MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Combining Multi-metric Queries

MASTER'S THESIS

Petra Kohoutková

Brno, May 2008

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: prof. Ing. Pavel Zezula, CSc.

Acknowledgement

I would like to thank my advisor, prof. Ing. Pavel Zezula, CSc., for his support and guidance during my work on this thesis. I am also very grateful to RNDr. Michal Batko, Ph.D. for his advice, willingness to help and patience. And I am indebted to Bc. Tereza Kohoutková for her language and style corrections. Last but not least I would like to thank my family for their patience and general support.

Abstract

This thesis focuses on the problem of processing combined queries in the MUFIN similarity search system. The combined multi-metric queries allow a user to define the similarity function for any two objects, based on several descriptors of the objects (metrics). The existing baseline algorithm is explored and a modification for approximate searching is proposed. This novel method corresponds with the MUFIN distributed architecture and conforms to the cost restrictions of an interactive application. The resulting implementation experimentally evaluated on a large real-life data collection.

Keywords

Similarity search, distributed top-k query, combined query, approximation, quality evaluation

Contents

Chapter 1

Introduction

Today's society is often described as the Information Society, because information is its most valuable property. People produce and use large quantities of information, most of which exists in digital form nowadays and is freely accessible. However, the amount of digital data rapidly grows in both size and variety. To cope with such volumes, we need ways of efficient storing and retrieval of the relevant data and therefore suitable data structures and algorithms have to be found.

This fact is well recognized from the beginning of computer science and a significant part of research focuses in this area (which is confirmed by the big conferences VLDB, SIGMOD, ICDE etc.). For the basic data types, the problem is solved by traditional retrieval techniques, typically based on sorting routines and hash tables. However, as various new data types such as multimedia appear, these methods are no longer sufficient. One of the reasons is that sorting is impossible for many of the new data domains. For illustration, there is no way of ordering colors. We can sort them according to their similarity with respect to a specific hue, for example pink. But we can't sort the set of all colors in such a way that, for each hue, its immediate neighbor is the hue most similar to it.

Therefore, more flexible methods are required which take into account the needs of particular users and particular application domains. Similarity searching based on metric space model is a feasible approach which has been studied in the last years. Its usability is very wide, as it does not require the data to have any special properties – except for the ability to determine the distance or similarity of any two data objects. Thus, similarity queries retrieve objects similar to their query objects rather than ones exactly matching in some attributes. Such proximity concept is much more usable for many data domains while it also supports the traditional exact match.

Similarity searching has become a fundamental computational task in a variety of application areas, including multimedia information retrieval, data mining, pattern recognition, biomedical databases, data compression and statistical data analysis. New index structures and queries were proposed as well as algorithms for similarity searching, with some of them using distributed environment to enable processing of large data collections. However, many aspects of similarity searching remain open to further improvements, among which the speed is one of the most pressing. Even the best structures and algorithms proposed so far run into problems when the data collection becomes too large.

In this work, we focus on approximation as the way of increasing speed of query evaluation for a specific type of queries over large datasets. Approximation techniques make

use of the fact that, in large collections, the very best results are not much different from quite good results while the time needed to acquire the best results is significantly longer. Thus, we gain speed without losing much of the result precision. An approximate algorithm is proposed in this thesis and implemented within the MUFIN[1] system developed at the Faculty of Informatics of Masaryk University. We confirm the validity of our concept by thorough experiments which evaluate usefulness of our approximation and find optimal values of its parameters. Furthermore, we present and test a method for estimating quality of the approximate results.

1.1 Work Objectives

The aim of this work is to design and experimentally evaluate a technique for solving combined queries in the MUFIN system as well as propose a measure for quality estimation of results in the case of approximate searching. Specifically, the objectives are:

- to discuss possible ways of effective and efficient combined query implementation;
- to propose quality measures and design experiments for evaluating the implementation of combined queries;
- to test the quality of approximate search in thorough experiments and discuss the results;
- to propose a method for real-time estimation of the quality of results returned to the user by approximate search;
- to test and discuss the quality of the proposed estimation.

The thesis is organized as follows. Chapter 2 covers the theoretical background of similarity searching, basic types of similarity queries, reasons for approximate searching and the use of distributed environment. In Chapter 3, we describe the MUFIN system as a general multi-purpose software and its specific application on images. In Chapter 4 we focus on complex queries. The baseline algorithm as well as its possible implementations are discussed. Next, we demonstrate the need for approximate algorithm, suggest such algorithm and present several strategies for evaluation of approximate results. We also propose an algorithm for results quality estimation that can be executed in parallel with the query execution. Chapter 5 embraces the results of thorough testing of both proposed algorithms. We conclude the results in Chapter 6 and outline our future work.

1. MUlti-Feature Indexing Network

Chapter 2

The Similarity Search

This chapter gives an overview of similarity approach to the searching problem. First, the motives of this approach are explained. Then the underlying mathematical concept of metric space is defined and its fundamental properties are explained. We show how to use this to build specific data structures and what similarity queries can be evaluated over these structures. Finally, we explain how parallelism can be utilized to create scalable systems.

2.1 The Similarity Paradigm

As we have mentioned earlier, the traditional database systems are no longer sufficient for storing and searching modern data with high internal-structure complexity. Algorithms that work well for numbers and strings cannot be used for images or video straightforwardly. One of the reasons is that no total sorting function exists for many of the new data domains. Therefore, more general approaches have to be employed to store and retrieve data, such as the similarity search model [6]. A general abstract model for this approach is the metric space, which only requires that pairwise distances of objects can be measured. This distance determines the similarity or dissimilarity of the two objects. A query posed to a similarity search system will return objects most similar to the query object.

2.2 The Metric Space

The metric space is considered to be the most general data structure of similarity which can still be indexed and searched efficiently. It treats data as unstructured objects together with a function that measures distance (or similarity) between pairs of objects. This function is used to sort objects in a dataset with respect to their distance from a given query object, and determine the ones most similar to the query object. Though the sorting can be done using any function, experience has shown that some functions are more suitable for metric searching than other ones because of certain properties that can be used with advantage in searching algorithms. Therefore, several restrictions were defined that must hold for a function to make it eligible for a distance measure.

A metric space $\mathcal{M} = (\mathcal{D}, d)$ is defined [6] as a pair of domain of objects (or the objects' keys or indexed features) $\mathcal{D}$ and a total (distance) function[1] d. The properties required of the

1. function d is also called a *metric function* or simply a *metric*

function $d : \mathcal{D} \times \mathcal{D} \mapsto R$, sometimes called the *metric space postulates*, are typically defined as follows:

$\forall x, y \in \mathcal{D}, d(x, y) \geq 0$	non-negativity,
$\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$	symmetry,
$\forall x, y \in \mathcal{D}, x = y \Leftrightarrow d(x, y) = 0$	identity (reflexivity),
$\forall x, y, z \ \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality.

2.3 Distance Measures

The distance functions express similarity of objects from a given domain in the metric space. Different functions are used for various types of data and specific applications so that such characteristics of the data that are important for the application are reflected by the distance measure. In this section, we present some examples of distance functions and describe one simple metric function used in our system in more detail. In practice, distance functions are usually more complex and are specified by domain experts. However, the choice of distance function does not influence the range of queries that can be asked with this metric.

Depending on the character of values returned, distance measures can be divided into two groups:

- discrete – distance functions which return only a small (predefined) set of values, and
- continuous – distance functions in which the cardinality of the set of values returned is very large or infinite.

An example of a continuous function is the *Euclidean distance* between vectors, while the edit distance on strings (expressing the smallest number of changes that must be performed to convert one string into the other) represents a discrete function. Another example of a distance function is the *Jacard's Coefficient* used for sets which is defined as the ratio between the cardinalities of intersection and union of the sets.

2.3.1 Minkowski Distances

The *Minkowski distance* functions [6] form a whole family of metric functions, designated as the L_p metrics, because they differ only in the parameter p. These functions are defined on n-dimensional vectors of real numbers as:

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

The L_1 metric is called the *Manhattan distance* or the *City-Block distance*, for in two-dimensional space it can be explained as the shortest way between two blocks in a city with regular rectangular net of streets. The L_2 distance is known as the Euclidean distance, and the $L_\infty = \max_{i=1}^{n} |x_i - y_i|$ is called the *maximum distance* or the *chessboard distance* – in two dimensions it can be seen as the number of chessboard squares visited on the shortest way between two squares. The L_p metrics are used to compare vectors with independent coordinates, e.g. found in measurements of scientific experiments.

