were indexed by PM-tree (for various p_{hr} and p_{pd}) as well as by M-tree. Some statistics about the created indices are described in Table 5.1.



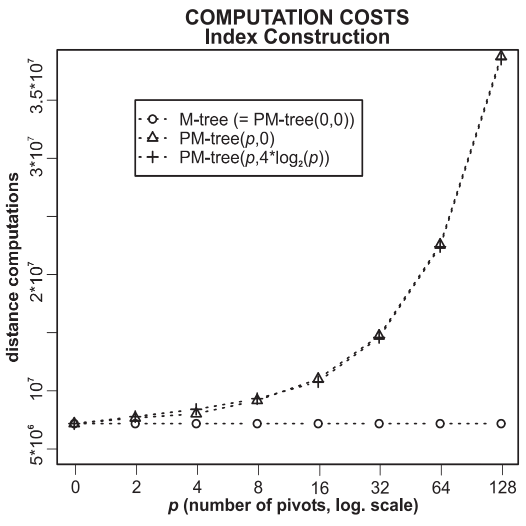


Figure 5.13 Construction costs: (a) I/O costs (b) Computation costs

Construction methods:	`SingleWay + minMAX_RAD (+ SlimDown)`		
Dimensionalities:	4,8,16,20,30,40,50,60	Inner node capacities:	10 – 28
Index file sizes:	4.5 MB – 55 MB	Leaf node capacities:	16 – 36
Pivot file sizes[4]:	2 KB – 17 KB	Avg. node utilization:	66%
Node (disk page) sizes:	1 KB ($D = 4, 8$), 2 KB ($D = 16, 20$), 4 KB ($D \geq 30$)		

Table 5.1 PM-tree index statistics (synthetic datasets)

Index Construction

Index construction costs (for 30-dimensional indices), according to the increasing number of pivots, are presented in Figure 5.13. The I/O costs for PM-tree indices with up to 16 pivots were similar to those for `M-tree` index (see Figure 5.13a). For `PM-tree(128, 0)` and `PM-tree(128, 28)` indices, the I/O costs were about 1.4 times higher than those for the `M-tree` index. The increasing trend of computation costs (see Figure 5.13b) depended mainly on the p object-to-pivot distance computations performed during each object insertion – additional distance computations were needed after leaf splitting, in order to create HR arrays of the new routing entries.

Range Queries

In Figure 5.14 the range query costs (for 30-dimensional indices and query selectivity 50 objects) according to the number of pivots are presented. The I/O costs

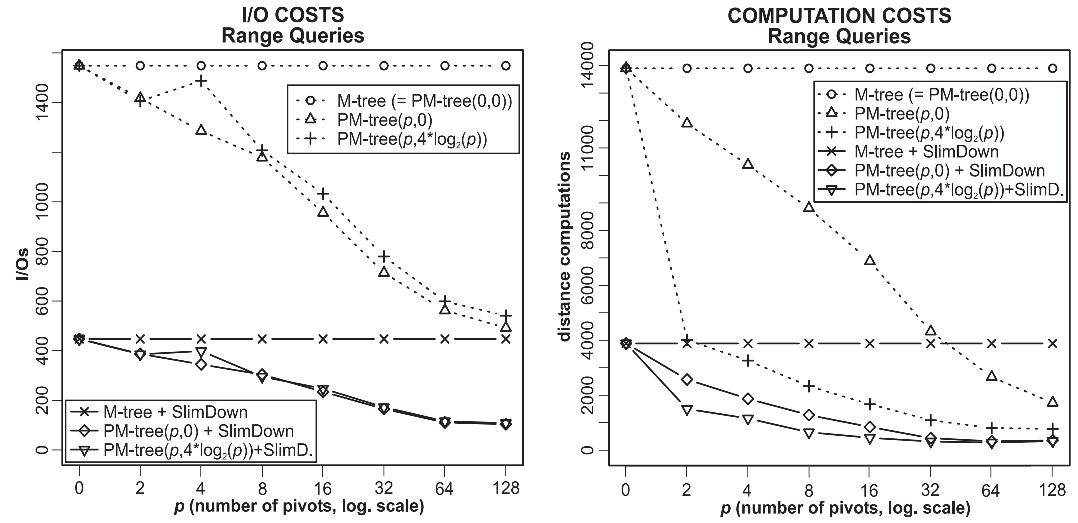


Figure 5.14 Range queries: (a) I/O costs
(b) Computation costs

rapidly decreased with the increasing number of pivots. The `PM-tree(128, 0)` and `PM-tree(128, 28)` indices needed only 27% of I/O costs spent by the `M-tree` index. Moreover, the PM-tree was superior even after the slim-down algorithm post-processing, e.g. the "slimmed" `PM-tree(128, 0)` index needed only 23% of I/O costs spent by the slimmed `M-tree` index (and only 6.7% of I/O costs spent by the not slimmed `M-tree`). The decreasing trend of computation costs was even more steep than for I/O costs, the `PM-tree(128, 28)` index needed only 5.5% of the `M-tree` computation costs.

[4]Access costs to the pivot files, storing pivots P_t and the scaling intervals for all pivots (see Section 5.2.1), were not considered because of their negligible sizes.

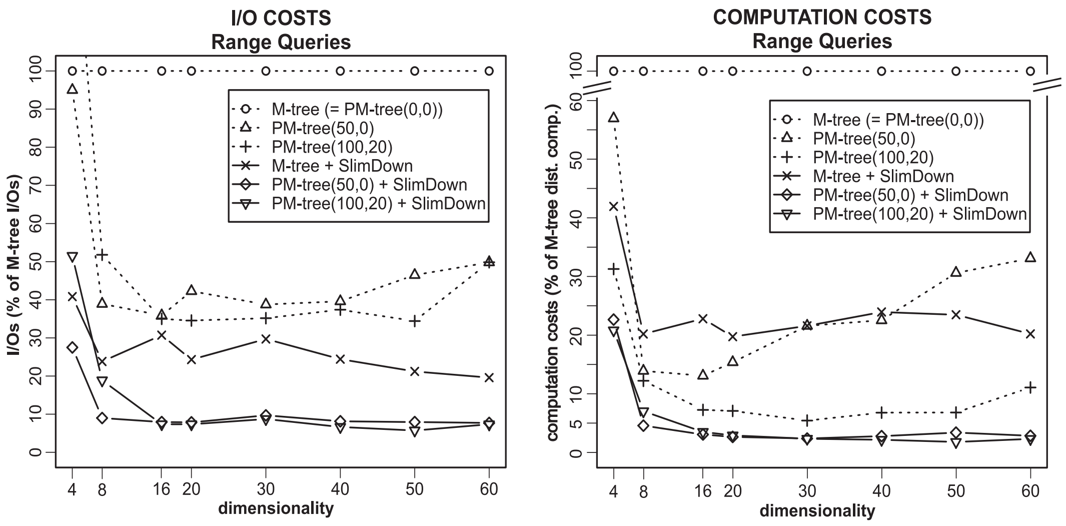


Figure 5.15 Range queries: (a) I/O costs
(b) Computation costs

The influence of increasing dimensionality D is depicted in Figure 5.15. Since the disk page sizes for different indices grew with the increasing dimensionality, the presentation of I/O costs as well as the computation costs of PM-tree indices is related (in percent) to the I/O costs (CC resp.) of M-tree indices. For $8 \leq D \leq 40$ the I/O costs stayed approximately fixed, for $D > 40$ they slightly increased.

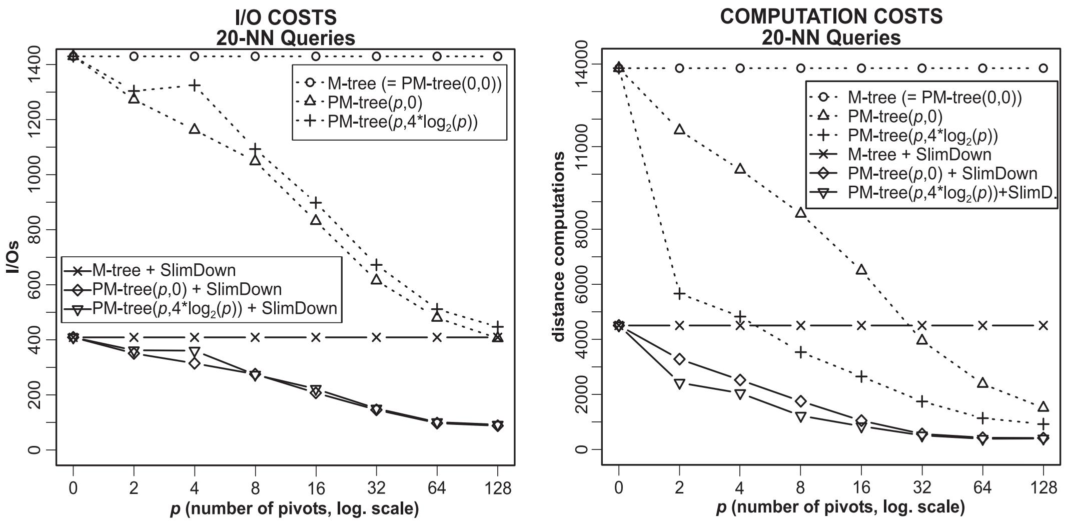


Figure 5.16 20-NN queries: (a) I/O costs
(b) Computation costs

k-NN Queries

Since the *k*-NN algorithm is optimal in I/O costs, the results for *k*-NN queries are similar to those for range queries. In Figure 5.16 the 20-NN query costs (for 30-dimensional indices) according to the number of pivots are presented. The I/O costs as well as the computation costs rapidly decreased with the increasing number of pivots. The influence of increasing D is depicted in Figure 5.17.

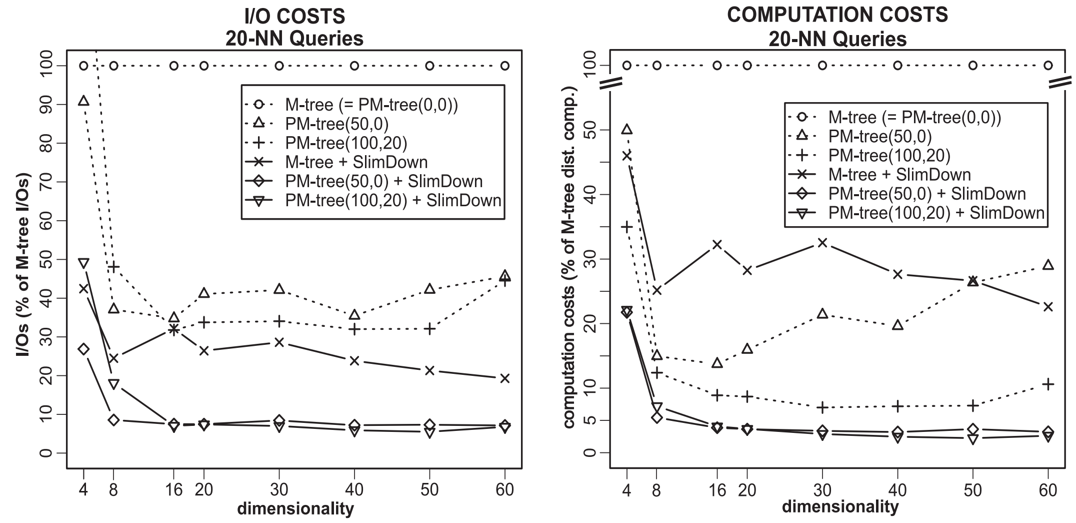


Figure 5.17 20-NN queries: (a) I/O costs
(b) Computation costs

5.7.2 Image Database

For the second set of experiments, a collection of approximately 10,000 web-crawled images [109] was used (see a sample in Figure 5.18). Each image was converted into a 256-level gray scale, and a gray frequency histogram (256-dimensional vector) was extracted and indexed. See the statistics about image indices in Table 5.2.

Construction methods:	`SingleWay + minMAX_RAD (+ SlimDown)`		
Dimensionality:	256	Inner node capacities:	10 – 31
Index file sizes:	16 MB – 20 MB	Leaf node capacities:	29 – 31
Pivot file sizes:	4 KB – 1 MB	Avg. node utilization:	67%
Node (disk page) size:	32 KB		

Table 5.2 PM-tree index statistics (image database)

Range Queries

In Figure 5.19 the range query costs (for query selectivity 20 objects) according to increasing number of pivots are presented. We can observe that e.g. the

Figure 5.18 A sample of WBIIS image database

"slimmed" `PM-tree`(1024,50) index consumed only 42% of I/O costs spent by the slimmed `M-tree` index, see Figure 5.19a. The computation costs (see Figure 5.19b) for $p \leq 64$ decreased down to 36% of `M-tree` computation costs.

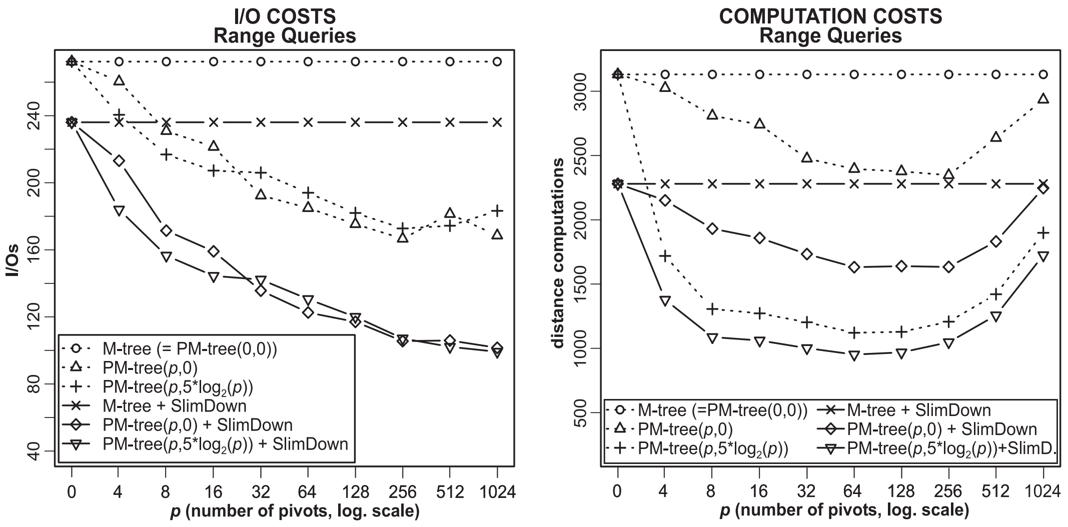
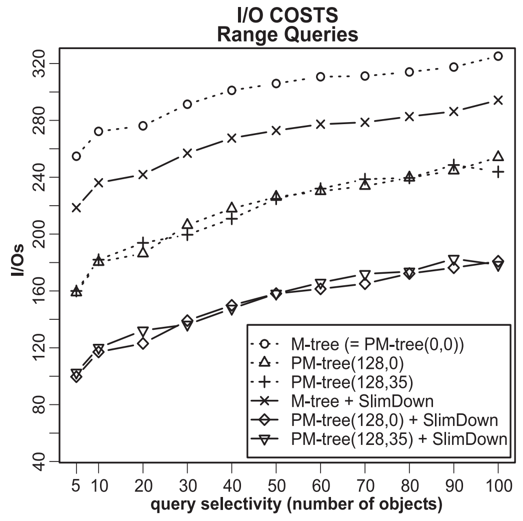


Figure 5.19 Range queries: (a) I/O costs
(b) Computation costs

However, for $p > 64$ the overall computation costs grew, because the number of computed query-to-pivot distances (i.e. p distance computations for each query) was proportionally too large. Nevertheless, this fact is dependent on the database size – for 100,000 objects the proportion of p query-to-pivot distance computations would be smaller, when compared to the overall computation costs.

The range query costs according to increasing selectivity are presented in Figure 5.20. The I/O costs stayed below 73% of `M-tree` I/O costs (below 58% in case of slimmed indices). The computation costs stayed below 43% (49% resp.).