2.4 Similarity Queries

A similarity query is usually defined by a query object q (even though similarity queries without query object also exist) and conditions that must be satisfied by resulting objects, typically expressed as a constraint on their distance from the query object. The query result may return all qualifying objects or just the best ones if the result size is restricted (the best ones being the ones most similar to the query object). On the following pages, we first define elementary types of similarity queries shortly (see [6] for more detailed description), and then discuss possibilities of combining more similarity measures in a single query. As we work with images later, we will use images and their features (color and shape) to illustrate the use of each query type.

2.4.1 Range Query

A *range query* is one of the most common types of similarity query. It is used to acquire objects within certain distance from a given object. Range query $R(q, r)$ is specified by a query object $q \in D$ and some query radius r that restricts the distance. The query retrieves all objects found with distance to q at most equal to r, formally:

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

Objects in the response set are usually ordered with respect to their distance from q. Let us observe that the query object q need not be in the collection $X \subseteq \mathcal{D}$ that is searched, and the only restriction on q is that it belongs to the metric domain $\mathcal{D}$. In an image searching application, a range query can formulate the requirement: *Give me all images whose "redness" differs from the "redness" of my image of a red ball by maximally* $10\,\%$.

A specific type of range query is the $R(q, 0)$ query called a *point query* or *exact match*. This returns identical copies of the query object (with respect to the given metric function). Point queries are mainly used in delete algorithms to locate an object that is to be removed from the database. Let us notice that in the case of images and redness measure, all images with the same colors will form the exact match, even though the pictures may be quite different.

2.4.2 Nearest-Neighbor Query

When searching for similar objects using a range search, we need to specify the maximal distance of qualifying objects. However, this may be difficult if we are not familiar with the distance function. In some cases it is easy to understand the semantics of the query radius, e.g. $r = 3$ for edit distance represents less than four edit operations between compared strings. On the other hand, Euclidean distance of two color-histogram vectors of images is a real number whose quantification cannot be so easily interpreted. If a too small radius is chosen, too few objects may be returned, while too large radius may lead to expensive computations and many irrelevant objects in the result set. Moreover, the maximal distance for range query should also be chosen with respect to the given dataset, for a query with the same radius can return empty result set in one case and too many objects in another.

An alternative way to search for similar objects is to use nearest neighbor queries. Here we look for a fixed number of objects, that are closest to the query object q. Such objects are called the *nearest neighbors* of q. Depending on the number k of neighbors we want to find, we define the $kNN(q)$ query as follows:

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

$1NN(q)$, also denoted as $NN(q)$, returns the nearest neighbor of q from the database; if q is present in the searched collection, $NN(q) = q$. When there are less that k objects in the collection, the whole dataset is returned as the result. Again, objects in response set are sorted with respect to their distance from q. If more objects are at the same distance from q, the ties are solved arbitrarily. In our image application, we can pose a query: *Tell me which three images have a shape closest to my ball.*

2.4.3 Complex Similarity Queries

All the previous queries have dealt with just one similarity predicate, e.g. color or shape. However, in many applications we want to cover more than one feature of our object. When using just colors, we cannot distinguish red circle from a red square. Therefore we need to combine more features in our query to make it more precise. Such queries, where an *aggregation function* is used to combine several metrics describing individual features into one complex descriptor, are called *complex similarity queries*.

It may seem that such combined descriptor is just another metric, only more complex. However, there is one substantial difference. Queries that consider one metric are resolved with the use of specific index structures where objects are sorted with respect to their distance under the given feature. Complex queries, on the other hand, combine existing simple metric functions (with respective index structures) through a user-defined aggregation function. In other words, user chooses the features that shall be included in the complex search and their respective weights for the overall rating at the time of query creation. Therefore, no index structures can be built in advance for overall distances to ensure efficient query pro-

cessing. Only the index structures for individual features and general properties of metric functions can be used to answer a complex query.

Standard algorithm used to resolve complex queries is called the *Threshold Algorithm*. As complex queries are the main topic of this thesis, this algorithm is fully described in Chapter 4.

2.5 Approximate Similarity Search

Similarity search in metric spaces is unfortunately very expensive for large data collections and no existing search methods can provide acceptable response times for highly interactive applications. When quick responses are required, the only possibility is to trade preciseness for efficiency and use approximate search, which is usually performed much faster but may not find the best results. Fortunately, the following observations have been made that justify the use of approximate search algorithms:

- Similarity is subjective, different objects (e.g. images) may be perceived as similar by different people. Moreover, it is difficult to describe how we "measure" similarity in our heads and express this process as a distance measure. Thus, objects returned by similarity search, even though precise according to the chosen similarity measure, may not suit the user's needs. Therefore, it does not make much difference whether the results are the very best ones or just some good ones, for either of these may seem better to the user. However, it does matter if he/she has to wait several minutes for an answer instead of several seconds.
- Similarity searching is usually an iterative process, where result from one query is used as a query object for next query. For instance, a user may be looking for an image of sunflower. The user knows what the image should look like but cannot transfer his/her ideas to the computer. So he/she searches for any flower, chooses one image most similar to his/her idea from the result, issues a new query, finds more flowers, etc. Again, speed is much more important than preciseness in this process.

There are several approaches that address the issue of approximate similarity searching. Some of them try to reduce the subset of data that is examined by search algorithms, other work with transformations of the metric space. The latter are more suitable for vector spaces, the former can be further divided into two classes: early termination techniques and relaxed branching strategies. We shall focus on the first strategy which is especially useful for iterative algorithms such as complex query processing.

2.5.1 Early Termination Strategies

Many similarity search algorithms work in iterations. At the beginning, some qualifying objects are found, then better and better objects are added to the result set replacing worse ones until some stop condition is met that ensures that no further improvement is possible. Early

termination strategies use the fact that usually good objects are found quickly but many iterations are needed to meet the stop condition. Thus, later iterations increase the costs but do not add much to the results quality. Approximation algorithms therefore terminate the query processing earlier, as soon as they detect that the chance of getting significantly better results is small.

2.6 Similarity Searching in Distributed Environment

The challenges brought by current information explosion are not just the variety of data types but also the rapidly growing amounts of data which need to be processed in real-time. Experience shows that algorithms over centralized index structures cannot search large data with reasonable response time. Thus the usability of centralized indexes for growing datasets, i.e. its *scalability*, is limited.

This problem can be solved by employing more computing centers (called *nodes*) connected through a network. Nodes cooperate and solve parts of a given task in parallel. Computing resources (CPUs, memory, etc.) can be added to the system when the amount of data increases so that time needed for task processing remains low.

2.6.1 Scalable and Distributed Data Structures

The paradigm of *Scalable and Distributed Data Structures* (SDDS) [10] defines three properties the data structures must satisfy to behave efficiently in the distributed environment:

- *scalability* – data migrate to new nodes effectively and efficiently, new nodes are added only when nodes already connected in the network are sufficiently loaded;
- *no hotspot* – there is no master node that would store centralized information about navigation in the structure, each node knows part of this information and can navigate to its neighbors;
- *independence* – primitive operations, such as the search, insertion, or node split, do not require changes of the whole network.

In such systems, data objects are stored on specialized network nodes called *servers* that offer both storage capacity and computing power. Other nodes, called *clients*, can modify and search the data through insert, delete, and search operations executed by the servers. The number of clients is unlimited and any client can request an operation at any time.

2.6.2 Peer-to-Peer Systems

In the *Peer-to-Peer* (P2P) environment, no specialized nodes are used. All network nodes, called *peers*, are treated equally. Each peer offers some resources (i.e. acts as a server) but can also use resources of the other peers (act as a client). Peers are connected through a large-scale network that may be unreliable.

P2P systems are similar to SDDSs but add new requirements to overcome problems with network failures. The basic principles of P2P can be summarized as follows:

- *peer* – every node behaves as both a client and a server, i.e. can perform queries and, at the same time, store part of the processed data;
- *fault tolerance* – the system can operate even if some of the nodes are not working. All defined operations can still be performed but the results may be imprecise as part of the dataset can be inaccessible;
- *redundancy* – data objects are replicated and stored on multiple nodes to increase availability in case of network failures.

2.7 Index Structures for Similarity Searching

To evaluate similarity queries efficiently, we need a quick access to the relevant data. This is provided by specialized index structures that organize objects from the given domain. The organization is based on the distances between objects under the given distance measure. Typically, the structure is designed as some kind of a tree, where objects are divided into subtrees with respect to their distance from the object in the parent node (e.g. half of objects with lower distances in the left subtree, the other half in the right subtree). All similarity searching structures use properties of distance function (as defined by the metric postulates) to navigate between substructures (e.g. the triangle property is often employed to decide whether a substructure can contain qualifying objects).

Search structures can be divided into two main groups:

- centralized structures have been designed to run on a single computer and work well for small data collections; *Vantage Point Tree*, *Generalized Hyperplane Tree* or *M-tree* are only a few examples of centralized indexes (see [6] for more information about these structures);
- distributed structures employ parallel computing power of a network of computers, thus enabling to create scalable systems; *GHT** [6], *Skip-Graphs* [2] or *M-chord* [3] are possible implementations of distributed index structure (their description and comparison is provided in [12]).

Chapter 3

Similarity Searching in MUFIN

In the previous chapter, we have shown that only distributed systems can effectively deal with scalability in similarity searching. In this chapter, we will present an example of such a system called MUFIN, which is being implemented at the Faculty of Informatics of Masaryk University (FI MU). We describe both the underlying library and the system itself, and show one real application of this multi-purpose system in the area of image searching. Efficient complex query evaluation for this application will then be the main topic of the following chapters.

3.1 The Metric Similarity Search Implementation Framework

The *Metric Similarity Search Implementation Framework* (MESSIF) [7] is a development platform which facilitates implementation of similarity search systems. It was created at FI MU as a part of research in the area of similarity searching. MESSIF provides basic operations for data management and enables easy implementation of queries. It also supports sharing and reusing the code as well as efficient testing and comparison of the results. General overview of functionality provided by MESSIF is given in the following list:

- encapsulation of the metric space concept – developers can use the data objects transparently regardless of the specific dataset and metric function;
- management of the metric data – data objects are stored in primitive storage units called *buckets*, buckets-splitting based on the metric indexing principles supports scalability;
- concept of operations – a uniform interface is provided for queries and operations that modify the data, complex similarity queries can be implemented in multi-metric spaces;
- support for distributed data structures – load balancing system and communication layer is provided;
- automatic performance measurement and collecting of various statistics including a uniform interface for accessing and presenting the results.

3.2 The MUlti-Feature Indexing Network

The *MUlti-Feature Indexing Network* (MUFIN) is a general tool for effective and efficient similarity searching in large and quickly growing data collections. It is able to index and search any metric data and supports definition of several search indexes (called *overlays*) that describe different features of data objects. The system is built over P2P network so it can manage large volumes of data and provide acceptable response times.

3.2.1 Distributed Metric-search Structures

The basic schema of P2P environment is depicted in Figure 3.1. Each peer manages part of the indexed data collection. Different strategies can be employed to store and search the data, either a simple list with sequential search applied for searching, or more sophisticated methods. Every peer also holds some local navigation information and can communicate with other peers via messages sent through the underlying network.

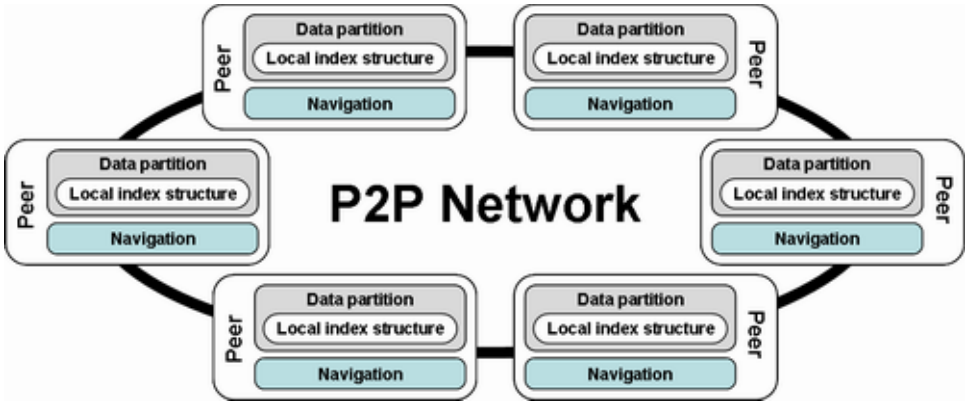

Figure 3.1: A typical schema of a P2P network

As the system is based on P2P architecture, a user can issue a query at any peer. Steps which are executed to answer the query are depicted in Figure 3.2. The query is forwarded to peers that may hold qualifying objects (solid arrows). Since the network changes in time as objects are added or deleted, navigation can be imprecise and query has to be forwarded several times until it reaches all relevant peers. Each peer with promising data partition executes local search and returns all qualifying object to the initiatory peer (dotted arrows) where the final answer is merged and presented to the user.

3.2.2 Multi-layer System

As anticipated earlier, it is often not possible to simulate human understanding of similarity between complex objects (such as images, music, video, etc.) using a single metric. Much better results can be achieved when several important features of data objects are described

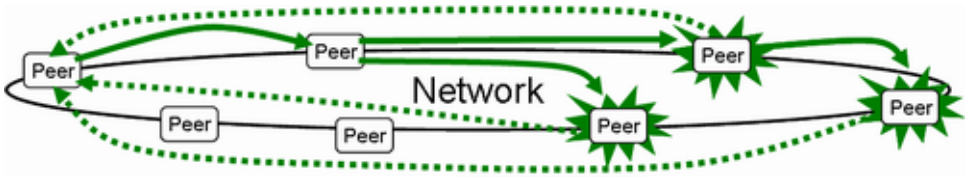

Figure 3.2: A typical query processing in P2P network

and compared by different metrics, and overall similarity of objects is then computed from these partial similarities using some aggregation function.

MUFIN supports this approach and enables definition of several indexes for a given data collection. Each index is built over a metric that describes certain property of data objects. Each object is thus represented by a set of descriptors and a unique identifier which together form a metaobject. For instance, an image can be represented by color and shape descriptors. For each descriptor, a P2P index (overlay) is built, storing the descriptor value and the metaobject's identifier. Moreover, additional index called *zero overlay* is defined where the complete metaobjects are stored. The zero overlay enables efficient retrieval of objects once their identifier is known while the descriptor overlays allow similarity searching with respect to the given descriptor. All the overlays can share the same physical infrastructure.

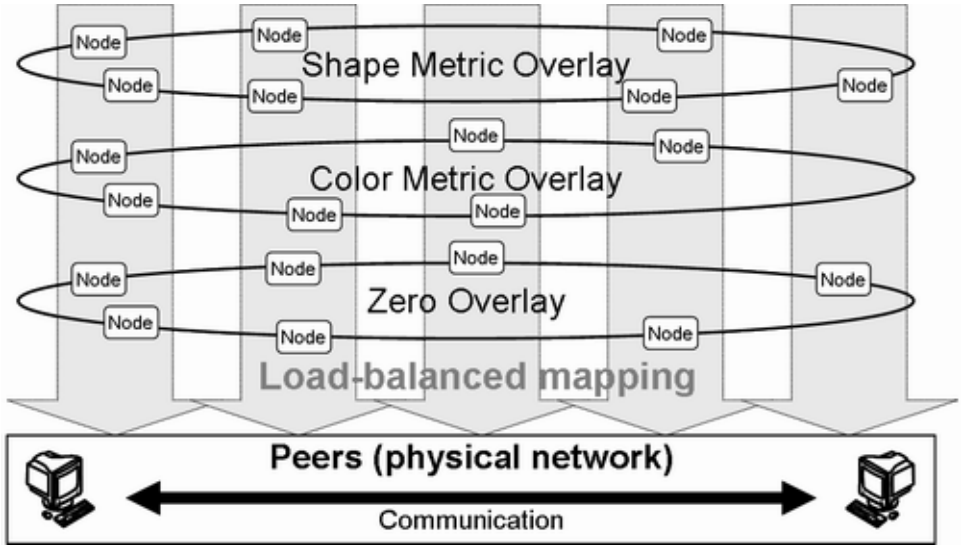

Figure 3.3: Multi-metric overlay setting

Figure 3.3 depicts a system with three overlays which can be used for image searching. The first overlay is built for color descriptors, the second indexes shapes, and the third represents the zero overlay. A metaobject assignment to these overlays is also illustrated in the figure. Observe that each overlay consists of multiple nodes and their specifics are left up to

the particular distributed index structure used in the overlay. These nodes are maintained by physical peers (illustrated by the gray arrows). Each peer usually manages one node from every overlay. Such a mapping is completely transparent for overlay index structures and, in general, it is automatically done by the loadbalancing mechanism.