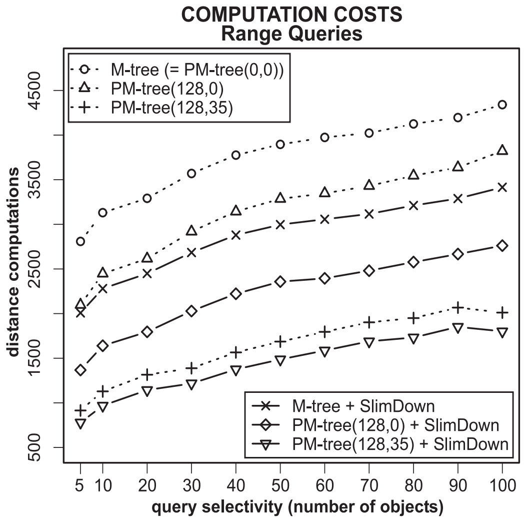
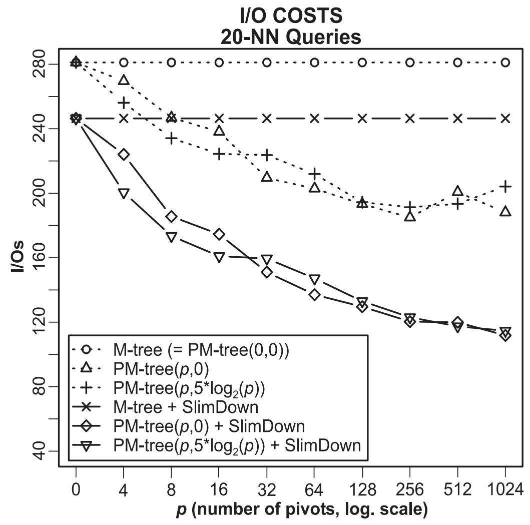


Figure 5.20 Range queries: (a) I/O costs (b) Computation costs

k-NN Queries

In Figure 5.21 the *k*-NN query costs for increasing number of pivots are presented. Again, the results were similar to those for range queries. The query costs ac-



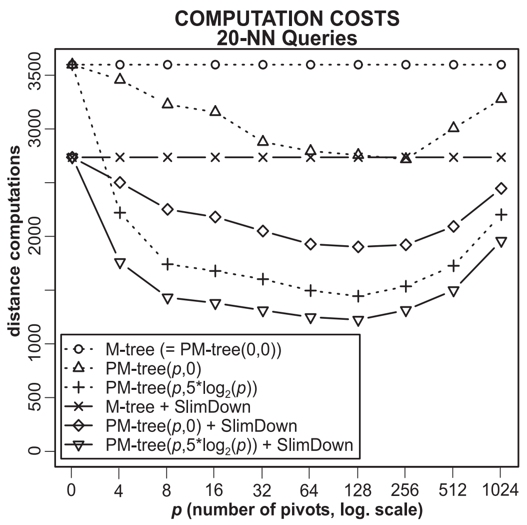


Figure 5.21 20-NN queries: (a) I/O costs (b) Computation costs

cording to increasing number of nearest neighbours are presented in Figure 5.22.

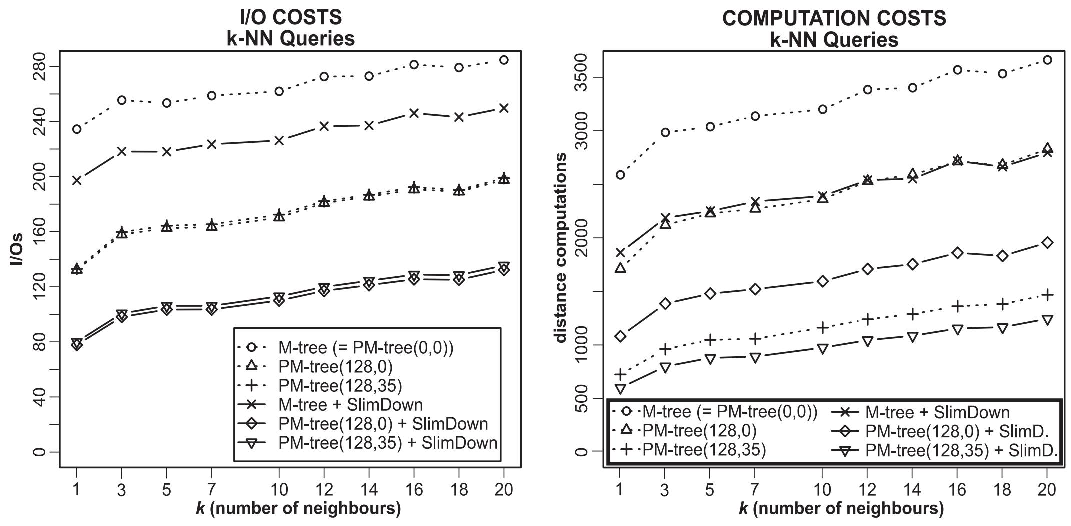


Figure 5.22 20-NN queries: (a) I/O costs (b) Computation costs

Range Queries vs. k-NN Queries

In the last experiment, we examined the computation costs for 50-NN queries as well as for the equivalent range queries of selectivity 50 objects. Since the k-NN query algorithm is not guaranteed to be optimal in computation costs, the costs for k-NN queries have been supposed to be a little higher than the costs of range queries. As we can see in Figure 5.23, the hypothesis was fairly verified.

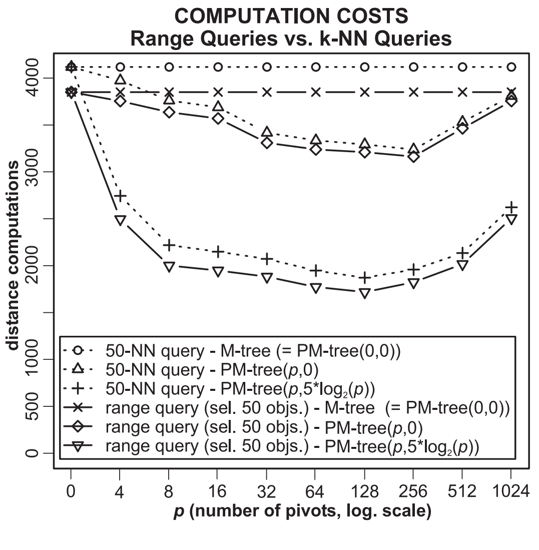


Figure 5.23 Range queries vs. k-NN queries – Computation costs

Moreover, an interesting trend of computation costs can be observed with respect to the ratio of k-NN query costs to the range query costs. For `PM-tree(0,0)` index (i.e. the `M-tree` index) the ratio was about 107%, while for the `PM-tree(`p`,0)` index the ratio was decreasing down to 101.4%.

5.7.3 Summary

The results of experiments, made on synthetic datasets and mainly on the image database, have demonstrated the general benefits of PM-tree. The index construction (object insertion respectively) is dynamic and still preserves the logarithmic time complexity. For suitably high p_{hr} and p_{pd}, the index size growth is minor, which is true especially for high-dimensional datasets (e.g. for the 256-dimensional image dataset), where the size of pivoting information stored in ground/routing entries is negligible when compared to the size of the data object (i.e. the vector) itself. A particular (but not serious) limitation of the PM-tree is that a part of the dataset must be known in advance for the static selection of pivots, and when using object-to-pivot distance distribution histograms.

Furthermore, the PM-tree could serve as a constructionally much cheaper alternative to the slim-down algorithm on M-tree – the above presented experimental results have shown that the search efficiency of PM-tree (having sufficiently high p_{hr}, p_{pd}) is comparable or even better than an equivalent "slimmed" M-tree. Finally, a combination of PM-tree and the slim-down algorithm makes the PM-tree a very efficient metric access method.

5.8 Future Work

In the future, we plan to adjust the dynamic insertion algorithm in a way that also the hyper-ring information will be used for the decision about the optimal target leaf choice. Such a modification should improve the quality of PM-tree hierarchy, leading to a more efficient query processing.

Second, the above presented experimental results were performed on PM-trees, where the entire dataset $\mathbb{S}$ was available before PM-tree construction. In order to verify the strength of PM-tree as a completely dynamic MAM, there must be designed and implemented various policies of selecting the pivots dynamically, as it has been mentioned in Section 5.5. We also plan to examine another pivot selection techniques, besides the `RandomNmax` method.

Part II

Approximate Search

Chapter 6

Approximate Similarity Search

In this part, we are concerned with metric access methods providing a kind of *approximate search*. Unlike the exact MAMs (discussed in Chapter 2), the approximate MAMs are aimed to efficiently search in a given dataset such that a small proportion of false drops is allowed.

6.1 Motivation

In various areas of Information Retrieval, the models of similarity are inherently inexact (or fuzzy) and, consequently, searching using a similarity measure is more or less approximate. In order to evaluate the subjective effectiveness of IR models, there are widely used two classifiers, the *precision* P and the *recall* R, defined for a query Q and a dataset $\mathbb{S}$ as $P = \frac{R_Q}{A_Q}$, $R = \frac{R_Q}{R_F}$, where R_Q is the number of relevant objects in the query result, R_F is the number of all relevant objects in the dataset $\mathbb{S}$, and A_Q is the total number of objects in the query result. The parameters R_Q, R_F are obviously subjective, i.e. assessed by human. In real applications, the precision/recall values do never reach up to 100% while, moreover, the query object Q itself is often a "guess". From this point of view, a second approximation involved in the approximate search methods may be acceptable, since it can only (slightly) lower the precision/recall values.

On the other side, in order to compensate the lack of inaccuracy, the approximate search methods are expected to be much more efficient than the exact MAMs. Moreover, the approximate MAMs are sometimes the only efficient solution for searching in metric datasets, which are poorly *intrinsically* structured. In this chapter we discuss just the approximate search in poorly structured datasets.

6.2 Intrinsic Dimensionality

A property common to all MAMs (either exact or approximate) is an ability to recognize a kind of "diversity" present within the indexed dataset. This diversity has to be transformed by a MAM into a metric index, that allows to discard irrelevant parts of the dataset for a given query. However, if there is only a low diversity present in the dataset, the exact MAMs fail in both creating the structure and, mainly, in searching efficiently.

The diversity in metric dataset can be formalized by a concept of intrinsic dimensionality [32], a generalization of the classic topological (also called "embedding") dimensionality into metric spaces. There have appeared several definitions of intrinsic dimensionality (e.g. the fractal dimensionality [51]), however, we use the concept [31] based on statistical analysis of distance distribution histogram.

Definition 6.1 (intrinsic dimensionality)
Let H be a distance distribution histogram of $\mathbb{S}$ (denoted also as d-DDH), considering distances $d(O_i, O_j)$ between all pairs of objects $O_i, O_j \in \mathbb{S}$. Then

$$\rho = \frac{\mu^2}{2\sigma^2}$$

is called the *intrinsic dimensionality* of dataset $\mathbb{S}$. The μ and σ^2 are the mean and the variance of the distance distribution histogram H. □

Note: If all the pairs of indexed objects are almost equally distant, then intrinsic dimensionality of $\mathbb{S}$ is high (i.e. the mean is high and the variance is low), which means the dataset is poorly intrinsically structured.

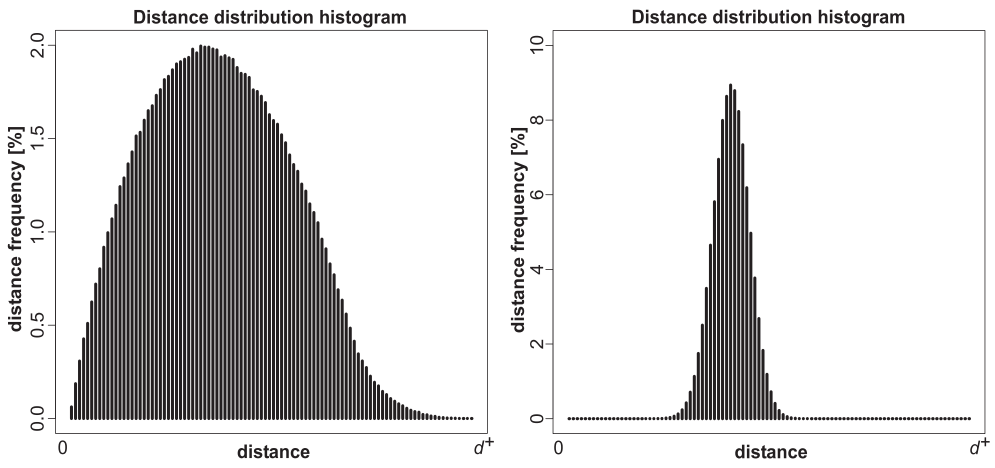


Figure 6.1 DDHs indicating (a) low (b) high intrinsic dimensionality

Example 6.1
In Figure 6.1 see DDHs of two metric datasets. We can observe that first dataset is well structured, because variance of the distances is high – represented by the wide bell in the histogram. On the other side, the intrinsic dimensionality of the second dataset is high, because the variance of distances is low – reflected by the narrow bell in the histogram. Generally, the more narrow and/or right-shifted is the bell in histogram, the higher is also intrinsic dimensionality of $\mathbb{S}$.

6.2.1 Curse of Dimensionality

In the area of multi-dimensional indexing, the spatial access methods suffer from a similar problem, known as *curse of dimensionality* [18, 32], causing that exact SAMs become inefficient for searching in high-dimensional[1] vector datasets. Instead of general metric search in D-dimensional spaces, most of the SAMs have been developed to support *window queries*, through which a vector dataset can be searched for objects located inside a hyper-rectangle $[a_1, a_2, ..., a_D][A_1, A_2, ..., A_D]$. However, processing a window query is equivalent to processing a metric range query, where the metric is a weighed L_∞ distance, thus we can think those SAMs as special MAMs. Consequently, the curse of dimensionality can be considered as a special case of the problem of high intrinsic dimensionality.

In case of vector datasets, the intrinsic dimensionality negatively depends on the correlations among coordinates of vectors in the dataset. The intrinsic dimensionality ρ can reach up to the value of the classic dimensionality D, i.e. $\rho \approx D$, which is the case of uniformly distributed vector datasets.

The curse of dimensionality is apparent even in real-world datasets, where the (groups of) coordinates are correlated. However, in such cases the intrinsic dimensionality is lower, it is affected by the number of independent coordinates.

Example 6.2

Let us show how the classic dimensionality D influences the distance distribution and, hence, the intrinsic dimensionality. For simplicity, we choose L_1 metric and, furthermore, we consider a dataset of vectors where the coordinate values are all independent and uniformly distributed over interval $\langle 0, k\rangle$ of integers.

The distribution of i-th coordinate of a vector O_1 can be represented by a random variable $\mathbf{u_i}$. Similarly, a random variable $\mathbf{v_i}$ can represent the distribution of i-th coordinate of a vector O_2. Then L_1 distance between two randomly chosen vectors O_1 and O_2 is defined as $L_1(O_1, O_2) = \sum_{i=1}^{D} |\mathbf{u_i} - \mathbf{v_i}|$.

For $D = 1$ the distance distribution of $L_1(O_1, O_2)$ is not uniform. The probability of zero distance is the same as probability of the mean distance. Otherwise, the greater distance, the lower probability (the less occurrences in distance distribution histogram). Specifically, the probability that $|\mathbf{u_1} - \mathbf{v_1}| = 1$ is higher than probability that $|\mathbf{u_1} - \mathbf{v_1}| = 2$ which is higher than probability that $|\mathbf{u_1} - \mathbf{v_1}| = 3$ and so on. Thus, for $D = 1$ the most probable distance is 1, while the least probable distance is k.

For $D > 1$ the distance $L_1(O_1, O_2)$ is a sum of D 1-dimensional distances $|\mathbf{u_i} - \mathbf{v_i}|$. The effect of adding the 1-dimensional distances is similar to creating an average 1-dimensional distance, having D attempts for sampling the most "representative" 1-dimensional distance. The higher dimensionality D, the more attempts and the stronger influence of distance averaging reflected by a more

[1]As high-dimensional we usually consider datasets with dimensionality $D \geq 20$ [18].

right-shifted and more narrow bell in the distance distribution histogram. In Figure 6.1a see L_1-DDH for a 2D dataset, in Figure 6.1b see L_1-DDH for a 30D dataset. Finally, for $D \to \infty$ the distance $L_1(O_1, O_2)$ between two randomly chosen vectors O_1, O_2 statistically tends to a single value (but infinite).

6.2.2 Intrinsic Dimensionality and MAMs

In order to demonstrate the effects of high intrinsic dimensionality on exact MAMs, we have to consider the following properties of high-dimensional datasets[2]:

1. All objects in the dataset are almost equally distant from each other, i.e. only such triplets (O_i, O_j, O_k) occur, that $d(O_i, O_j) \approx d(O_j, O_k) \approx d(O_i, O_k)$.

2. Distance of Q's nearest neighbour is similar to distance of its farthest neighbour.

3. Query radius of any reasonable range query (having selectivity at least one object except Q) is very large, it is close to query radius covering all the objects.

Considering the above mentioned observations, we can demonstrate the behaviour of LAESA and M-tree, when a high-dimensional dataset has to be searched.

LAESA

For a high-dimensional dataset, the distance distribution histogram related to an arbitrary pivot P_t (i.e. considering $d(P_t, O_i), \forall O_i \in \mathbb{S}$) is similar to the overall distance distribution histogram (i.e. considering $d(O_i, O_j), \forall O_i, O_j \in \mathbb{S}$).

Given a pivot P_t, all the objects in $\mathbb{S}$ are located inside a "virtual" hyper-ring region $(P_t, min(d(P_t, O_i)), max(d(P_t, O_i)))$, where $min(d(P_t, O_i))$ is distance to the nearest neighbour of P_t, and $max(d(P_t, O_i))$ is distance to the farthest neighbour of P_t. Due to the above mentioned observations, the hyper-ring region is very thin, because $min(d(P_t, O_i)) \approx max(d(P_t, O_i))$. Since the distribution of query objects is assumed to be equal to the distribution of data objects, an arbitrary query object Q will be located inside the hyper-ring region as well.

However, for any reasonable range query (Q, r_Q) the query radius r_Q is large, $r_Q = min(d(Q, O_i)) \approx min(d(P_t, O_i))$. On the other side, the width of hyper-ring $max(d(P_t, O_i)) - min(d(P_t, O_i))$ is far smaller than r_Q.

As a consequence, the interval $\langle min(d(P_t, O_i)), max(d(P_t, O_i))\rangle$ is entirely nested inside the interval $\langle d(P_t, Q) - r_Q, d(P_t, Q) + r_Q\rangle$, thus no elimination of objects can be performed by LAESA (even when many pivots are used), and all the objects must be directly compared against Q, see Figure 6.2a.

[2]In the following we think "high-dimensional" as "intrinsically high-dimensional".

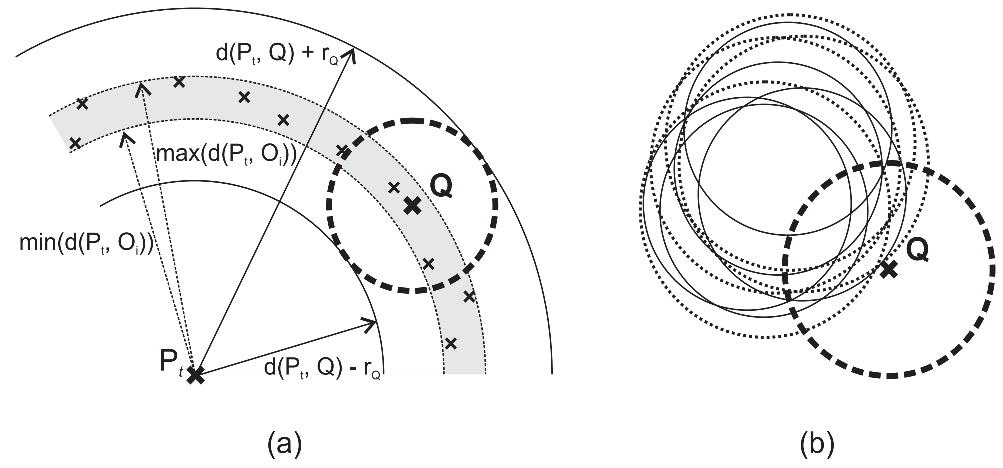


Figure 6.2 Effects of high intrinsic dimensionality (a) LAESA (b) M-tree

M-tree

In case of M-tree, the indexing of a high-dimensional dataset will lead to a hierarchy of nodes, where almost all the metric regions are entirely overlapped. Obviously, in order to process a range query in such an M-tree, all nodes have to be accessed (see Figure 6.2b).

6.3 Related Work

Nowadays, an efficient search in high-dimensional datasets is feasible solely by usage of approximate methods. The approximate metric search is realized by either *approximately correct methods* or *probabilistic methods.* Among many approaches to the approximate search, in this section we outline several representative methods, that were applied to M-tree, LAESA and sequential search.

6.3.1 Approximately Correct Search

In [113] the authors have proposed three techniques of approximate search in M-tree, differentiating in the way how to give up the query result precision for the sake of improved search efficiency. The techniques were applied to k-NN search.

Relative Distance Error

The first technique[3] of approximate k-NN search considers a user-defined *relative distance error* $\epsilon \geq 0$. The error ϵ states that the distance between Q and an approximation of k-th nearest neighbor O_A^k must be no more than $(1+\epsilon)$ times

[3]A similar approach, but oriented to BBD-trees, was proposed in [5].

farther, than the real k-th nearest neighbour O_N^k, i.e. $d(Q, O_A^k) \leq (1+\epsilon)d(Q, O_N^k)$. Then O_A^k is called the $(1+\epsilon)$ k-th nearest neighbour.

In order to provide approximate search, the k-NN query algorithm of M-tree (described in Section 3.2.2) is adjusted in a way that query radius r'_Q, used for discarding irrelevant regions from PR queue, is $1+\epsilon$ times smaller than the proper dynamic query radius r_Q (i.e. $r'_Q = r_Q/(1+\epsilon)$). Due to the faster reduction of the dynamic query radius r'_Q, the entries of PR queue are discarded more frequently, which leads to an earlier termination of the whole k-NN query processing, as is shown in the following example.

Example 6.3

In Figure 6.3 see a situation of 1-NN query processing, right after the node B was processed, and its entry removed from the PR queue. The new NN candidate is now O_4. Although the dynamic query region (Q, r_Q) overlaps the region of node A (containing the real nearest neighbour O_2), the $(1+\epsilon)$-reduced "discarding region" (Q, r'_Q) does not overlap A, thus A's entry is removed from PR queue. In the next step, the node C is processed as the last relevant entry remaining in the PR queue.

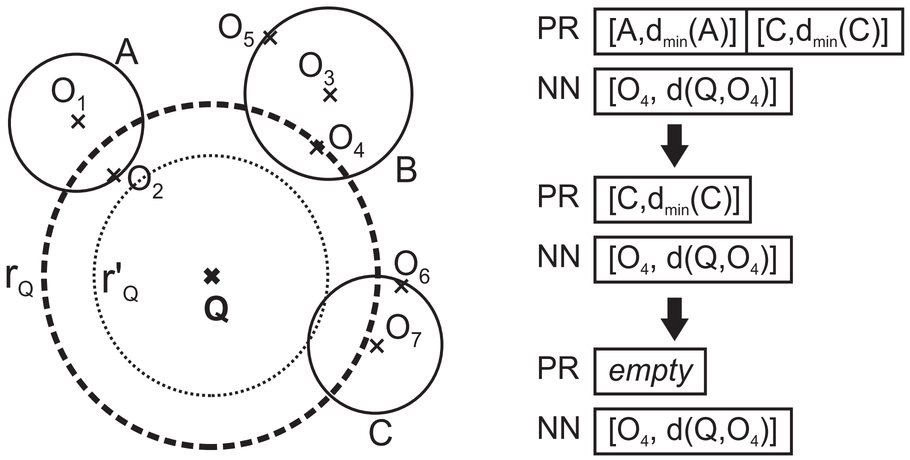


Figure 6.3 Approximate k-NN search in M-tree using relative dist. error

Distance Distribution

The second technique proposed in [113] is a usage of distance distribution $F_Q(x)$ (as described in Section 5.6) for approximate k-NN search. More specifically, $F_Q(x)$ represents the fraction of objects in the dataset, for which the distance to Q is less than or equal to x. Provided there are n objects in the dataset, $n \cdot F_Q(x)$ objects should have distance to Q not greater than x.

Given a k-NN candidate O_A^k, $F_Q(d(Q, O_A^k))$ determines the fraction of the best approximations of k-NN. In other words, there are estimated $n \cdot F_Q(d(Q, O_A^k))$

k-th nearest neighbour candidates that are better (closer) than O_A^k. Such a property can be easily exploited for the approximate k-NN search. Given a user-defined threshold $\rho \in \langle 0, 1\rangle$, we retrieve k objects among those belonging to the fraction ρ of the best k-NN approximations. The stop condition is defined as $F_Q(d(Q, O_A^k)) \leq \rho$, and used in the k-NN algorithm to terminate the search, possibly before the real nearest neighbours are found.

Slowdown of Distance Improvements

The third approximation heuristic proposed in [113] stops the k-NN algorithm processing as soon as the intermediate results in NN array change only slowly. In such cases, most of the k-NN algorithm runtime is spent on negligible changes in NN array, where $d(Q, O_A^k) = \mathrm{NN}[k].d_{max}$ is changing even slower.

Assume a strictly decreasing continuous function $f : \mathbb{R}_0^+ \mapsto \{d(Q, O_A^k)\}$ representing, in a time $t \in \mathbb{R}_0^+$, the distance between Q and the current k-NN candidate O_A^k. The continuous time is approximated by the number of distance computations, already performed during the k-NN algorithm running. Then, the derivative $f'(t)$ represents the chance for finding a better k-NN candidate in a given time t.

Finally, given a user-defined constraint $\kappa > 0$, we stop the k-NN algorithm processing as soon as $f'(t) \leq \kappa$.

6.3.2 Probabilistic Search

Given a user-defined threshold $\delta \in \langle 0, 1\rangle$, the probabilistic methods search such that probability of a false drop in query result is at most δ. The user can tune the δ parameter in order to achieve an optimal efficiency/accuracy trade-off.

Probabilistic LAESA

A probabilistic approach to LAESA-like methods has been proposed in [32], ”stretching” the triangular inequality by a multiplier $\beta \geq 1$. Specifically, the filtering condition $|d(O_i, P_t) - d(Q, P_t)| > r_Q$ is stretched by β as:

$$\beta|d(O_i, P_t) - d(Q, P_t)| > r_Q \quad \text{, i.e.,} \quad |d(O_i, P_t) - d(Q, P_t)| > \frac{r_Q}{\beta}$$

The effect of stretching is visible in Figure 6.4, where for higher β the interval $\langle d(P_t, Q) - r_Q/\beta, d(P_t, Q) + r_Q/\beta\rangle$ gets smaller, thus a probability of discarding an irrelevant object becomes higher. Unfortunately, the probability of a false drop becomes higher as well.

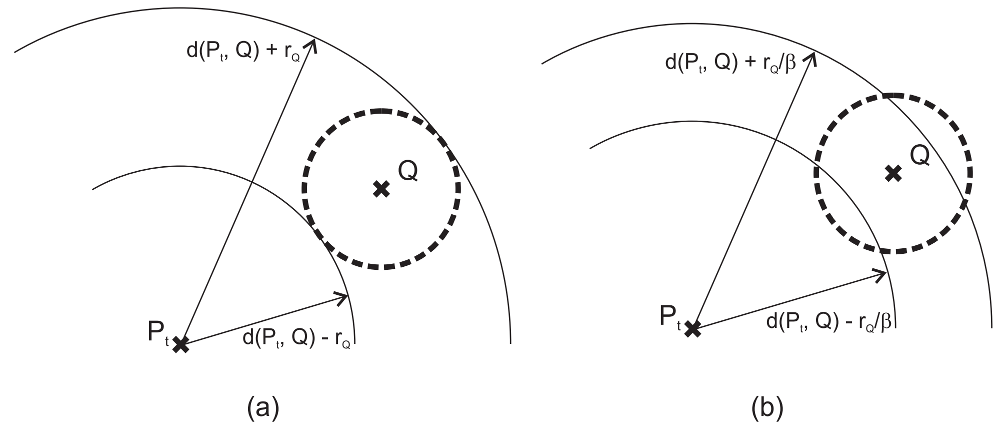


Figure 6.4 (a) Exact LAESA-like filtering, i.e. $\beta = 1$
(b) Approximate LAESA-like filtering, i.e. $\beta > 1$

In order to preserve a user-defined probability δ of a false drop, the upper-bound for β is computed as (for details see [32])

$$\beta \leq \frac{r_Q\sqrt{1-(1-\delta)^{\frac{1}{p}}}}{\sqrt{2}\sigma}$$

where p is the number of pivots, and σ^2 is variance of the dataset's DDH.

In general, this approach is similar to the first technique for M-tree, presented in Section 6.3.1 (replacing $1+\epsilon$ by β), but here the search accuracy β is upper-bounded by a probability δ of a false drop.

PAC Queries

When searching in a high-dimensional dataset, even the approximately correct methods often search inefficiently. In [35] the authors propose probably approximately correct (PAC) nearest neighbour search, even more reducing the search costs at the expense of only probabilistic search. The method extends the $(1+\epsilon)$ nearest neighbour search (presented in Section 6.3.1) by a user-defined confidence parameter δ. The method searches so that the retrieved object is a $(1+\epsilon)$ nearest neighbour with probability at least δ.

Given a distance distribution $F_Q(x)$, the distance r_Q to the nearest neighbour O_A is estimated as δ-radius of Q, denoted r_Q^δ, representing the maximum value of distance from Q, for which the probability that there exists at least one object O_i where $d(Q, O_i) \leq r_Q^\delta$ is not greater than δ. In other words, the δ-radius r_Q^δ guarantees that query region (Q, r_Q^δ) is empty with probability at least δ. The δ-radius is constructed as $r_Q^\delta = G_Q^{-1}(\delta)$, where $G_Q(x)$ is distance distribution of the nearest neighbour, given by $G_Q(x) = \mathbf{Pr}\{r_Q \leq x\} = 1-(1-F_Q(x))^n$, where n is size of the dataset.

The NN algorithm is enhanced by a stop condition, which terminates the NN search if the distance to the current $(1+\epsilon)$ NN candidate falls below r_Q^δ. The PAC-NN algorithm has been applied to M-tree as well as to sequential file index.

Recent Work

Recently, an approach [2] has been proposed, predicting a probability that intersection of a query region and a data region contains some indexed objects. If the probability is lower than a user-defined threshold, the data region (e.g. M-tree node) is not processed. The probability is predicted by means of *region proximity*, computed using the dataset's distance distribution.

Very recently, a method of probabilistic metric search based on *compact partitions* has been introduced in [24]. The idea is to fix in advance the limit of distance computations allowed to answer a query. Moreover, an advanced version exploits a ranking of the regions to be searched, so that the most promising regions are processed at first.

Chapter 7

Semi-metric Search

Recently, we have proposed [95, 97] an approach to approximate search, based on a straightforward reduction of the dataset's intrinsic dimensionality. The reduction is achieved by utilizing so-called semi-metric modifications of the original metric, due to which the variance of distance distribution histogram is increased and/or the mean is decreased.

7.1 Semi-metric Modifications

As a key tool, we define increasing modification d^f of an original metric d, preserving the similarity ordering among objects as follows.

Definition 7.1 (increasing modification)
Given a metric d, we call a function d^f

$$d^f(O_i, O_j) = f(d(O_i, O_j))$$

the *increasing modification of* d, where $f : \langle 0, d^+\rangle \rightarrow R_0^+$, called *modifying function*, is a strictly increasing function for which $f(0) = 0$. $\square$

Definition 7.2 (similarity ordering)
Let $s : \mathbb{U} \times \mathbb{U} \rightarrow R_0^+$ be a similarity function (or a distance function). We define a function $SimOrder_s : \mathbb{U} \rightarrow \mathcal{P}(\mathbb{S} \times \mathbb{S})$ as

$$\langle O_i, O_j\rangle \in SimOrder_s(Q) \Leftrightarrow s(O_i, Q) < s(O_j, Q)$$

$\forall O_i, O_j \in \mathbb{S}, \forall Q \in \mathbb{U}$. In other words, the function $SimOrder_s$ orders the objects of $\mathbb{S}$ according to the distances to a query object Q. $\square$

Lemma 7.1
For a metric d and an increasing modification d^f, the following equality holds:

$$SimOrder_d(Q) = SimOrder_{d^f}(Q), \forall Q \in \mathbb{U}$$

Proof: "$\subset$": The function f is strictly increasing. If for each $O_i, O_j, O_k, O_l \in \mathbb{U}$, $d(O_i, O_j) > d(O_k, O_l)$ holds then $f(d(O_i, O_j)) > f(d(O_k, O_l))$ must hold as well. "$\supset$": The second part of proof is similar. ■

As a consequence of Lemma 7.1, if a query is processed sequentially (i.e. by processing of all objects of the dataset $\mathbb{S}$), then it does not matter if we use either d or any d^f, because both of the ways will return the same query result.

Lemma 7.2

Let f be a modifying function. Then

(a) If f is *subadditive* (i.e. $\forall x, y \in \mathbb{R}_0^+, f(x) + f(y) \geq f(x+y)$), then f is *metric-preserving function* [39], i.e. an increasing modification d^f is still metric, considering *any* metric d.

(b) A strictly concave function f is metric-preserving.

(c) A convex (even partially convex) function f is not metric-preserving, we call such function as *metric-violating*.