3.3 MUFIN Application in Image Searching

Since MUFIN is a multi-purpose system, it can be applied to many similarity search problems. Given a specific application, a MUFIN instance only needs a specification of the following parameters to organize the data:

- the metric space for each descriptor – the domain and the distance function;
- the index structure of each overlay – the actual implementation of a single-metric P2P system used for individual overlays;
- the local storage index for each overlay – different local structures can be used within nodes of different overlays.

In our research, we use a MUFIN instance for image searching, where the results can be easily visualized and compared with human understanding of image similarity [9]. To test MUFIN performance for large volumes of data, we have indexed a collection of 2 million images (about 35 GB of data). Most of the images are outdoor and indoor taken photos but there are also a few images of e-shops products (e.g. mobile phones, PDAs, CD players, etc.), cartoon characters or handdrawings and paintings.

In this image search system, objects are compared using features (characteristics) extracted from them that give evidence about the visual content of the images. Namely, we use features defined in the MPEG-7 standard [8]. The MPEG-7 also defines a metric function for each of the features to measure the distance (dissimilarity) within the particular feature domain. Each feature is represented by one layer in the MUFIN system.

3.3.1 MPEG-7: Multimedia Content Description Interface

MPEG-7 is an ISO/IEC standard for describing the multimedia content data. MPEG-7 Visual Description Tools included in the standard consist of basic structures and descriptors that cover the following basic visual features: color, texture, shape, motion, localization, and face recognition. From the total number of twenty descriptors, five have been chosen to represent each image in MUFIN: *Scalable Color*, *Color Structure*, *Color Layout* (all of them describing color), *Edge Histogram* and *Homogeneous Texture* (both describing texture).

3.3.2 Distance Functions

Various metrics have been chosen to compare the extracted features. Simple L_1 metric is used for Scalable Color and Color Structure descriptors. For the other features, special func-

tions have been designed by experts. For example, the distance between the features of Color Layout descriptor is defined as a sum of weighted L_2 metrics for individual color channels.

3.3.3 Aggregation Function

When searching in a collection with several descriptors, we need to be able to determine the overall distance between any pair of objects. This is computed from partial descriptor distances via an aggregation function. In the basic setting of our image searching application we use a weighted sum of the partial distances. The weights (shown in Table 3.1) have been determined experimentally trying to reflect the human notion of image similarity. However, a user can choose a different aggregation function that reflects his/her preferences.

MPEG-7 Feature	Distance	Weight
Scalable Color	L_1 metric	2
Color Structure	L_1 metric	3
Color Layout	special	2
Edge Histogram	special	4
Homogeneous Texture	special	0.5

Table 3.1: MPEG-7 descriptors, the respective distance measures and the weights for aggregation sum

3.3.4 Network Settings

For our experiments we were using a MUFIN instance with six overlays (five for the descriptors and the zero overlay). There were 100 peers employed in the network, each of them holding up to five logical nodes of any of the six overlays. These peers are run on a physical infrastructure with 16 CPUs (AMD Opteron 2.6MHz) and 64GB RAM in total. The GHT* [6] structure was used as the overlay index, and M-Trees [6] were employed to store the descriptors locally at respective peers. For the zero-overlay, Skip-Graphs [2] distributed hash-table was used.

Chapter 4

Combined Queries in MUFIN

In Chapter 2 we have presented several index structures that can be used for efficient evaluation of similarity queries over a single metric. However, the usability of such queries is limited since the user cannot influence the similarity measure. Depending on the situation and individual preferences, we may wish to focus on various properties of a query object in particular queries (as depicted in Figure 4.1).

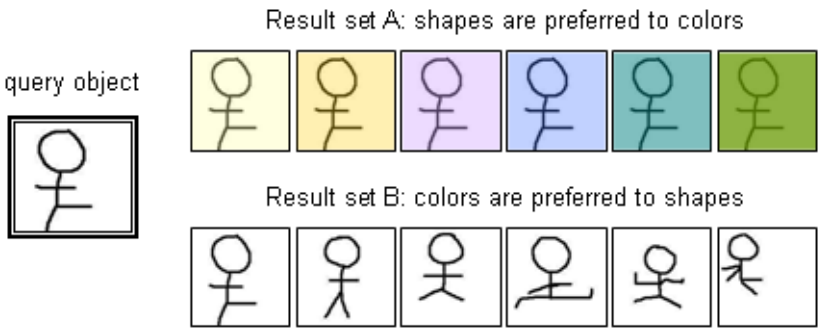

Figure 4.1: Combined query results with focus on different descriptors

For this type of searching we cannot build any index structure in advance. Instead, we define several descriptors for objects of given domain that are then used to compute the overall distance. For these descriptors we can prepare the search structures. In the time of combined query evaluation the structures give information about objects' proximity to the given query object under the particular descriptor. To get a good similarity searching tool, several problems have to be solved. We need to find the right descriptors, combine them suitably to get the overall distance measure, and also find an implementation that will provide correct and fast results.

In this work we do not address the first two issues but we focus on the last problem in the specific environment of the MUFIN system. We will present the basic algorithm used for *top-k complex similarity queries* (i.e. the queries that ask for k nearest objects to a given query object) and discuss its possible implementations within MUFIN. In the following sections, we will give reasons for the use of an approximate algorithm, describe its implementation and propose methods for its evaluation. We will also present an algorithm designed to estimate the quality of results acquired by approximate searching.

4.1 The Basic Algorithm

The general top-k combined query problem can be defined as follows: Assume a dataset of n objects, each having m grades, or scores, one for each of m attributes. The grades are determined with respect to a query object, where high grades signify interesting objects. For each attribute, there is a sorted list, which lists each object and its grade under that attribute, sorted by the grades (highest grade first). Each object is assigned an overall grade, that is obtained by combining the attribute grades using a fixed monotone aggregation function. Our task is to find k objects with the highest overall grades.

A naive way of determining the top k objects in this model is to scan all the lists, compute the overall scores and select the k highest scored objects. However, this would need accessing all objects which is very inefficient for large databases. The Threshold Algorithm (TA), which was proposed by Ronald Fagin et al. in [1], solves the task much more efficiently (and is the best general algorithm known for this problem). Even though our model in similarity searching is different – the distances express dissimilarity instead of scores, which means that the best matches have the lowest distance as opposed to the highest grades in Fagin's model – the algorithm can be inverted easily to suit the distance measures. In the following sections, we will present a modification of TA for the similarity model and prove that it gives correct answers to combined queries.

In the beginning, we need to define basic terminology: let q be the query object and $o_1, o_2, ...$ the objects to be searched. Let $d(q, o_1)$ be the overall distance between q and o_1 and $d_i(q, o_1)$ the distance under the feature i (*feature* or *descriptor* is an equivalent of attribute in our model).

4.1.1 The Aggregation Function

When two complex objects are compared, we obtain their distances in each feature under consideration. The aggregation function is used to combine these distances into one overall distance representing the overall similarity of these two objects. For this purpose, we naturally demand the aggregation function to be monotone: if for every feature the distance of object o_1 from object q is lower than or equal to the distance of object o_2 from object q, then the overall distance of o_1 from q must be lower than or equal to the overall distance of o_1 from q. Typical examples of aggregation function are the *minimum* or the *average*.

Formally, let us consider m features describing each object and m-ary aggregation function f. If $d_1(q, o_1), \ldots, d_m(q, o_1)$ are the distances of object o_1 from object q under each of the m features, then $f(d_1(q, o_1), \ldots, d_m(q, o_1)) = d(q, o_1)$ is the (overall) distance between q and o_1. The monotonicity condition is defined as

$$d_1(q, o_1) \leq d_1(q, o_2), ..., d_m(q, o_1) \leq d_m(q, o_2) \Rightarrow d(q, o_1) \leq d(q, o_2)$$

Usually, the overall distance is also a metric but it is not necessary. Let us consider a simple aggregation function $f(x) = x + 1$, which is monotonous but the reflexivity property

does not hold: $f(d(q,q)) = f(0) = 1$. However, the overall distance expresses the similarity in the same way as metric functions – the higher is the value of the distance the more dissimilar the objects are.