Proof: We refer to [39]. ■

A modification d^f, where f is a metric-violating function, is generally a *semi-metric*, i.e. a function satisfying all the metric axioms except the triangular inequality. In such case, we call d^f a *semi-metric modification of d*.

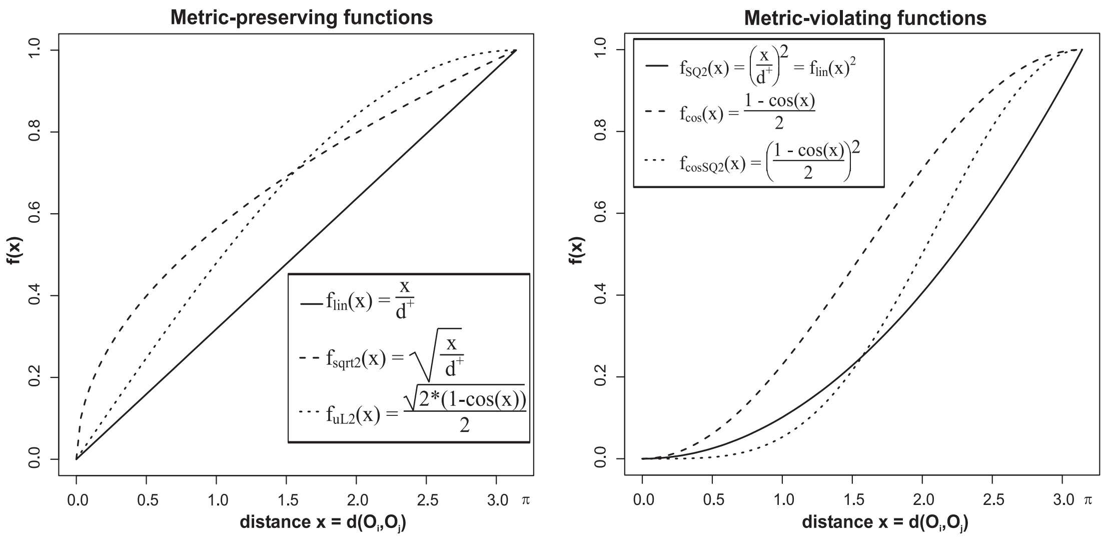


Figure 7.1 (a) Metric-preserving functions (b) Metric-violating functions

Example 7.1

Let the maximal distance be $d^+ = \pi$ and $Im(f) = \langle 0, 1 \rangle$. In Figure 7.1a see several concave (i.e. metric-preserving) functions, while in Figure 7.1b see (partially) convex (i.e. metric-violating) functions.

7.1.1 Clustering Properties

Let us discuss the clustering properties of modifications d^f (see also Figure 7.1). Considering concave functions f, two objects $O_i, O_j \in \mathbb{U}$ close to each other according to $d(O_i, O_j)$ will be more distant according to $d^f(O_i, O_j)$. Conversely, for convex f the close objects according to d will be even closer according to d^f.

As a consequence, the concave modifications d^f have a negative influence on clustering, since the object clusters become indistinct. On the other side, the convex modifications d^f even more tighten the object clusters, making the cluster structure of the dataset more evident. From another point of view, the convex modifications increase the DDH's variance (and/or decrease the mean), decreasing the intrinsic dimensionality. Obviously, the semi-metric modifications violate the original topology of d, we will discuss the violating properties in the following section.

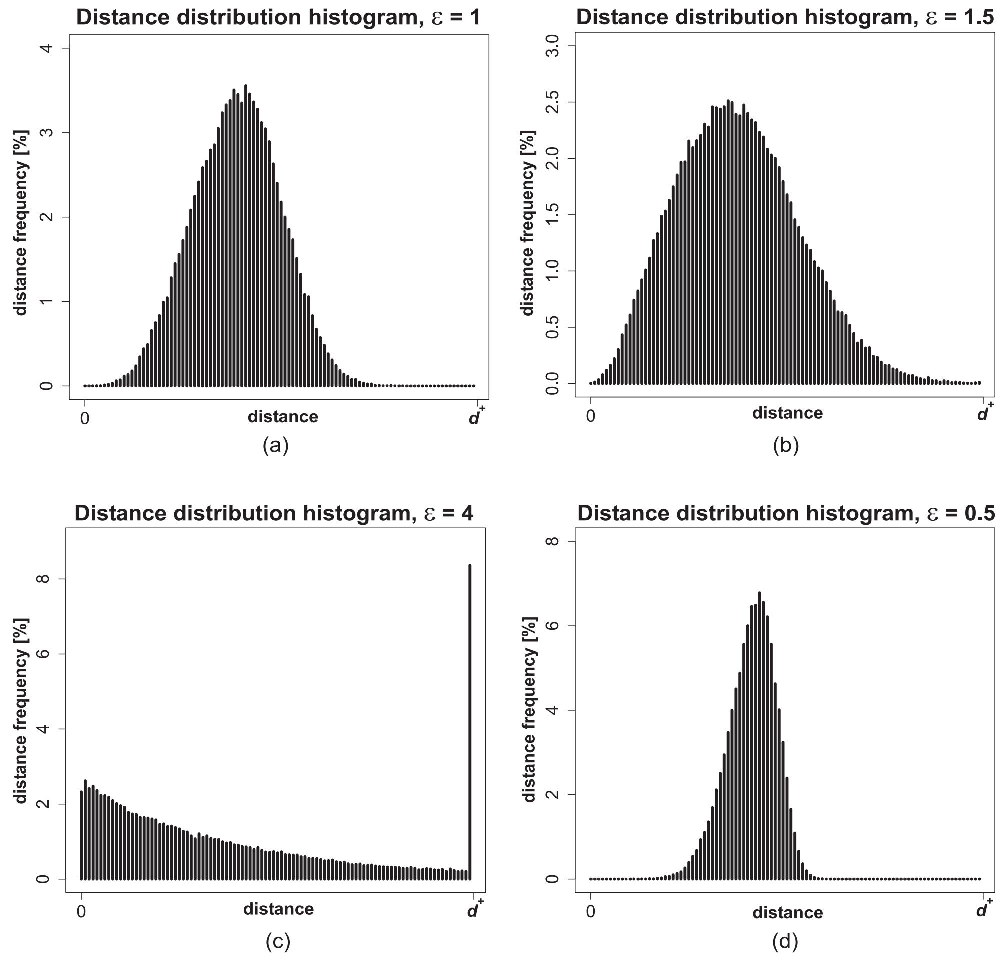


Figure 7.2 DDHs for 5-dimensional dataset, $d = L_2, f(x) = x^\epsilon$
(a) original metric d (d^f, $\epsilon = 1$) (b) semi-metric d^f, $\epsilon = \frac{3}{2}$
(c) semi-metric d^f, $\epsilon = 4$ (d) metric d^f, $\epsilon = \frac{1}{2}$

Example 7.2

Consider a 5-dimensional uniformly distributed dataset embedded inside unitary hyper-cube. The dataset's L_2-DDH is presented in Figure 7.2a. The Figures 7.2b and 7.2c[1] show the dataset's d^f-DDHs, where d^f is semi-metric modification, $d = L_2$, $f(x) = x^{\frac{3}{2}}$, $f(x) = x^4$. In Figure 7.2d a DDH for metric modification d^f, $f(x) = \sqrt{x}$ is shown.

Note that for a more convex f the bell in DDH is more left-shifted (the decreased mean) and also more wide (the increased variance). Conversely, for a concave f (i.e. metric modification d^f) the bell in DDH is right-shifted and more narrow.

As a consequence of the above mentioned clustering properties, the intrinsic dimensionality is lower considering d^f-DDHs where $f(x) = x^{\frac{3}{2}}$, $f(x) = x^4$ and higher for $f(x) = \sqrt{x}$.

7.1.2 Analysis of Metric-violating Functions

Although all convex functions are metric-violating, the functions can be differentiated by their convexity. Generally, due to the convexity of modifying function f, the distance distortion of d^f is more apparent for close objects than for distant objects. In other words, the original topology is more violated for the close objects, which are even closer, but it is less violated for the distant objects.

Intuitively, the more convex a function f is, the more could one expect a modification d^f will violate the triangular inequality. In order to quantify "how much" a particular modification d^f violates the triangular inequality, we inspect which triangular triplets are preserved after an application of f, and which are not.

Definition 7.3 (triangular triplet)

A triplet (a, b, c), where $a, b, c \in \langle 0, d^+ \rangle$ are arbitrarily chosen distances, is called *triangular triplet* iff $a + b \geq c \wedge b + c \geq a \wedge a + c \geq b$. $\Box$

Given a metric d, any triplet $(d(O_i, O_j), d(O_j, O_k), d(O_i, O_k)), O_i, O_j, O_k \in \mathbb{U}$ is triangular triplet (which follows from the triangular inequality property). Furthermore, given a metric-violating function f, some of the triplets are not preserved by d^f, i.e. some triplets $(f(d(O_i, O_j)), f(d(O_j, O_k)), f(d(O_i, O_k)))$ are not triangular triplets. In the following definition we quantify the proportion of triangular triplets preserved by d^f.

Definition 7.4 (triangle-violation error δ^f)

Let f be a strictly convex modifying function. Let $\Omega \subset \langle 0, d^+ \rangle^3$ be a region consisting of points representing all possible triangular triplets (a, b, c). Let $\Omega^f \subset \Omega$

[1] In order to make the histogram more legible, the tall column at the right of Figure 7.2c represents sum of frequencies of all remaining distances beyond d^+.

be a subregion consisting of points representing such triangular triplets (a, b, c) for which also $(f(a), f(b), f(c))$ are triangular triplets. Then, for function f we define the *triangle-violation error* δ^f as

$$\delta^f = 1 - \frac{V(\Omega^f)}{V(\Omega)}$$

where $V(\Omega^f)$ is the volume of region Ω^f and $V(\Omega)$ is the volume of region Ω. The volume of Ω can be determined as

$$V(\Omega) = \int_{c=0}^{d^+} 2d^+c - \frac{3}{2}c^2 \ \mathbf{dc} = \frac{d^{+3}}{2}$$

i.e. the volume is one half of the cube $\langle 0, d^+\rangle^3$. Since f is increasing, there must exist an inverse function f^{-1}, and the volume of Ω^f can be determined as

$$V(\Omega^f) = d^{+3} - \int_{c=0}^{d^+}\int_{a=0}^{c} f^{-1}(f(c)-f(a))\mathbf{da}\,\mathbf{dc} - 2\int_{c=0}^{d^+}\int_{a=c}^{d^+} f^{-1}(f(a)-f(c))\mathbf{da}\,\mathbf{dc}$$

□

In other words, a modifying function f preserves (at least) a fraction $1 - \delta^f$ of all possible triangular triplets.

Example 7.3

Given a modifying functions of form $f(x) = x^\epsilon, \epsilon \geq 1$ and $d^+ = 1$, then $V(\Omega) = \frac{1}{2}$, while the volume of Ω^f is

$$V(\Omega^f) = 1 - \int_{c=0}^{1}\int_{a=0}^{c} \sqrt[\epsilon]{c^\epsilon - a^\epsilon}\ \mathbf{da}\,\mathbf{dc} - 2\int_{c=0}^{1}\int_{a=c}^{1} \sqrt[\epsilon]{a^\epsilon - c^\epsilon}\ \mathbf{da}\,\mathbf{dc}$$

Note that Ω^f is really a subregion of Ω, which is achieved by the requirement on strictly convex f (consider $f(x) = x^\epsilon$). In Figures 7.3a–d see 4 c-cuts of Ω and Ω^f for different c and ϵ. In Figure 7.4a see the dependence between δ^f and ϵ.

Suppose a strictly concave function f is used (e.g $f(x) = \sqrt{x}$), then the situation is inverse, Ω is a subregion of Ω^f (see Figure 7.3e). This also means, when a concave modifying function is used, that some triplets (a, b, c) are not triangular, but their modifications $(f(a), f(b), f(c))$ are triangular triplets.

In case that only partially convex function f is used, the two previous situations are combined, i.e. the regions Ω^f and Ω overlap such that $\Omega^f \nsubseteq \Omega$ and $\Omega \nsubseteq \Omega^f$ (see Figure 7.3f). Generally, for partially convex functions the triangle-violation error δ^f is useless, because Ω^f and Ω are not nested regions, making the volume ratio $\frac{V(\Omega^f)}{V(\Omega)}$ insignificant.

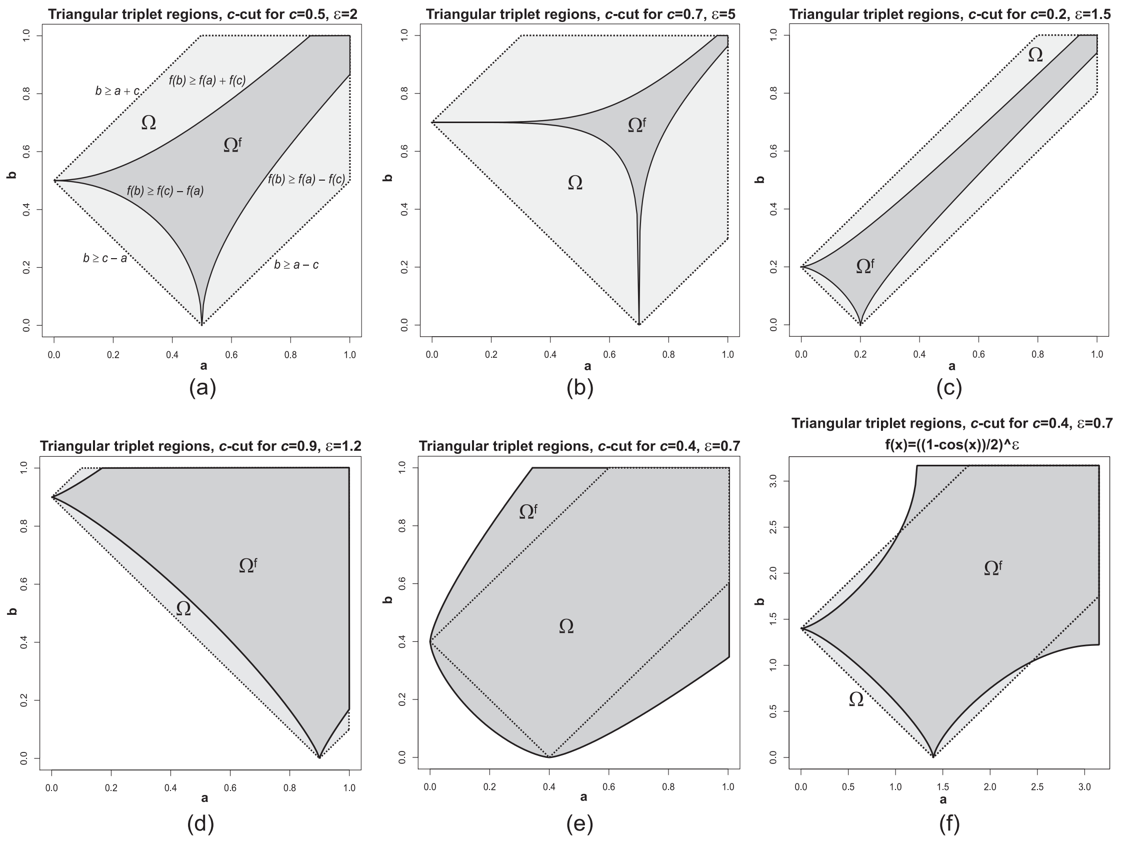


Figure 7.3 Triangular triplet regions Ω and Ω^f, c-cuts for different c and ϵ
(a) – (d) Strictly convex modifying function $f(x) = x^\epsilon, \epsilon \geq 1$
(e) Strictly concave modifying function $f(x) = \sqrt{x}$
(f) Partially convex modifying function $f(x) = \left(\frac{1-cos(x)}{2}\right)^\epsilon, \epsilon = 0.7$

Moreover, the triangle-violation error is meaningful primarily for metrics generating all of the triangular triplets, e.g. the L_2 metric. An example of metric which does not generate e.g. triplets of form $(a, b, a+b), a, b > 0$ is the "normed" L_2 metric, i.e. $NL_2(v_1, v_2) = L_2(\frac{v_1}{||v_1||}, \frac{v_2}{||v_2||})$. Using NL_2 the points are L_2-normed (i.e. projected onto the surface of a unitary L_2-hyper-sphere) and then measured by L_2 metric, see Figure 7.4b.

7.2 Semi-metric Search in M-tree

The increasing modifications d^f can be utilized by MAMs, instead of the original metric d. In order to preserve the selectivity of a range query (Q, r_Q), the user-defined query radius r_Q must be modified to $f(r_Q)$.

We have utilized the semi-metric modifications for approximate search in

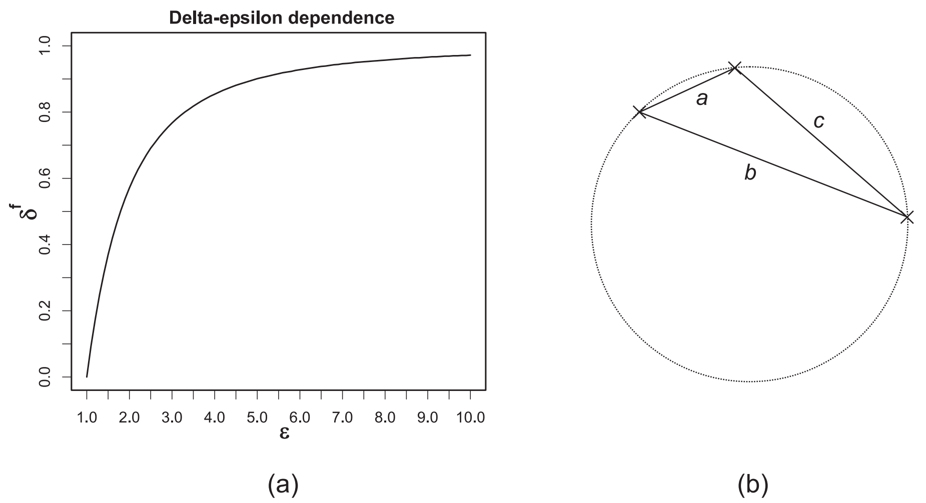


Figure 7.4 (a) Delta-epsilon dependence (b) NL_2 triangular triplet (a, b, c)

M-tree. Since in case of a semi-metric modification d^f the dataset is of lower intrinsic dimensionality, the query processing in M-tree is more efficient because of smaller overlaps among metric regions. On the other side, processing of semi-metric queries introduces a probability of false drops (see the definition of relative output error in Section 7.2.3), making the metric search only approximate.

Unfortunately, a usage of metric modifications for which a false drop cannot occur is not beneficial, because their clustering properties are worse and the overlaps among metric regions in M-tree become larger.

7.2.1 Semi-Metric Indexing vs. Search

Instead of metric d, a semi-metric modification d^f can be used for all operations on the M-tree, i.e. for M-tree building as well as for M-tree querying. Moreover, in the following we show that an M-tree built using a metric d can be queried the same way as it was built by any modification d^f.

Theorem 7.1

Let the dataset $\mathbb{S}$ be dynamically indexed by two M-trees (using the `minMAX_RAD` splitting policy), the first M-tree built using the metric d while the second M-tree built using a semi-metric d^f. Then, both M-trees are of the same hierarchy of metric regions.

Proof: An M-tree hierarchy is dynamically built at two moments. First, for the inserted object a suitable leaf must be found. The single-way leaf choice (see Section 3.3.2) as well as the multi-way leaf choice (see Section 4.2) traverse such path(s) in the M-tree, for which the inserted object is closest to the appropriate routing object. However, since the similarity ordering between the inserted object and the candidate routing objects is preserved (by Lemma 7.1), there is the same leaf chosen, regardless of using either metric d or any semi-metric d^f.

Second, an overfull node must be split such that two new routing objects are sampled and the node content is redistributed. Suppose that `minMAX_RAD` promoting policy is used. Since f is increasing, the minimal radius for d is minimal for d^f as well, thus the same pair of routing objects is chosen. ■

Hence, an M-tree built using a metric d can be queried using any modification d^f, allowing to specify the modifying function just at the query time. Such *semi-metric queries* must be extended by a modifying function f, which is an additional parameter. A semi-metric range query is defined as SM-range(Q,r_Q,f,$\mathbb{S}$) while a k-NN query is defined as SM-kNN(Q,k,f,$\mathbb{S}$). The query algorithms of M-tree have to be modified such that during a semi-metric query processing the function f is applied to each value computed using d as well as it is applied to the metric region radii and precomputed distances stored in the routing entries.

7.2.2 Semi-metric Search Behaviour

The difference between metric search and semi-metric search in M-tree resides in the testing whether a given query region (Q, r_Q) overlaps a metric region (O_i, r_{O_i}) described by a routing entry $rout(O_i)$.

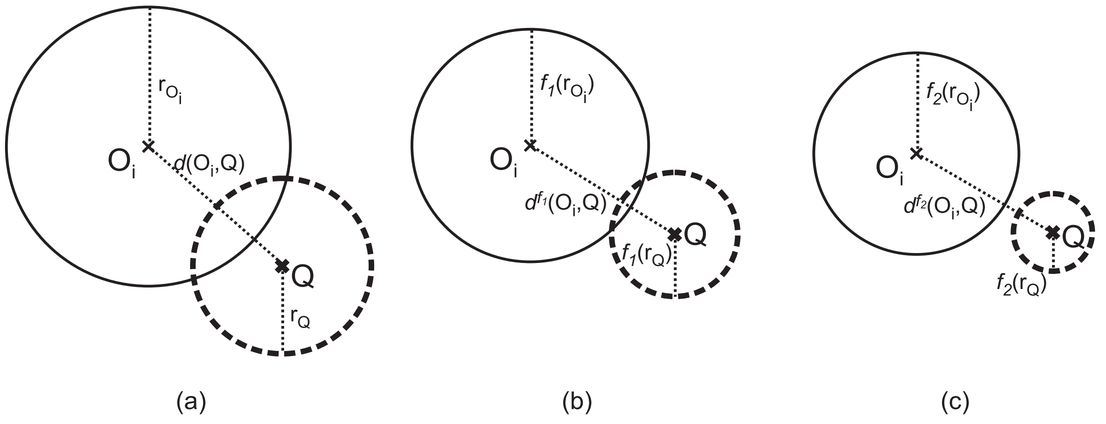


Figure 7.5 (a) d-overlapping regions
(b) d^{f_1}-overlapping regions
(c) d^{f_2}-non-overlapping regions

Given a convex modifying function f, we can distinguish three cases:

1. If the regions are overlapped (using d) such that Q is located inside (O_i, r_{O_i}) or O_i is located inside (Q, r_Q), then the regions are overlapped also when

used any semi-metric modification d^f, i.e. $d(O_i, Q) \leq max(\{r_Q, r_{O_i}\}) \Rightarrow d^f(O_i, Q) \leq f(r_Q) + f(r_{O_i})$.

2. If the regions are not overlapped (using d) then the regions are not overlapped also when used any semi-metric modification d^f, i.e. $d(O_i, Q) > r_Q + r_{O_i} \Rightarrow d^f(O_i, Q) > f(r_Q) + f(r_{O_i})$.

3. The difference between metric and semi-metric search becomes evident in the last case, i.e. when $d(O_i, Q) > max(\{r_Q, r_{O_i}\}) \wedge d(O_i, Q) \leq r_Q + r_{O_i}$ (see Figure 7.5a). The regions do overlap (using d^f) just in case that the triangle-violation error δ^f is small enough. Since $(d(O_i, Q), r_Q, r_{O_i})$ is a triangular triplet, the probability that $(d^f(O_i, Q), f(r_Q), f(r_{O_i}))$ is also triangular triplet grows with decreasing δ^f. In Figure 7.5b see that the regions do overlap when used a modification d^{f_1} having a small δ^{f_1}. In Figure 7.5c the regions do not overlap since the used modification d^{f_2} has too high δ^{f_2}.

Another interpretation of approximate semi-metric search is that, given a semi-metric modification d^f, a region is discarded in case when the volume of its intersection (roughly indicated by $r_{O_i}, r_Q, d(O_i, Q)$) with the query region is too small (relatively to δ^f), i.e. a probability that some indexed objects are located inside the intersection is small as well. Such an interpretation is somewhat similar to the idea of region proximity (cited in the previous chapter), but the semi-metric approach is much simpler, utilizing just the information about region distances and covering radii.

7.2.3 Relative Error of Semi-metric Search

Since a semi-metric does not satisfy the triangular inequality required for exact search in M-tree, the semi-metric search will return more or less approximate results. In order to quantify the relative output error we define a *normed overlap error* as

$$E_{NO} = 1 - \frac{|result_{Mtree} \cap result_{scan}|}{max(|result_{Mtree}|, |result_{scan}|)}$$

where $result_{Mtree}$ is a query result returned by the M-tree (using a semi-metric query), and $result_{scan}$ is a result of the same query returned by sequential search over the entire dataset. The error E_{NO} can be interpreted as a *relative precision* of the M-tree query result with respect to the result of full sequential scan.

In the following chapter we present an experimental evaluation of the semi-metric search in M-tree, as a part of a real-world application – searching in vector model of Text Retrieval.

7.3 Summary

An advantage of semi-metric approach is that reduction of the dataset's intrinsic dimensionality is achieved without an information about the dataset's distance distribution, i.e. no static dataset preprocessing is necessary. Moreover, the modifying function f (and thus the intrinsic dimensionality reduction) can be specified at the query time, which allows to tune the precision/efficiency trade-off. By the way, the modifying function is computationally cheap, thus calculations of f have negligible negative influence on the computation costs. In general, due to the above mentioned properties the method is fully applicable for dynamic MAMs.

7.4 Future Work

In the future we would like to propose a version of semi-metric search utilizing different modifying functions f at different levels of the searched M-tree. As an example, the functions f used at upper levels can have a higher δ^f while the functions used at the leaf level can have a smaller δ^f.

Another important issue for the future is a formulation of an analytical error model for the semi-metric search in M-tree, allowing to predict and control the normed overlap error E_{NO}. The error model might utilize the triangle-violation error δ^f, together with the dataset's distance distribution. However, usage of the analytical error prediction can make the method dependent on a certain dataset preprocessing, and thus the method can lose its dynamicity.

Finally, we want to apply the semi-metric search principles into several other MAMs, e.g. the PM-tree or LAESA.

Chapter 8

Searching in Vector Model of Text Retrieval

As a real application, we have exploited the methods of metric and semi-metric search for efficient query processing in the vector model of Text Retrieval[1].

The models of Text Retrieval [15, 9] provide a formal framework for methods aimed to search in large collections of text documents. The classic vector model as well as its algebraic extension LSI have been proved to be more effective (according to the precision/recall measures) than the other existing models[2]. However, the current methods of vector query processing are inefficient for processing many-term queries and, moreover, in the LSI model they are inefficient for processing any reasonable query.

8.1 Classic Vector Model

In the classic vector model, each document D_j in a collection C ($0 \leq j \leq n$, $n = |C|$) is characterized by a single vector d_j, where each d_j's coordinate is associated with a term t_i from the set of all unique terms in C ($0 \leq i \leq m$, where m is the number of terms). The value of a vector coordinate is a real number $w_{ij} \geq 0$, representing the *weight* of the i-th term in the j-th document. Hence, a collection of documents can be represented by an $m \times n$ *term-by-document* matrix A. There are many ways how to compute the term weights w_{ij} stored in A – a popular weight construction is computed as $tf * idf$ (see e.g. [15]).

Example 8.1

In Table 8.1 see an example of term-by-document matrix A containing five 6-dimensional document vectors (the columns).

[1]Besides the Text Retrieval and Image Retrieval applications, we have also used the M-tree for similarity search in XML databases [58] and face collections [59].

[2]For a comparison over various Text Retrieval models we refer to [86, 41].

document term \	D_1	D_2	D_3	D_4	D_5
database	0	0.48	0.05	0	0.70
vector	0.23	0	0.23	0	0
index	0.43	0	0	0	0
image	0	0	0.10	0	0.54
compression	0	0	0	0	0.21
multimedia	0.12	0.52	0.62	0	0

Table 8.1 Term-by-document matrix A

8.1.1 Sparse Matrix Management

Since a document usually contains only a small subset of all the terms, the matrix A is very sparse. For the purpose of vector model, the favourable sparse matrix storage format is the Compressed Column Storage (CCS)[3] or the Compressed Row Storage (CRS) [15]. Because the matrix storage volume can reach up to the order of gigabytes, there is a need for a secondary memory management. The CCS and CRS formats can be easily implemented on the disk.

CCS Format

In CCS format, the sparse matrix is represented by three arrays: `val`, `row_ind`, and `col_ptr`. The `val` array stores the nonzero elements of the term-by-document matrix A as they are traversed columnwise, i.e. stores all the weights of each document. The `row_ind` array stores the corresponding row indices of the weights in the `val` array, hence, `val[`k`]`$= w_{ij} \Rightarrow$ `row_ind[`k`]`$= i$. The `cols_ptr` array stores the positions in the `val` array that begin a column.

For the purposes of efficient secondary memory management, the `row_ind` and `val` arrays can be stored in a single `RowVal` array, where each element is a pair (r, w) representing a weight w stored on position r in a column (document vector). Each pair (r, w) takes one integer for r and one float for w, say 8 bytes. Suppose the average number of nonzero weights in a document vector is 512, then storage of a sparse document vector takes $512 \cdot 8$ bytes $= 4$ kB in average. If we realize, in such case a random access into the matrix is not much worse in I/O efficiency when compared to the sequential scan, because the storage of document vectors is usually large (say 4 kB), so that a random access to the matrix is always aligned to 4 kB or greater blocks.

In order to achieve an efficient behaviour, the large `RowVal` array is stored on disk while the small `cols_ptr` is loaded in the main memory. In fact, the `cols_ptr` array serves as a lookup table to the matrix columns.

[3]also known as the Harwell-Boeing sparse matrix format [48]

8.1.2 Queries

The most important problem about the vector model is the querying mechanism that searches matrix A with respect to a query Q, and returns only the relevant document vectors (appropriate documents respectively). The query Q is represented by a vector q the same way as a document d_j is represented. The goal is to return the most similar documents to the query. For this purpose a similarity measure must be defined, assessing a similarity value to each pair of query and document vectors (q, d_j). In the context of Text Retrieval, the *cosine measure*

$$\mathrm{SIM}_{cos}(q, d_j) = \frac{\sum_{i=1}^{m} q_i w_{ij}}{\sqrt{\sum_{i=1}^{m} {q_i}^2 \cdot \sum_{i=1}^{m} {w_{ij}}^2}}$$

is widely used. During query processing the columns of A (the document vectors) are compared against the query vector using the cosine measure, while documents sufficiently similar to Q are returned as a result.

Generally, there are two ways how to specify a query Q. First, a *few-term query* is specified by several terms, while an appropriate vector for such a query is very sparse. Second, a *many-term query* is specified using a whole text document, thus appropriate query vector will be more dense. We focus just on the many-term queries, because they better satisfy the similarity search paradigm which the vector model should follow.

8.2 LSI Vector Model

Simply said, the LSI (latent semantic indexing) model [41, 15] is an algebraical extension of the classic vector model. First, the term-by-document matrix A is decomposed by singular value decomposition (SVD) as

$$A = U\Sigma V^T$$

The matrix U contains *concept vectors*, where each concept vector is a linear combination of the original terms. A concept can be interpreted as a kind of *meta-term* appearing in the original documents. While the term-by-document matrix A stores document vectors, the *concept-by-document* matrix ΣV^T stores *pseudo-document* vectors. Each coordinate of a pseudo-document vector represents a weight of an appropriate concept in a document.

8.2.1 Latent Semantics

The concept vectors are ordered with respect to their significance (appropriate singular values in Σ). Consequently, only a small number of concepts is really significant – these concepts represent (statistically) the main themes present in the collection – let us denote the number as k. The remaining concepts are unimportant (noisy concepts) and can be omitted, thus the dimensionality is reduced from m to k. Finally, we get an approximation (rank-k SVD)

$$A \approx U_k \Sigma_k V_k^T$$

where for sufficiently high k the approximation error will be negligible. Moreover, for a low k the effectiveness can be subjectively even higher (according to the precision/recall values) than for a higher k. When searching a real-world collection, the optimal k is usually ranged from several tens to several hundreds. Unlike the term-by-document matrix A, the concept-by-document matrix $\Sigma_k V_k^T$ and the concept base matrix U are dense.

Example 8.2

In Table 8.2 see an example of concept-by-document matrix $\Sigma_4 V_4^T$ ($k = 4$) containing five 4-dimensional pseudo-document vectors (the columns).

document concept \	D$_1$	D$_2$	D$_3$	D$_4$	D$_5$
concept$_1$	-0.21	0.48	-0.05	0.10	0.70
concept$_2$	0.32	0.20	-0.13	0.45	0
concept$_3$	-0.49	0.02	0.77	0.24	-0.06
concept$_4$	0.31	-0.01	0.11	0	0.28

Table 8.2 Concept-by-document matrix $\Sigma_4 V_4^T$

8.2.2 Queries

Querying for documents in the LSI model is performed the same way as in the classic vector model, the difference is that matrix $\Sigma_k V_k^T$ is searched instead of A. Moreover, the query vector q must be projected into the concept base, i.e. $U_k^T q$ is the *pseudo-query vector* used by LSI. Since the concept vectors of U are dense, the pseudo-query vector is dense as well.