When executing a query, a set of objects and their distances from the query object is returned to the user. However, if an object o and distance 1.2456 is returned to the user, he may not have much information on how good the result is if he/she is not familiar with the distribution of distances over the dataset. In such case, it may be better to transform the aggregation function in such a way that all the distances fall into the interval [0, 1] (with 0 being the best match and 1 the worst). This can be done easily when the maximal distances under individual features are known for the dataset. Let $\delta_1^{max}, \ldots, \delta_m^{max}$ be the respective maximal distances. Then $f_{NORM}(d_1(q,o_1), \ldots, d_m(q,o_1)) = t(d_1(q,o_1), \ldots, d_m(q,o_1))/f(\delta_1^{max}, \ldots, \delta_m^{max})$ is the normalized aggregation function.

The aggregation function used for image searching in MUFIN is a weighted sum of distances under the individual features (see Section 3.3.3 for more details). As all the coefficients (weights) in the sum are non-negative, the function is monotonous and the resulting overall distance is a metric.

4.1.2 Requirements on the Model

Let us now describe the model more formally. We assume that each dataset consists of a finite set of objects (we will typically use n as the number of objects). For each object o and each feature i, the distance $d_i(q,o)$ can be acquired. We consider two modes of access to data – the sorted (or sequential) access and the random access.

- Sorted access is used to get the sorted list S_i of objects from the dataset and their distances to the query object under a given feature i. The list is sorted by distances, the lower is the distance the sooner the object appears in the list.
- Random access can retrieve the distances between the query object q and any given object o for all the descriptors (that is, the partial distances $d_1(q,o), \ldots, d_m(q,o)$ if m descriptors are used) and thus compute the overall distance $d(q,o)$.

Figure 4.2 shows how both the modes can be applied in the case of image similarity searching with two descriptors, color and shape. The sequential access is used to find interesting objects (i.e. the objects near to the query object under some features), the random access then gives complete information about such objects so that the overall distance can be computed.

4.1.3 The Threshold Algorithm

The Threshold Algorithm is the basic algorithm used for combined query evaluation. In a modification for similarity searching, the TA works as follows:

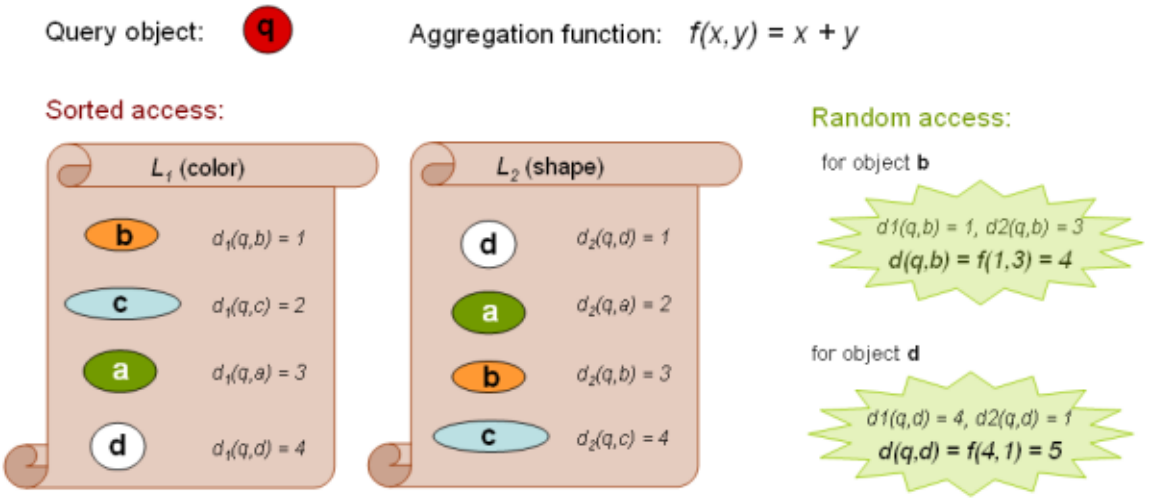

Figure 4.2: Sorted and random access mode for the multi-feature search

1. Do sorted access in parallel for each of the m descriptors in order to get the sorted lists $S_1, \ldots, S_m$. Let o_1^1 be the top-most object in S_1, o_1^2 the second from the top with respect to sorting in S_1 etc.

2. In iteration j, get the j^{th} object from each sorted list, i.e. objects $o_1^j \in S_1, \ldots, o_m^j \in S_m$ having the respective descriptor's distances $d_1(q, o_1^j), \ldots, d_m(q, o_m^j)$.

3. Do random access to the other lists and use aggregation function to compute the overall distances $d(q, o_1^j), \ldots, d(q, o_m^j)$ for objects that haven't yet been examined. Update the list R of the resulting k objects with the lowest overall distances (ties are broken arbitrarily, so that only k objects and their grades need to be remembered at any time). Let d^{max} be the distance of the k^{th} object, i.e. the maximal distance of the result.

4. Define the threshold value t to be $f(d_1(q, o_1^j), \ldots, d_m(q, o_m^j))$, i.e. the result of aggregation function applied on the highest distances seen so far for each of the descriptors. Start next iteration in step 2 unless the result list has k objects and $d^{max} \leq t$, or all objects from the dataset have been examined.

The output is then the sorted set $\{(o, d(q, o)) \mid o \in R\}$.

4.1.4 Proof of Correctness

If there are less then k objects in the dataset, the algorithm trivially gives the correct result. For datasets D with at least k objects we will prove the following theorem:

For any monotone aggregation function f, the Threshold Algorithm correctly finds the k nearest objects to the query object q , i.e. the k objects $o_R^1, o_R^2, \ldots, o_R^3$ such that $\forall 1 \leq i \leq k, \forall o \in D \setminus R : d(q,o) \geq d(q,o_R^i)$.

It is obvious that the algorithm always stops in finite time, in the worst case when all objects from the (finite) dataset have been seen (see the stop condition in step 4.). We only need to show that when the algorithm stops, the result set R contains the k objects nearest to the query object q.

Let us suppose there exists an object x which has not been searched but has the overall distance lower than some of the best k results found before the algorithm stopped in the j^{th} iteration. In other words, suppose $d(q,x) < d^{max}$. However, for each feature i, the distance of x under this feature is at least equal to $d_i(q,o_i^j)$ (the distance of the last object seen under sorted access in list S_i), otherwise it would have appeared in the sorted access earlier. The aggregation function is monotone, therefore the overall distance of x must be at least t. But all objects in the result set have overall distance lower than or equal to t. Thus, no object with overall distance lower than d^{max} can exist, which completes the proof.

4.2 TA Implementation within MUFIN

The MUFIN system and its functionality (as described in Chapter 3) suit the requirements of the TA very well. Nearest neighbor queries on descriptor overlays can be used as sorted access to objects under that descriptor. The zero overlay provides random access to the complete metaobject when its identifier is known, so the distances can be computed for all descriptors. However, implementing the algorithm exactly as described in the previous section would be very expensive in the distributed environment. We wouldn't use the advantages of parallelism and, moreover, single object retrieval is very inefficient in a distributed system. In the P2P network, queries are typically forwarded between several peers and many objects are accessed before the result is returned. So the network is better utilized if batch queries are issued, where more objects are retrieved simultaneously. Therefore, batch mode is used for both sorted and random access in our implementation.

For the sorted access, the issuing peer breaks the query metaobject into its descriptors and executes the nearest neighbor query for every descriptor in the respective similarity-search overlay. These are evaluated in parallel, and a sorted list of the top-most similar objects is returned by the system for each descriptor. However, not the full lists of all objects in the dataset are requested as it is probable that only a part of each list will be used. Therefore, a κ_1-nearest neighbor query is executed for each descriptor, where κ_1 is a suitable coefficient determined by experiments (we also refer to this as *batch sorted access* and call κ_1 the *batch size*). Objects retrieved from these queries are submitted to random access to get distances for missing descriptors, and overall distances are computed. If there are not enough objects with their overall similarity under the threshold value, the descriptor overlays are requested to provide additional batch of objects until this condition is met. This additional batch can be executed either as incremental $\kappa_2 NN$ query, if the system enables it, or a new query must

be issued for $\kappa_1 + \kappa_2$ nearest objects (which is of course less efficient, but presently the only possible way in the GHT* structure, which is used as the overlay index in MUFIN).

In the case of random access, we do not issue a new query for each object to find the values of all its descriptors. On the contrary, we process the whole set of objects in one batch query. However, there are more possible strategies of implementing random access which differ in several aspects. One or more peers may be allowed to issue the random query and may use different ways of finding all the descriptors for a given object. On the following pages, we will introduce three different strategies and discuss their advantages and disadvantages. We will conclude this section by choosing one of the strategies and showing the limits of its practical usability.

4.2.1 Single Peer Random Access with Zero Overlay

Let p_1 be the peer from which the combined query is run. This peer splits the query metaobject into descriptors and starts the sorted accesses, i.e. issues κNN queries in descriptor overlays. All the peers that have found promising objects send them back to p_1 where the sorted lists for each descriptor are created. More precisely, for each such object, they send the object identifier and one descriptor value, as only this part of the whole metaobject is stored in one layer. Peer p_1 goes through the lists, finds if any objects are present in all lists, and eventually computes their overall distances. For the rest of objects in sorted lists, p_1 issues a query (i.e. the random access) to the zero layer, where the whole metaobjects are stored and can be accessed through the identifiers. The overall distances are evaluated and returned to p_1, where the threshold value is computed. If not enough objects under threshold have been found, the whole procedure is repeated with more objects acquired in the sorted accesses.

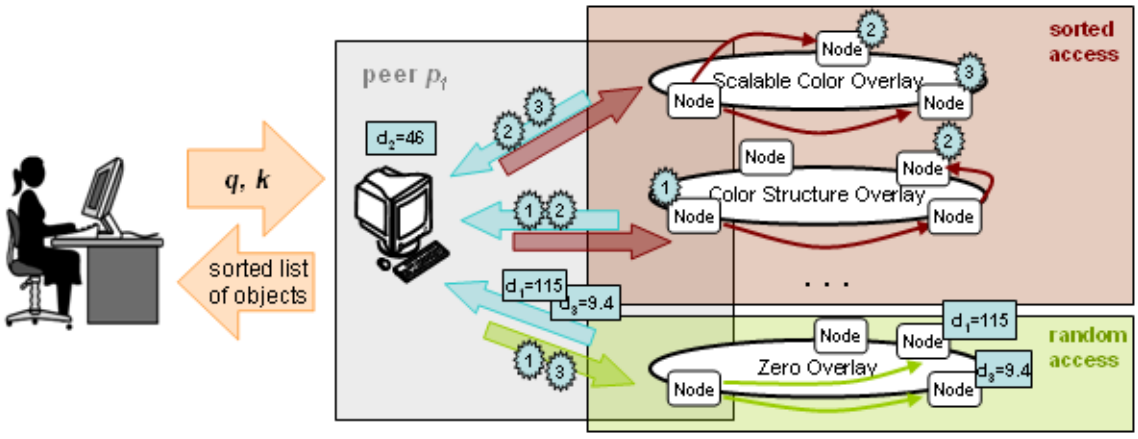

Figure 4.3: Single peer random access with zero overlay

The execution of a query following this strategy is demonstrated in Figure 4.3. We can see that p_1 is a central peer that does most of the work. Thus it becomes a bottleneck – all

objects must be first sent here from the sorted access before the random access can be run. An advantage of this implementation is that the random access is used at most once for every object (if an object is seen in all lists after the sorted access it can be even skipped from the random access completely).

4.2.2 Multiple Peer Random Access with Zero Overlay

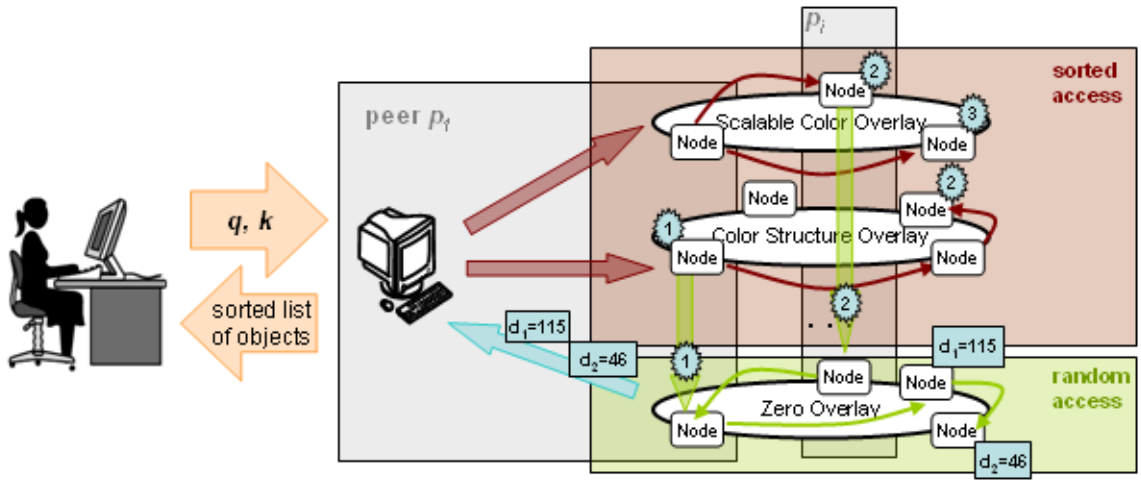

Figure 4.4: Multiple peer random access with zero overlay

Another strategy is depicted in Figure 4.4. The principal difference is that peers accessed by sorted access do not return qualifying objects immediately to p_1. Instead, the peers run the random access for these objects, i.e. issue a query to the zero layer where the whole metaobjects are found, get their overall distances evaluated, and send the distances and the respecive identifiers back to p_1. Thus, the random access is run in parallel from a number of peers, which increases the speed of query execution. On the other hand, an object can be found by more than one peer (when it qualifies for more than one descriptor, which happens quite often). In that case, unnecessary random queries are issued. Additionally, network traffic is much higher.

4.2.3 Sorted Access with Objects Replication

While we have discussed in the previous strategies which peers shall query the zero layer to get all the descriptors, here we examine a quite different approach. The zero layer and random access is not needed at all since the whole metaobjects are stored in each layer (i.e. each metaobject is stored several times, one instance for each layer). Such approach is depicted in Figure 4.5.

If we use this architecture, the qualifying objects can be sent back with their overall distances computed directly from the sorted accesses. However, in this case much more in-

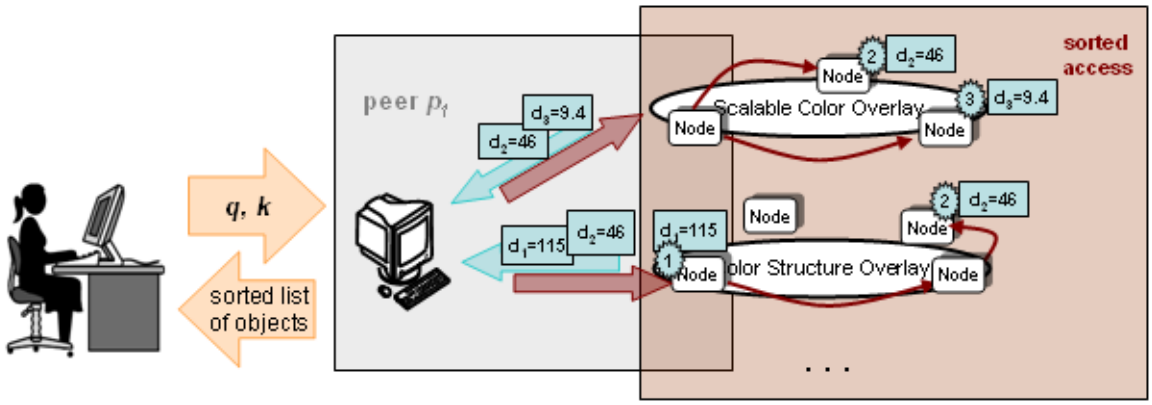

Figure 4.5: Sorted access with objects replication

formation needs to be stored in each layer, which requires more peers to be employed and searched. When the zero overlay is used, each metaobject is stored twice – one copy in the zero overlay and parts of the other copy in the descriptor overlays. Without the zero overlay we would need to have a separate copy of the metaobject for each overlay, which in case of five descriptor overlays means that 2.5 times more data needs to be stored. In addition, the same objects may be found in different overlays, which would result in needless data sending.