8.3 Related Work

In this thesis we focus on efficiency of similarity search. In case of vector model, we can say that a query is processed efficiently if only a small proportion of the matrix storage volume is needed to load and process. In the following we outline several existing approaches to the vector query processing.

8.3.1 Document Vector Scanning

The simplest method how to process a query is the sequential scanning of all the document vectors (i.e. the columns of A, $\Sigma_k V_k^T$ respectively). Each document vector is compared against the query vector using the similarity measure while sufficiently similar documents are returned to the user. It is obvious that for any query the whole matrix must be processed.

8.3.2 Term Vector Filtering

For sparse query vectors (few-term queries respectively), there exists a more efficient scanning method. Instead of document vectors, the term vectors (i.e. the rows of the matrix) are processed. The cosine measure is being computed simultaneously for all the document vectors, "orthogonally" involved in the term vectors. Due to the simultaneous cosine measure evaluation, a set of n accumulators (storing the evolving similarities between each document and the query) must be maintained in memory. The advantage of term filtering is that only those term vectors must be scanned, for which the appropriate term weights in the query vector are nonzero. The term vector filtering can be easily provided using inverted file [67], as a part of the boolean model implementation.

The simple method of term filtering was improved by an approximate approach [81], reducing the time as well as space costs. Generally, the improvement is based on early termination of query processing, exploiting a restructured inverted file where the term entries are sorted according to the decreasing occurrences of a term in document. Thus, the most relevant documents in each term entry are processed first. As soon as the first document is found in which the number of term occurrences is less than a given addition threshold, the processing of term entry can stop because all the remaining documents have the same or less importance as the first rejected document. Since some of the documents are never reached during a query processing, the number of used accumulators can be smaller than n, which saves also the space costs. Another improvement of the inverted file, exploiting *quantized weights*, was proposed recently [3], even more reducing the search costs.

Despite the above mentioned improvements, the term vector filtering is generally not much efficient for many-term queries where the number of filtered term vectors is small. Moreover, the term vector filtering is completely useless for the LSI model because each pseudo-query vector is dense, thus none of the term vectors can be skipped.

8.3.3 Signature Methods

Signature files are a popular filtering method in the boolean model [49], however, there have been only a few attempts made to use them in the vector model, because their usage is not so straightforward due to the term weights. Weight-partitioned signature files (WPSF) [60] try to solve the problem by recording the term weights in so-called TF-groups. Unfortunately, a sequential file organization was chosen for the WPSF which caused excessive search of the signature file. An improvement was proposed recently [68], using the S-trees [43] to speedup the signature file search. Another signature-like approach is the VA-file [17].

Generally, usage of the signature methods is still complicated for the vector model, and the results achieved so far are rather poor.

8.4 Metric Search in the Vector Model

In our approach, we have turned the problem of searching using cosine measure into a problem of searching in a metric space. The objects O_i of metric space are represented by the (pseudo-)document vectors d_i, i.e. by columns of either term-by-document or concept-by-document matrix, respectively. We cannot use directly the cosine measure $\mathrm{SIM}_{cos}(d_i, d_j)$ as a metric function, because it does not satisfy the metric axioms (and e.g. $1 - \mathrm{SIM}_{cos}(d_i, d_j)$ still violates the triangular inequality). As an appropriate metric to the cosine measure, we use the *deviation metric* (also known as *angular distance* [88]), defined as

$$d_{dev}(d_i, d_j) = arccos(\mathrm{SIM}_{cos}(d_i, d_j))$$

which is, instead of cosine, the angle between two vectors d_i, d_j (i.e. $d^+ = \pi$). Informally, the deviation metric can be interpreted as an Euclidean-like distance along the surface of a unitary hyper-sphere.

Note: The function *arccos* in strictly decreasing, thus, by Definition 1.4, the deviation metric d_{dev} is a dissimilarity function complementary to the cosine measure SIM_{cos}.

Consider a similarity range query $\texttt{range}(Q, \alpha, \mathbb{S}, \mathrm{SIM}_{cos})$. In order to obtain an equivalent metric range query $\texttt{range}(Q, r_Q, \mathbb{S}, d_{dev})$, we set the query radius as $r_Q = arccos(\alpha)$. The similarity k-NN query needs not to be explicitly transformed into its metric variant, because the query radius is computed implicitly (dynamically respectively), during the query processing.

Application of M-tree

Since we have turned the similarity queries of vector model into their metric equivalents, we can use the M-tree to index and search in a collection of text documents (in the respective term-by-document or concept-by-document matrices respectively).

8.4.1 Implementation Issues

We have implemented a secondary memory management for both the sparse term-by-document matrix (in CCS format, as presented in Section 8.1.1) as well as for the dense concept-by-document matrix. In case of dense matrix, the implementation was straightforward, because each document vector was stored in a disk page of fixed size, thus random access to the matrix was easily provided.

The entries of M-tree nodes stored just the document vector identifiers (i.e. pointers to the matrix columns), thus the M-tree storage volume was minimized. Due to this fact, the respective matrix was accessed separately for each particular (pseudo-)document vector. Moreover, a document vector was accessed just in case that a distance computation was required to compute, thus the I/O costs were equal to the computation costs.

8.5 Experimental Results

We have used the Los Angeles Times collection (a part of TREC 5 [74]) for the experiments, consisting of 131,780 newspaper articles. The entire collection contained 240,703 unique terms. As ”rich” many-term queries, we have used articles consisting of at least 1000 unique terms. The experiments were focused on I/O costs spent during k-NN queries processing. Each k-NN query was repeated for 100 different query documents and the results were averaged. The I/Os were aligned to 512B blocks, considering both access to the M-tree index as well as to the respective matrix.

Since the deviation metric is computationally cheap (it is of linear complexity according to the dimensionality), in the experiments we have inspected just the I/O costs. The overall query I/O costs are presented either in megabytes or in a proportion of the matrix volume needed to process. In Table 8.3 the M-tree configuration used in the experiments is presented.

Page size:	512 B; Capacity (leaves: 42, nodes: 21)
Construction:	`MinMax` + `SingleWay` + `SlimDown`
Tree height:	3-4; Avg. util. (leaves: 56%, nodes: 52%)

Table 8.3 The M-tree configuration

The labels `DevSQ`xx, `Cos`, `DevSQRT`, and `UL2` in the figures below stand for modifying functions f used by semi-metric search. In particular, several convex (i.e. metric-violating) functions of form $\texttt{DevSQ}p(\alpha) = \left(\frac{\alpha}{\pi}\right)^p$, $\texttt{Cos}(\alpha) = \frac{1-cos(\alpha)}{2}$, and two concave (i.e. metric-preserving) functions $\texttt{DevSQRT}(\alpha) = \sqrt{\frac{\alpha}{\pi}}$, and $\texttt{UL2}(\alpha) = \sqrt{2(1-cos(\alpha))}/2$ were examined as the modifying function f, turning the metric d_{dev} into a semi-metric modification d^f_{dev}. The queries labeled as `Dev` represent the original metric queries (utilizing d_{dev}), as presented in Section 8.4.

8.5.1 Classic Vector Model

First, we performed tests for the classic vector model. The storage of the term-by-document matrix (in CCS format) took 220 MB (consisting of the 131,780 document vectors), while storage of the M-tree index was about 4MB (i.e. 1.8% of the matrix storage volume (MSV)).

For an idea about the intrinsic dimensionality of the collection of document vectors (where the classic dimensionality $D = m = 240,703$ is extremely high), see the distance distribution histograms in Figure 8.1. In the first graph d_{dev}-DDH is presented, where almost 60% of the pairwise distances are approximately equal. With increasing convexity (ϵ parameter) of the modifying function $f(x) = x^\epsilon$, the d_{dev}^f-DDH bell gets wider and left-shifted (see the remaining graphs), thereby decreasing the intrinsic dimensionality. In order to preserve legibility, different scales of the distance axis were chosen for each DDH graph in the figure.

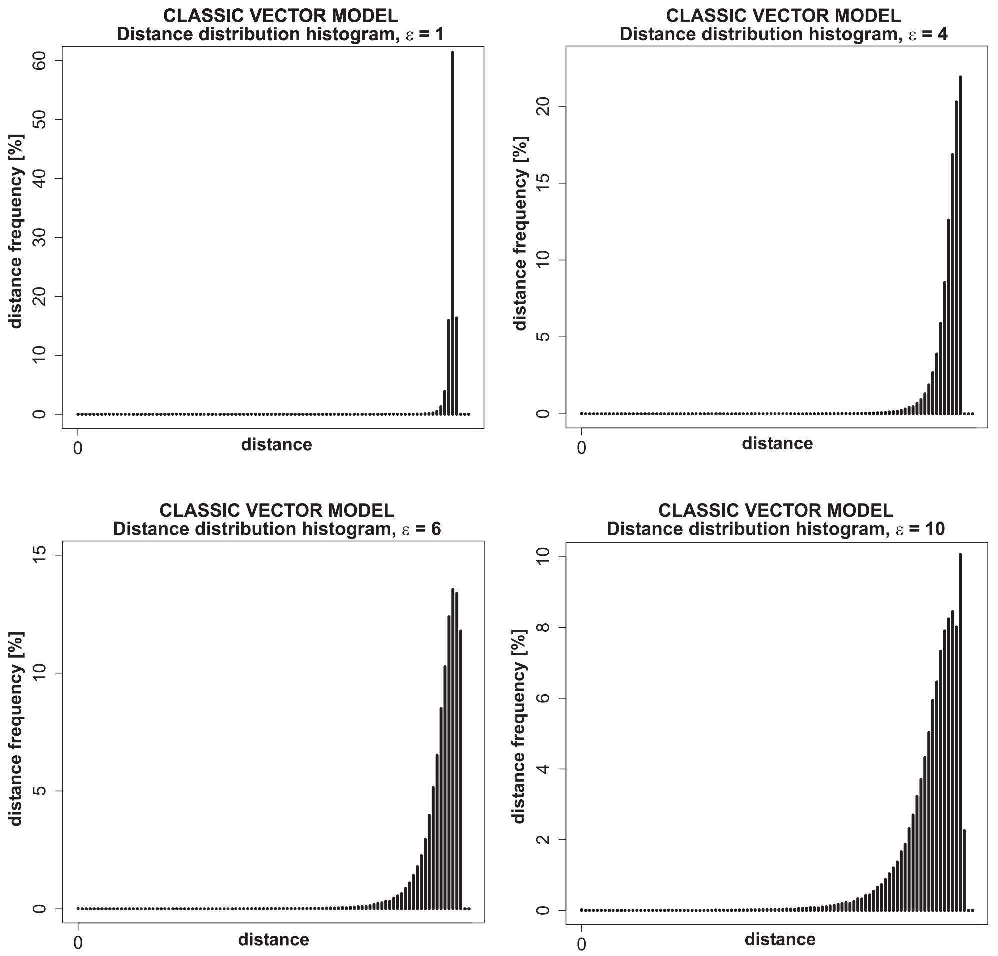


Figure 8.1 Classic vector model: DDHs for d_{dev} and d_{dev}^f

In Figure 8.2a the comparison of document vector scanning, simple term vector filtering as well as metric and semi-metric search is presented. It is obvious that using document vector scanning the whole matrix (i.e. 220 MB I/Os) has to be loaded and processed. Since the query vector contains many zero weights, the term vector filtering works more efficiently (76 MB I/Os, i.e. 34% of MSV).

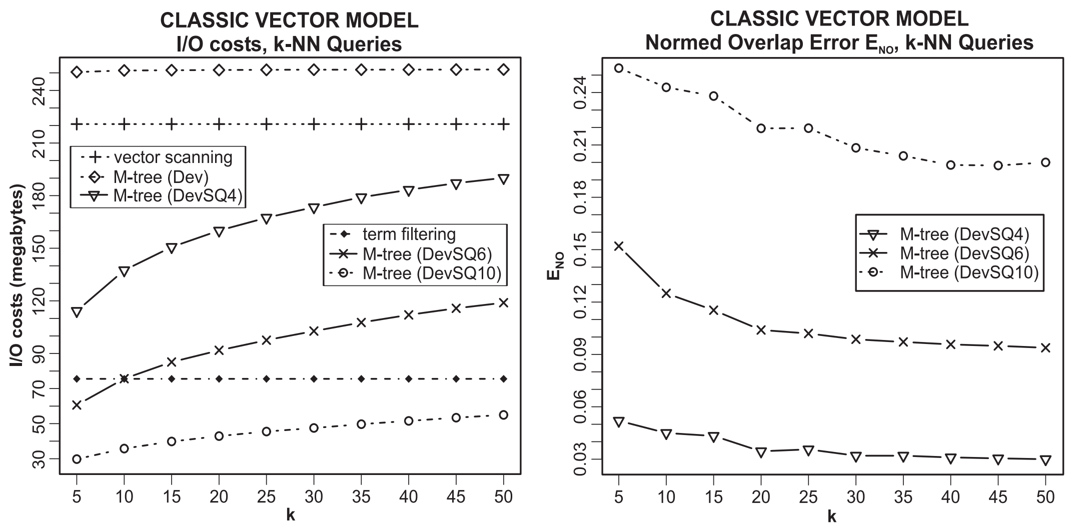


Figure 8.2 Classic vector model: (a) I/O costs (b) E_{NO} error

The metric search `Dev` was not performing well – the curse of dimensionality (m = 240,703) forced almost 100% of the matrix to be processed. The extra 30 MB I/Os overhead (beyond the 220 MB of MSV) was caused by non-sequential access to the matrix columns. On the other side, the semi-metric search performed quite efficiently. The `DevSQ10` queries for $k = 5$ consumed only 30 MB I/Os (i.e. 13.6% of MSV).

Figure 8.2b shows the normed overlap error E_{NO} of the semi-metric search. For `DevSQ4` queries the error was negligible. The error for `DevSQ6` and $k > 35$ remained below 0.1. The `DevSQ10` queries were affected by a relatively high error from 0.25 to 0.2 (with increasing k).

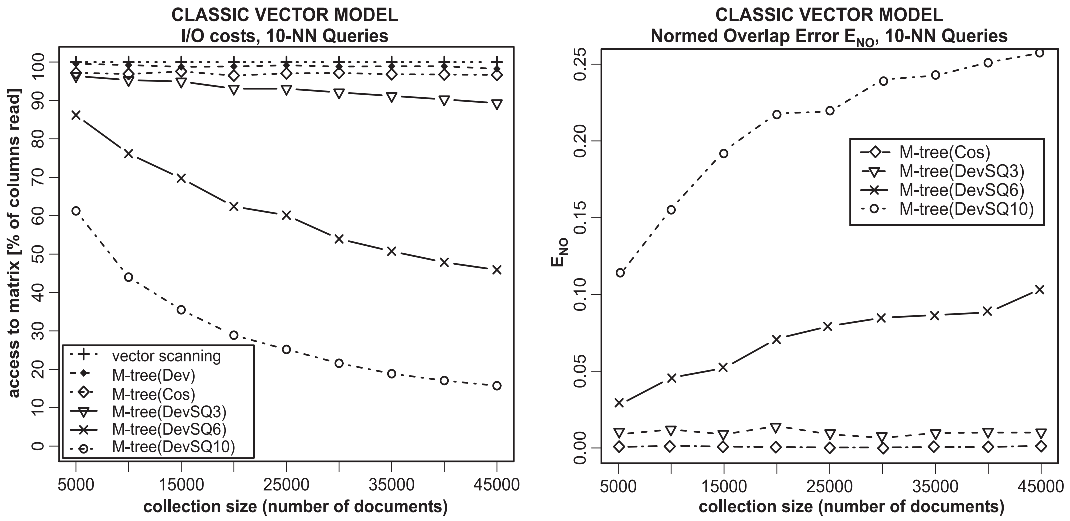


Figure 8.3 Classic vector model: (a) I/O costs (b) E_{NO} error

In Figure 8.3a the I/O costs according to increasing number of documents are presented for 10-NN queries. With the growing size of collection the proportion of MSV needed to process decreased. The error E_{NO} grew quickly for `DevSQ10`, for the remaining functions the error was kept below 0.1, see Figure 8.3b.

8.5.2 LSI Model

The second set of tests was made for the LSI model. The target (reduced) dimension was set to 200. The storage of the concept-by-document matrix took 105 MB, while size of the M-tree index was about 3 MB (i.e. 2.9 % of MSV).