4.2.4 Access Strategies Conclusion

From the above described possibilities, the first one was chosen as the one which probably has the lowest cost to performance ratio. Both the other possibilities are likely to be quicker, but far more expensive in network communication (in case of multiple peer random access) or memory (if objects replication was used). Therefore, the TA was implemented in the MUFIN system by RNDr. Michal Batko, Ph.D. using single peer random access and the zero overlay. The other proposed methods were not realized as they would require non-trivial changes in the current MUFIN implementation. However, it might be interesting to compare the real performance of all these methods and see if our expectations about their efficiency get confirmed. Therefore, implementation of the TA with multiple random access and object replication is one of the objectives of future research.

Even though we have been looking for the most efficient TA implementation, it still takes a lot of time to get under the threshold for a huge data collections. For example, a top-50 query in a dataset of 1.6 million images takes more than one minute to evaluate even for a batch size of 1,000 objects. Therefore we need to look for ways of speeding up the query processing.

4.3 Approximate TA

The primary reason for considering approximate approach was the low speed of the TA implemented as described in the previous section. Fortunately, this approach is very suitable for this type of searching because the interpretation of similarity itself is highly individual and even optimal results of the search may not satisfy user needs. In such case, the user will issue additional queries to get better results. Therefore, it is more reasonable to give good-enough results quickly even if they are not precise.

In this section, we will present an approximate algorithm for complex query execution which is based on observations of the system behavior during processing of the standard Threshold Algorithm.

4.3.1 Observations on TA Behavior

Before we could start working on approximations, we had to understand the standard TA well. Therefore the first experiments were focused on describing the evolution of the result set and the threshold value during the execution of the algorithm. In these experiments, we stored the result sets (i.e. objects in the result set and their distances from the query object) and threshold values computed in individual iterations of the TA.

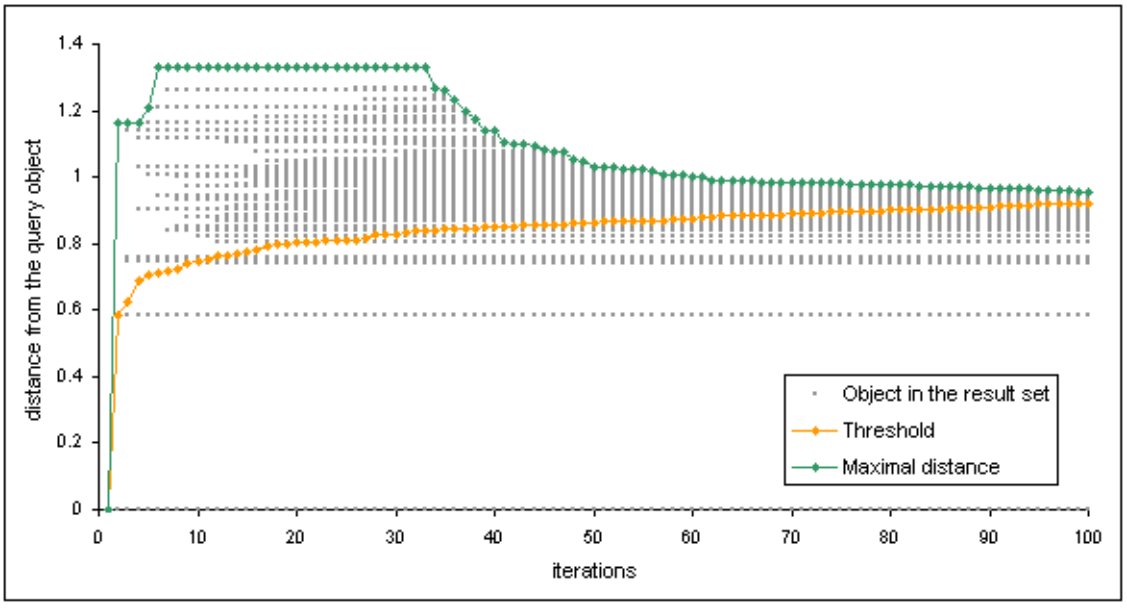

Figure 4.6: Visualization of the standard TA

The evaluation of a top-50 query on a collection of 1.6 million objects is visualized in Figure 4.6 for iterations 1 to 100. All five descriptors are combined using the aforementioned weighted-sum aggregation. Let us note that in each iteration only one step of the standard

TA is computed. Gray squares represent distances of objects in the result set in those particular iterations. The upper line represents the maximal distance d_i^{max} in iteration i, the lower curve shows the respective threshold value t_i. Let us observe that the d_i^{max} curve increases at first until the initial 50 unique objects are found. Since we have five sorted lists, 50 unique objects can be found earlier than in the 50th iteration because up to five objects can be added to the result set in each iteration (five objects are examined in each iteration but some of them may appear in several lists or may have been seen in the earlier iterations). In our case the result set was filled in the 33rd iteration. After that, the d_i^{max} curve can only decrease, because the result can only be updated with closer objects. On the other hand, the threshold can only grow. The intersection of the two curves marks the stop condition of the standard TA. The distance between the curves after each iteration corresponds to the quality of the actual result set. Experiments confirmed that the quality increases very quickly in the first couple of iterations while the rate slows down later. This observation forms the basis of our approximation.

4.3.2 The Approximate Algorithm

The above described observations led us to exploit an early termination strategy of approximation where the stop condition of the Threshold Algorithm is altered so that we end the processing prematurely after x iterations even if the threshold condition is not satisfied. More precisely, we only execute one sorted access for a batch of x objects, i.e. we issue xNN query on each of the overlays. Objects returned from the sorted access are processed as usual in the TA, and the best k objects are returned as the result even if their overall distances are not below the current threshold.

The sooner we stop the algorithm, the lower is the response time but also the worse is the results quality (i.e. the distance of the result set objects from the query object is growing). We need to choose such a number of iterations that would give good enough results in a reasonable time. The parameter x should naturally depend on k, as we need to examine at least k objects to fill the result set. Moreover, the most similar object is usually found sooner than the 10th most similar object, so it is advisable to use more sorted accesses for large k to get results with comparable qualities. Therefore we consider x to be $c.k$ for some coefficient c, and aim on determining such values of c for which the approximation would work well. However, we cannot expect to find optimal value of c that would work optimally for any data as the quality of approximate results depends on specific properties of the respective descriptors' metric spaces. If the distances between objects in the dataset are generally low, the objects found by approximation are very similar to the optimal result and not much information is lost by the approximation. On the other hand, in a system with objects far from each other, the approximate results may be very imprecise. Usually, both objects with many neighbors and solitaries appear in a dataset, which makes the choice of suitable coefficient for early termination even more difficult. Also the descriptors can influence the number of iterations needed to find good results. If there is some correlation between the descriptors, the same objects are likely to appear on the top positions in the sorted access lists. Such

objects get under the threshold much sooner and it is not probable to find good results in later iterations. In the extreme case of a linear dependence between the descriptors, all the sorted lists will be the same and the k best objects will be found in exactly k iterations (and in this case, no approximation is needed as the precise result is found in the minimal time). However, descriptors are usually not correlated; if they were, it would be better to use only one of them.

As the setting of approximate searching parameters depends on the environment that substantially, we have studied it for one specific example of similarity search system – a MUFIN instance loaded with images (see Section 3.3 for more details). This is useful both for this system which is intended for public use and also as a basis for more general studies of approximate algorithms for complex queries. In this MUFIN instance, we run thorough experiments to learn about the behavior of our approximate algorithm and to find optimal values of the termination coefficient c.

We would also like our system to tune the approximation quality according to user preferences or the actual system load. The system may accept harder approximation to keep the response times low or enable the user to choose that a longer run with more precise answer is preferred. This can be achieved by adjusting the stopping condition to these requirements. To help the system tune the value of c more precisely, we compute actual quality estimations during the iterations of the TA. These estimations are also used as a part of the query response to tell the user how good the approximate result is.