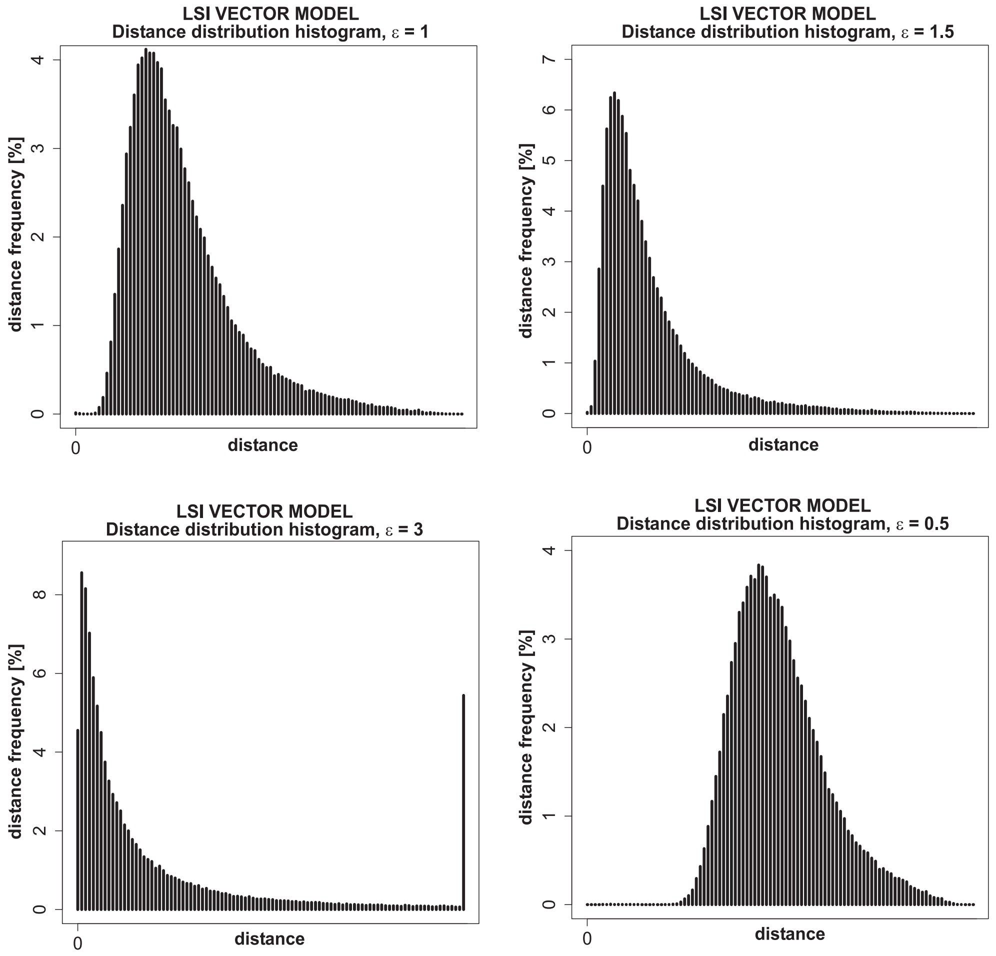


Figure 8.4 LSI model: DDHs for d_{dev} and d^f_{dev}

Because the size of term-by-document matrix was very large, the direct calculation of SVD was technically impossible. Therefore, we have used a two-step method [79], which in the first step calculates a *random projection* of the document vectors into a smaller dimensionality of *pseudo-concepts* (see e.g. [1, 16]). This is done by multiplication of a zero-mean unit-variance random matrix and the term-by-document matrix. Second, a rank-$2k$ SVD is calculated on the resulting *pseudoconcept-by-document* matrix, giving us a very good approximation of the classic rank-k SVD.

In Figure 8.4 see DDHs for the 200-dimensional collection of pseudo-document vectors. As for the classic vector model, the intrinsic dimensionality gets lower[4] with the increasing ϵ. In the fourth graph note that a metric modification (i.e. $f(x) = \sqrt{x}$) shifts the bell right, thereby increasing the mean (and also the intrinsic dimensionality as a whole).

The Figure 8.5a shows that already the metric search `Dev` itself was more than twice as efficient as the document vector scanning. Even better results were achieved by the semi-metric search. The `DevSQ3` queries for $k = 5$ consumed only 5.8 MB I/Os (i.e. 5.5% of MSV). Figure 8.5b shows the error E_{NO}. For `DevSQ1.5` queries the error was negligible, for `DevSQ2` it remained below 0.06. The `DevSQ3` queries were affected by a relatively high error.

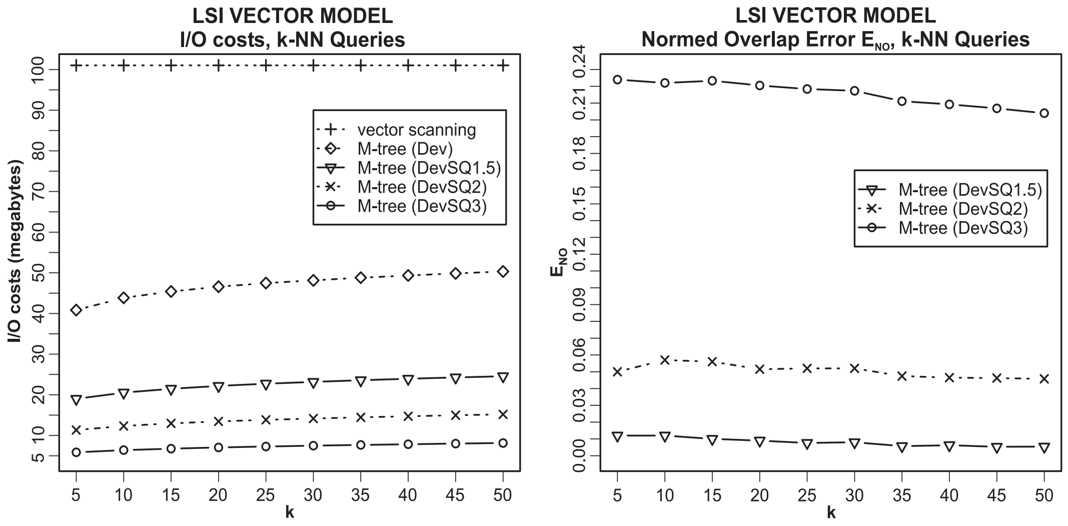


Figure 8.5 LSI model: (a) I/O costs (b) E_{NO} error

In Figure 8.6a the I/O costs according to growing collection size are presented. Besides the convex (i.e. semi-metric) modifications we also examined the concave (i.e. metric) modifications `DevSQRT` and `UL2`. As expected, we can observe that metric modifications performed worse than the original (not modified) metric d_{dev} (`Dev` respectively). In Figure 8.6b we can observe that the error E_{NO} for `Cos` and `DevSQ1.5` was below 0.05.

[4]The comparison of DDHs could be a bit difficult to the reader, since the graphs of DDHs do not share a common distance scale.

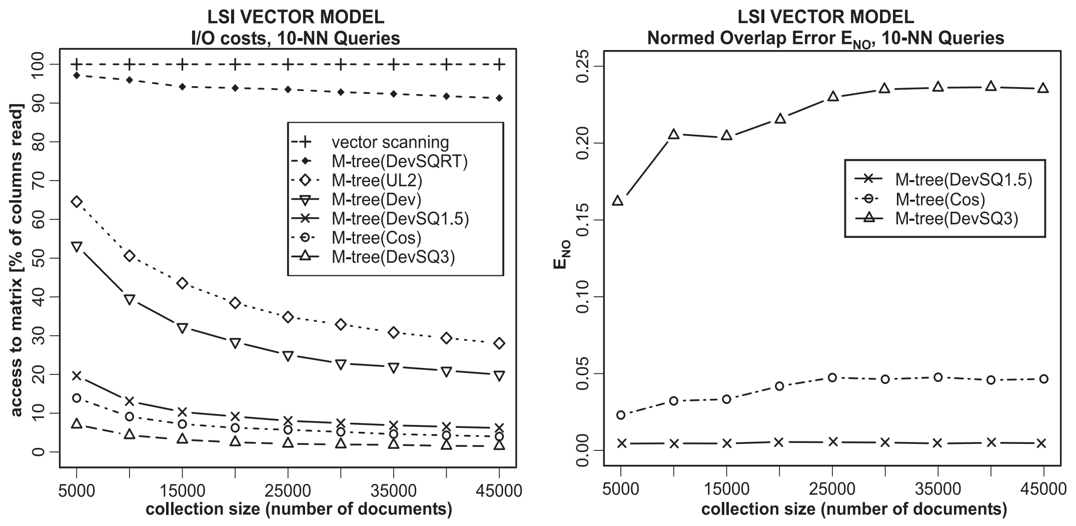


Figure 8.6 LSI model: (a) I/O costs (b) E_{NO} error

8.5.3 Summary

The experimental results have shown that metric indexing itself is suitable for an efficient search in the LSI model. The approximate semi-metric search allows us to provide quite efficient similarity search in the classic vector model. Furthermore, the semi-metric search provides a very efficient solution for search in the LSI model. The relative output error of semi-metric search can be effectively tuned by choosing such modifying functions, that preserve an expected accuracy sufficiently.

Chapter 9

Conclusions

In this thesis, we have presented several contributions to metric indexing in the context of Information Retrieval. We have proposed methods for exact as well as for approximate searching in large metric datasets. All of the contributions have been related to the M-tree, which is a dynamic indexing structure providing similarity search in large metric datasets.

We can summarize the main contributions as follows:

- In the field of exact metric search, we have proposed:
  - two algorithms of building the M-tree, allowing to achieve a higher search efficiency at the expense of higher construction costs
  - the PM-tree, a dynamic indexing structure exploiting – when compared to the M-tree – a more compact shape of metric regions, which is, in turn, reflected by even higher search efficiency together with only a little growth of construction costs
- In the field of approximate metric search, we have proposed a concept of semi-metric modifications that have been utilized for an approximate search using the M-tree. A particular advantage of this approach is a property that no information about the dataset's distance distribution is needed, making the method suitable for dynamic DBMS environments.
- Finally, we have applied the metric indexing approach in order to efficiently handle the problem of similarity search in vector model of Text Retrieval. As the experimental results have shown, the metric and semi-metric search is especially favourable in case of search in the LSI vector model, for which there has not been proposed any efficient method of similarity search yet.

The presented methods have been evaluated by many experiments on synthetic as well as real-world datasets. The new construction algorithms on M-tree achieved an increase of search efficiency by up to 300%. When comparing to the original M-tree, the PM-tree combined with the slim-down algorithm achieved an increase of search efficiency even by an order of magnitude. The approximate semi-metric search can be tuned to achieve a favourable trade-off between the efficiency and the accuracy of searching. In particular, when compared to the sequential scan, searching in a real text collection (in the LSI vector model) achieved 10 times higher efficiency, keeping the relative precision above 95%.

9.1 Outlook

In the future, we want to continue the ideas established in this thesis. Besides the conclusions stated at the end of particular chapters, we head our future efforts towards the following objectives:

- As for the PM-tree, we would like to propose even more compact shapes of metric regions for another M-tree modifications, reducing the probability of a ”region false hit” and thus improving the search efficiency.
- Our next goal is to propose algorithms for processing some other types of similarity queries in (P)M-tree, e.g. the reverse neighbour queries and the similarity joins.
- In the area of approximate search, we plan to formulate a probabilistic framework for the semi-metric search, providing a control of the search accuracy.
- Finally, we intend to compare various metric access methods in a single real-time application environment. More specifically, we want to implement an Image Retrieval System, providing similarity search in large image and video databases.

Bibliography

[1] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 274–281. ACM Press, 2001.

[2] Giuseppe Amato, Fausto Rabitti, Pasquale Savino, and Pavel Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Transactions on Information Systems*, 21(2):192–227, 2003.

[3] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42. ACM Press, 2001.

[4] Peter M. G. Apers, Henk M. Blanken, and Maurice A. Houtsma. *Multimedia Databases in Perspective.* Springer-Verlag New York, Inc., 1997.

[5] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[6] Ricardo A. Baeza-Yates. Searching: an algorithmic tour. *Encyclopedia of Computer Science and Technology*, 37:331–359, 1997.

[7] Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212. Springer-Verlag, 1994.

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

[9] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. String matching with metric trees using an approximate distance. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, pages 271–283. Springer-Verlag, 2002.

[11] Rudolf Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of World-Wide Computing and its Applications'97, WWCA'97, Tsukuba, Japan*, 1997.

[12] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-tree: Breaking the curse of dimensionality. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 142–153. ACM Press, 1998.

[13] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22nd Intern. Conf. on VLDB, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.

[14] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[15] Michael W. Berry and Murray Browne. *Understanding search engines: mathematical modeling and text retrieval*. Society for Industrial and Applied Mathematics, 1999.

[16] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250. ACM Press, 2001.

[17] Stephen Blott and Roger Weber. An Approximation-Based Data Structure for Similarity Search. Technical report, ESPRIT project HERMES (no. 9141), `http://www-dbs.ethz.ch/~weber/paper/TR1997b.ps`, 1999.

[18] Christian Böhm, Stefan Berchtold, and D Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[19] Ingwer Borg and Patrick J. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer-Verlag, 1997.

[20] Tolga Bozkaya and Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 357–368. ACM Press, 1997.

[21] Tolga Bozkaya and Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, 1999.

[22] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584. Morgan Kaufmann Publishers Inc., 1995.

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

[24] Benjamin Bustos and Gonzalo Navarro. Probabilistic proximity search algorithms based on compact partitions. *Journal of Discrete Algorithms*, 2(1):115–134, 2004.

[25] Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[26] Chad Carson, Megan Thomas, Serge Belongie, Joseph M. Hellerstein, and Jitendra Malik. Blobworld: A system for region-based image indexing and retrieval. In *Third International Conference on Visual Information Systems*. Springer, 1999.

[27] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley Professional, 2003.

[28] Edgar Chávez. Optimal discretization for pivot based algorithms. Manuscr. `ftp://garota.fismat.umich.mx/pub/users/elchavez/minimax.ps.gz`, 1999.

[29] Edgar Chávez, José L. Marroquín, and Ricardo Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, page 38. IEEE Computer Society, 1999.

[30] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed Queries Array: A Fast and Economical Data Structure for Proximity Searching. *Multimedia Tools Appl.*, 14(2):113–135, 2001.

[31] Edgar Chávez and Gonzalo Navarro. Measuring the Dimensionality of General Metric Spaces. TR/DCC-00-1. Technical report, Department of Computer Science, University of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/metricmodel.ps.gz`, 2000.

[32] Edgar Chávez and Gonzalo Navarro. A Probabilistic Spell for the Curse of Dimensionality. In *Proc. 3rd Workshop on Algorithm Engineering and Experimentation (ALENEX'01), LNCS 2153*, pages 147–160. Springer-Verlag, 2001.

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

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

[35] Paolo Ciaccia and Marco Patella. PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces. In *Proceedings of the 16th International Conference on Data Engineering*, page 244. IEEE Computer Society, 2000.

[36] Paolo Ciaccia and Marco Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems*, 27(4):398–437, 2002.

[37] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd Athens Intern. Conf. on VLDB*, pages 426–435. Morgan Kaufmann, 1997.

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

[39] Paul Corazza. Introduction to metric-preserving functions. *American Mathematical Monthly*, 104(4):309–23, 1999.

[40] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 667–676. Society for Industrial and Applied Mathematics, 2002.

[41] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[42] Frank Dehne and Hartmut Noltemeier. Voronoi trees and clustering problems. *Syntactic and structural pattern recognition*, pages 185–194, 1988.

[43] Uwe Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 77–87. ACM Press, 1986.

[44] V. Dohnal, C. Gennaro, and P. Zezula. A metric index for approximate text management. In *Proceedings of the IASTED International Conference Information Systems and Databases (ISDB 2002)*, pages 37–42. ACTA Press, March 2002.

[45] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Separable splits of metric data sets. In *9th Italian Conference on Database Systems (SEBD), Venice, Italy*, pages 45–62, 2001.

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

[47] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity Join in Metric Spaces Using eD-Index. In *Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings*, volume 2736 of *Lecture Notes in Computer Science*, pages 484–493. Springer, 2003.

[48] Iain Duff, Roger Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1):1–14, 1989.

[49] Christos Faloutsos. Signature-based text retrieval methods, a survey. *IEEE Computer society Technical Committee on Data Engineering*, 13(1):25–32, 1990.

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

[51] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 4–13. ACM Press, 1994.

[52] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.

[53] Jan Holub. Reduced nondeterministic finite automata for approximate string matching. In *Proceedings of the Prague Stringology Club Workshop*, pages 19–27, 1996.

[54] Jan Holub and Bořivoj Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In *Proceedings of Third International Workshop on Implementing Automata WIA'98, University of Rouen, France, LNCS 1660*, pages 92–99. Springer-Verlag, 1998.

[55] Jing Huang, S. Ravi Kumar, Mandar Mitra, Wei-Jing Zhu, and Ramin Zabih. Image indexing using color correlograms. In *Conference on Computer Vision and Pattern Recognition (CVPR '97)*, 1997.

[56] Iraj Kalantari and Gerard McDonald. A Data Structure and an Algorithm for the Nearest Point Problem. *IEEE Transactions on Software Engineering*, 9(5):631–634, 1983.

[57] Michael Kirby and L. Sirovich. Application of the karhunen-loeve procedure for the characterization of human faces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):103–108, 1990.

[58] Michal Krátký, Jaroslav Pokorný, Tomáš Skopal, and Václav Snášel. The Geometric Framework for Exact and Similarity Querying XML Data. In *Proceedings of the First EurAsian Conference (EurAsia-ICT), LNCS 2510, Shiraz, Iran*, pages 35–46. Springer-Verlag, October 27-31, 2002.

[59] Michal Krátký, Tomáš Skopal, and Václav Snášel. Efficient Searching in Face Collections (in czech). In *Proceedings of conference DATAKON, Brno, Czech Republic*, pages 237–248, 2003.

[60] Dik Kun Lee and Liming Ren. Document ranking on weight-partitioned signature files. *ACM Transactions on Information Systems*, 14(2):109–137, 1996.

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

[62] Rajiv Mehrotra and James E. Gary. Similar-shape retrieval in shape data management. *Computer, IEEE Computer Society Press*, 28(9):57–62, 1995.

[63] Bořivoj Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition, Volume II., Vienna, Austria*, pages 256–260. IEEE Computer Society Press, 1996.

[64] Maria Luisa Micó, José Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recogn. Lett.*, 17(7):731–739, 1996.

[65] Maria Luisa Micó, José Oncina, and Enrique Vidal. An algorithm for finding nearest neighbour in constant average time with a linear space complexity. In *International Conference on Pattern Recognition*, pages 557–560, 1992.

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

[67] Alistair Moffat and Justin Zobel. Fast ranking in limited space. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 428–437. IEEE Computer Society, 1994.

[68] Pavel Moravec, Jaroslav Pokorný, and Václav Snášel. Vector Query with Signature Filtering. In *Proceedings of the 6th Bussiness Information Systems Conference, Colorado Springs, USA*, 2003.

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

[70] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

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

[72] Gonzalo Navarro and Nora Reyes. Fully dynamic spatial approximation trees. In *Proceedings of String Processing and Information Retrieval (SPIRE'02), Lisbon, Portugal*, pages 254–270. Springer-Verlag New York, Inc., 2002.

[73] Andrew Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 2002.

[74] NIST. Text REtrieval Conference (TREC), `http://trec.nist.gov/`.

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

[76] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, pages 186–203. Springer-Verlag, 1992.

[77] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. A data structure for representing and efficient querying large scenes of geometric objects: MB* trees. In *Geometric modelling*, pages 211–226. Springer-Verlag, 1993.

[78] Alice J. O'Toole, Hervé Abdi, Kenneth A. Deffenbacher, and Dominique Valentin. Low-dimensional representation of faces in higher dimensions of the face space. *Journal of the Optical Society of America, series A*, 10(3):405–411, 1993.

[79] Christos H. Papadimitriou, Hisao Tamaki, Prabhakar Raghavan, and Santosh Vempala. Latent semantic indexing: A probabilistic analysis. In *Proocedings of the ACM Conference on Principles of Database Systems (PODS), Seattle*, pages 159–168, 1998.

[80] Marco Patella. *Similarity Search in Multimedia Databases.* PhD thesis, Dipartmento di Elettronica Informatica e Sistemistica, Bologna, `http://www-db.deis.unibo.it/Mtree/index.html`, 1999.

[81] Michael Persin. Document filtering for fast ranking. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 339–348. Springer-Verlag New York, Inc., 1994.

[82] Jaroslav Pokorný. *XML: a challenge for databases?*, chapter 13: Contemporary Trends in Systems Development, pages 147–164. Kluwer Academic Publishers, Boston, 2001.

[83] Pavel Praks, Václav Snášel, and Jiří Dvorský. Latent semantic indexing for image retrieval systems. In *SIAM Linear Algebra, International Linear Algebra Society (ILAS)*, 2003.

[84] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA*, pages 71–79, 1995.

[85] Yong Rui, Thomas S. Huang, and Shih-Fu Chang. Image retrieval: current techniques, promising directions and open issues. *Journal of Visual Communication and Image Representation*, 10(4):39–62, 1999.

[86] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill Publications, Inc., 1st edition, 1983.

[87] Simone Santini and Ramesh Jain. Similarity measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9):871–883, 1999.

[88] Didier Schwab, Mathieu Lafourcade, and Violaine Prince. Antonymy and conceptual vectors. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING), Taipei, Taiwan*, 2002.

[89] Thomas Seidl and Hans-Peter Kriegel. Efficient user-adaptable similarity search in large multimedia databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 506–515, 1997.

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

[91] Dennis Shasha and Kaizhong Zhang. *Pattern Matching Algorithms*, chapter 14: Approximate Tree Pattern Matching, pages 341–371. Oxford University Press, 1997.

[92] L. Sirovich and Michael Kirby. Low-dimensional procedure for the characterization of human faces. *Journal of Optical Society of America*, 4(3):519524, 1987.

[93] Tomáš Skopal. 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`, pages 21–31, 2004.

[94] Tomáš Skopal, Michal Krátký, and Václav Snášel. An Efficient Implementation of the Vector Model for Information Retrieval (in czech). In *Proceedings of conference DATAKON, Brno, Czech Republic*, pages 289–294, 2003.

[95] Tomáš Skopal, Michal Krátký, and Václav Snášel. Metric and Semi-Metric Indexing of Vector Models in Information Retrieval Systems (in czech). In *Proceedings of the 3rd Conference ZNALOSTI 2004, Brno, Czech Republic*, pages 154–165, 2004.

[96] Tomáš Skopal, Pavel Moravec, Michal Krátký, Václav Snášel, and Jaroslav Pokorný. An Efficient Implementation of the Vector Model in Information Retrieval. In *Proceedings of the 5th National Russian Research Conference on Digital Libraries (RCDL), St. Petersburg, Russia*, pages 170–179, 2003.

[97] Tomáš Skopal, Pavel Moravec, Jaroslav Pokorný, and Václav Snášel. Metric Indexing for the Vector Model in Text Retrieval. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE), LNCS 3246, Springer-Verlag, Padova, Italy*, pages 183–195, 2004.

[98] Tomáš Skopal, Jaroslav Pokorný, Michal Krátký, and Václav Snášel. Revisiting M-tree Building Principles. In *Proceedings of the 7th East-European conference on Advances in Databases and Informations Systems (ADBIS), LNCS 2798, Springer-Verlag, Dresden, Germany*, pages 148–162, 2003.

[99] Tomáš Skopal, Jaroslav Pokorný, and Václav Snášel. Nearest Neighbours Search in Multimedia Databases using the PM-tree. In *submitted, 12 pp.*, 2004.

[100] Tomáš Skopal, Jaroslav Pokorný, and Václav Snášel. PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases. In *Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS), Research Communications, Budapest, Hungary*, pages 99–114, 2004.

[101] Daniela Stan and Ishwar K. Sethi. Color patterns for pictorial content description. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 693–698. ACM Press, 2002.

[102] H. Tamura, S. Mori, and T. Yamawaki. Textual features corresponding to visual perception. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(6), 1978.

[103] Caetano Traina Jr., Agma Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777, 2000.

[104] Amos Tversky. Features of similarity. *Psychological review*, 84(4):327–352, 1977.

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

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

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

[108] Stephen Watson. The classification of metrics and multivariate statistical analysis. *Topology and Its Applications*, 99(2-3):237–261, 1999.

[109] WBIIS project: Wavelet-based Image Indexing and Searching, Stanford University, `http://wang.ist.psu.edu/`.

[110] Hugh E. Williams and Justin Zobel. Indexing nucleotide databases for fast query evaluation. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, pages 275–288. Springer-Verlag, 1996.

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

[112] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[113] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate Similarity Retrieval with M-Trees. *VLDB Journal*, 7(4):275–293, 1998.

[114] Xiangmin Zhou, Guoren Wang, Jeffrey Yu Xu, and Ge Yu. M^+-tree: A New Dynamical Multidimensional Index for Metric Spaces. In *Proceedings of the Fourteenth Australasian Database Conference - ADC'03, Adelaide, Australia*, 2003.

Appendix A

Author's Selected Publications

In the following we list 27 selected publications related to Information Retrieval and Database Indexing (including 1 accepted paper, 1 submission and 2 technical reports).

T. Skopal, J. Pokorný, V. Snášel
Nearest Neighbours Search using the PM-tree, *submitted*, 12 pp.

T. Skopal, P.Moravec, J. Pokorný, V. Snášel
Metric Indexing for the Vector Model in Text Retrieval, *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 183–195, LNCS 3246, Springer-Verlag, Padova, Italy, 2004

T. Skopal, J. Pokorný, V. Snášel
PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases, *Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Research Communications, pages 99–114, Hungarian Academy of Sciences, Budapest, Hungary, 2004

T. Skopal
Pivoting M-tree: A Metric Access Method for Efficient Similarity Search, *Proceedings of the 4th workshop DATESO*, pages 21–31, VŠB-TUO, Desná-Černá Říčka, Czech Republic, 2004

T. Skopal, M. Krátký, V. Snášel, J. Pokorný
A New Range Query Algorithm for the Universal B-trees, *to appear in Information Systems*, 26 pp., Elsevier Science, 2005

T. Skopal, M. Krátký, V. Snášel
Metric and Semi-Metric Indexing of Vector Models in Information Retrieval Systems (in czech), *Proceedings of the 3rd conference ZNALOSTI*, pages 154–165, VŠB-TUO, Brno, Czech Republic, 2004

M. Krátký, J. Pokorný, T. Skopal, V. Snášel
Implementation of XPath Axes in the Multidimensional Approach to Indexing XML Data (in czech), *Proceedings of the 3rd conference ZNALOSTI*, pages 338–349, VŠB-TUO, Brno, Czech Republic, 2004

M. Krátký, V. Snášel, P. Zezula, J. Pokorný, T. Skopal
Efficient Processing of Narrow Range Queries in the R-Tree, *ARG Technical Report ARG-TR-01-2004*, 11pp., Department of Computer Science, VŠB - Technical University of Ostrava, 2004, `www.cs.vsb.cz/arg/techreports/sigrtree.pdf`

M. Krátký, T. Skopal, V. Snášel
Multidimensional Term Indexing for Efficient Processing of Complex Queries, *Kybernetika*, 40(3):381–396, ISSN 0023-5954, Institute of Information Theory and Automation of the Academy of Sciences of Czech Republic, 2004

T. Skopal, P. Moravec, M. Krátký, V. Snášel, J. Pokorný
An Efficient Implementation of the Vector Model in Information Retrieval, *Proceedings of 5th National Russian Research Conference on Digital Libraries, RCDL*, pages 170–179, St. Petersburg State University Press, St. Petersburg, Russia, 2003

T. Skopal, J. Pokorný, M. Krátký, V. Snášel
Revisiting M-tree Building Principles, *Proceedings of the 7th conference on Advances in Databases and Information Systems (ADBIS)*, pages 148–162, LNCS 2798, Springer-Verlag, Dresden, Germany, 2003

M. Krátký, T. Skopal, V. Snášel
Efficient Searching in Face Collections (in czech), *Proceedings of conference DATAKON*, pages 237–248, Masaryk University, Brno, Czech Republic, 2003

T. Skopal, M. Krátký, V. Snášel
An Efficient Implementation of the Vector Model for Information Retrieval (in czech), *Proc. of conference DATAKON*, pages 289–294, Masaryk University, Brno, Czech Republic, 2003

T. Skopal, M. Krátký, J. Pokorný, V. Snášel
On Range Queries in Universal B-trees, *ARG Technical Report ARG-TR-01-2003*, 13 pp., Department of Computer Science, VŠB-Technical University of Ostrava, 2003, `www.cs.vsb.cz/arg/techreports/range.pdf`

D. Barashev, M. Krátký, T. Skopal
Modern Approaches to Indexing XML Data, *Transactions of the VŠB-Technical University of Ostrava, Computer Science and Mathematics Series*, pages 19–30, VŠB-Technical University of Ostrava, 2003

J. Dvorský, T. Skopal, V. Snášel
Word-based Compression Methods - Survey Report, *Transactions of the VŠB-Technical University of Ostrava, Computer Science and Mathematics Series*, pages 43–52, VŠB-Technical University of Ostrava, 2003

J. Dvorský, M. Krátký, T. Skopal, V. Snášel
Benchmarking the Multidimensional Approach for Term Indexing, *Proceedings of the 3rd workshop DATESO*, pages 70–81, VŠB–TUO, Desná-Černá Říčka, Czech Republic, 2003

M. Krátký, T. Skopal
Benchmarking the UB-tree, *Proceedings of the 3rd workshop DATESO*, pages 82–93, VŠB–TUO, Desná-Černá Říčka, Czech Republic, 2003

J. Dvorský, M. Krátký, T. Skopal, V. Snášel
Term Indexing in Information Retrieval Systems, *Proceedings of the international conference CIC'03*, pages 263–270, CSREA Press, Las Vegas, Nevada, USA, 2003

T. Skopal, V. Snášel, M. Krátký, V.Svátek
Searching the Internet Using Topological Analysis of Web Pages, *Proc. of the international conf. CIC'03*, pages 271–277, CSREA Press, Las Vegas, Nevada, USA, 2003

M. Krátký, T. Skopal, V. Snášel
Multidimenional Approach for Non-trivial Term Searching (in czech), *Proceedings of the 2nd conference ZNALOSTI*, pages 202–211, VŠB–TUO, Ostrava, Czech Republic, 2003

M. Krátký, J. Pokorný, T. Skopal, V. Snášel
Geometric Framework for Indexing and Querying XML Documents, *Proceedings of the First Eurasian conference EurAsia-ICT*, pages 35–46, LNCS 2510, Springer-Verlag, Shiraz, Iran, 2002

M. Krátký, J. Pokorný, T. Skopal, V. Snášel
The Geometric Approach for Indexing XML Data, *Proceedings of conference DATAKON*, pages 263–274, Masaryk University, Brno, Czech Republic, 2002

T. Skopal, M. Krátký, V. Snášel
Geometric Indexing and Querying of Multimedia Data (in czech), *Proceedings of conference DATAKON*, pages 329–334, Masaryk University, Brno, Czech Republic, 2002

T. Skopal, M. Krátký, V. Snášel
Properties Of Space Filling Curves And Usage With UB-trees, *Proceedings of the Workshop on Information Technologies - Applications and Theory (ITAT)*, pages 155–166, P.J. Safarik University, Malino Brdo, Slovakia, 2002

V. Snášel, T. Skopal, D. Ďuráková
Navigation Through Query Result Using Concept Order, *Proceedings of the 6th conference on Advances in Databases and Information Systems (ADBIS)*, Research Communications, pages 195–205, STU Press, Bratislava, Slovakia, 2002

T. Skopal
ACB Compression Method and Query Preprocessing in Text Retrieval Systems, *Proceedings of the 2nd workshop DATESO*, pages 36–43, VŠB–TUO, Desná-Černá Říčka, Czech Republic, 2002

Appendix B

Implementation Issues

The Framework

We have completely reimplemented the M-tree in C++ using Amphora Tree Object Model (ATOM) – a framework for advanced tree-based indexing – developed by Amphora Research Group (ARG) at VŠB–Technical University of Ostrava, Department of Computer Science.

Thus, we have completely reimplemented the M-tree (we have not used the public GiST-based implementation). In particular, in our implementation we have optimized some critical operations, e.g. we have eliminated majority of dynamic memory allocations, which is a very expensive operation comparable to a physical access to disk. Due to the optimizations our M-tree implementation is roughly 12 times faster (and more stable) than the original GiST-based one.

Measuring I/O Costs

We have measured logical I/Os in manipulation with indices, i.e. the I/O counters have been incremented whenever a node page retrieval/storage was requested, regardless if the node already remained in the disk cache. Although many physical disk accesses could be eliminated due to caching, the logical I/Os give a more clear view about the pure index performance.

Index