4.4 Approximate TA Evaluation

Approximate algorithms are generally used to solve difficult problems in a relatively short time, sacrificing the precision. Good approximate similarity search algorithms should have high efficiency, while still guaranteeing high accuracy of results. In some cases, it is possible to give some bounds on the quality of results, i.e. to prove that results given by an approximation are in the worst case x-times worse than the optimal results. However, such guarantees cannot be found for all problems and sometimes the only method of evaluating approximation quality is an experimental evaluation and comparison with precise results. This is the case of our algorithm, for there is no theoretical limit to "badness" of the approximate results: it is always possible to find an example of query object and dataset where the best objects are found in the very last TA iterations, and objects found in the first iterations have huge overall distances from the query object (this happens when the distance is low for one descriptor but high for all the others). Again we see here that the actual dataset must be taken into consideration for judging the usability of an algorithm.

In this section, we will present several methods that have been proposed in [1] and [6] for evaluating quality of results given by approximate algorithms. Some of these methods cannot be meaningfully used in our system but the rest of them will be applied to evaluate our approximate TA in experiments over the MUFIN image instance. All these methods compare approximate results with exact results, i.e. the results obtained by the original TA. Since we are using a fixed aggregation function in our experiments, we do not have to use

the precise TA to get the exact results (which would be costly). Instead, we define a new MUFIN overlay where objects are indexed with respect to their precomputed overall distance. This also enables us to use other query types than kNN, e.g. range query, to get the precise result set.

In the following, let R_A be the approximate result set and R_E the exact one with respect to some given query. We will also use the indexes E and A to distinguish objects in both result sets and their properties (such as the maximal distance or the threshold).

4.4.1 Fagin's Quality Measure

When Fagin et al. proposed the TA in [1] they also presented a modification that turns the TA into an approximation algorithm. This modification also uses the early termination strategy. However, the algorithm is not terminated after a given number of iterations, but instead, a modified stopping condition is used.

The authors of TA define a *θ-approximation* over their model with scores, which can be reformulated for our similarity searching model as follows: θ-approximation to the top k similarity query is a collection of k objects such that for each o_y among these k objects and each o_z not among these k objects, $d(q, o_y) \leq \theta.d(q, o_z)$. To find a θ-approximation to the top k answers, the stopping rule of the TA is modified so that as soon as at least k objects have been seen whose grade is at most equal to $\frac{t}{\theta}$, then the search algorithm halts.

We do not use this stopping condition because θ-approximation does not fulfill our idea of a good approximation method for similarity queries. We want such an algorithm that gives as good results as possible in a reasonable time. θ-approximation, on the other hand, gives results of a fixed quality without time restrictions. However, θ can be used as the quality indicator, computed as a ratio of maximal distance in the result set d^{max} and the last threshold value t^{last} computed by the approximate algorithm:

$$\theta = \frac{d_A^{max}}{t_A^{last}}$$

Let us notice that no knowledge about the precise result is needed to obtain θ. Thus in fact it does not provide reliable information about the approximation quality, only gives us the idea of how long it would take to reach the stop condition after the approximate algorithm last iteration. Theoretically, this value may be high even if the precise result is found by the approximate algorithm. This happens if the algorithm is terminated after the best objects have been found but before the stopping condition is fulfilled (an example of such case is provided in Figure 4.7a). However, it is much more probable that both the threshold and the maximal distance values develop all the time before the curves intersect (as depicted in Figure 4.7b). To cope with this, we define a modification of θ measure that takes the precise result into account, using d_E^{max} instead of t_A^{last}. We call it the *loss of quality*, LQ, as its values are higher for worse results.

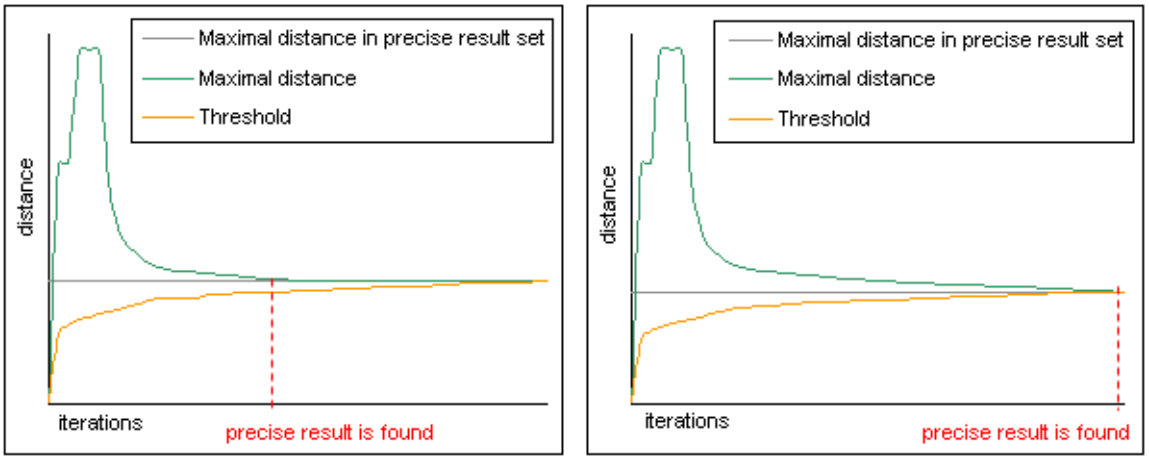

Figure 4.7: Maximal distance and threshold behavior

$$LQ = \frac{d_A^{max}}{d_E^{max}} - 1$$

The ratio is decreased by 1 to have the LQ of the precise result equal to 0. This measure only compares the worst objects in the approximate and precise result sets. The following measures work with all k resulting objects. The advantage of this measure is that it can be easily upper-bounded in each iteration of the TA.

4.4.2 Precision and Recall

Precision and *recall* are two standard measures used in Information Retrieval. Precision measures the ratio of the qualifying retrieved objects to the total of retrieved objects. Recall compares the retrieved qualifying objects with the total of qualifying objects. In our case, we consider objects from the exact result set to be the qualifying ones. The precision P and recall R can be defined as:

$$P = \frac{|R_E| \cap |R_A|}{|R_A|} \quad \text{and} \quad R = \frac{|R_E| \cap |R_A|}{|R_E|}$$

These measures are intuitive and easy to compute. However, both of them give exactly the same information in our application since the sizes of R_A and R_E are the same. Thus only recall will be used in our experiments. However, note that recall does not consider ranking in result sets or the distances of objects. The recall is the same when ten best objects from R_E are missing in R_A as well as when ten last objects are missing. Similarly, the recall

is the same whenever the distances of some objects differ by 10 or 1000. However, in both cases we see that the first result is better. Other measures are needed to cover this.

4.4.3 Relative Error on Distances

This measure compares the distances of objects in R_A to the distances of objects in R_E. Let o_A^i be the i^{th} object found by the approximate algorithm and o_E^i the i^{th} object found by the precise algorithm. Then the *relative error on distances* for this object, RE^i, is defined as

$$RE^i = \frac{d(o_A^i, Q)}{d(o_E^i, Q)} - 1$$

For the whole result set, the relative error can be computed as

$$RE = \sum_{i=1}^{k} RE^i = \sum_{i=1}^{k} \frac{d(o_A^i, Q)}{d(o_E^i, Q)} - 1$$

This measure does not take into account the actual distribution of distances in the dataset. Suppose that there is an object closer to the query object and all the other objects from the dataset are much further, but near to each other. Then the RE will be high if the first object is missing in the approximate result. On the other hand, if the first object is reported but some others are missing the RE will be lower. It is therefore useful to combine this measure with the recall. If both measures are high we know that some good object has been missed. On the other hand, when RE is low we can be satisfied with our result even if the recall is lower because despite the fact that we haven't found the precise result, we have very similar objects in the answer.

4.4.4 Error on the Position

Another possible way of the approximate similarity search accuracy assesing is the comparison of objects' positions in the ordered result lists R_A and R_E. It is based on the *Sperman footrule distance* used for computing the difference of two sorted lists. Let S_1 and S_2 be two sorted lists containing elements from the same set X, and $S_i(o)$ be the position of object $o$in the list S_i. The Sperman footrule distance of S_1 and S_2 is defined as

$$SFD = \sum_{i=1}^{|X|} |S_1(o_i) - S_2(o_i)|$$

For the approximate search results a slight modification of this measure is used. We only compare the (ordered) approximate result to a list S_E of all objects from the dataset sorted by their distance from the query object. As the position of any object from R_A can only be

higher than or equal to its position in S_E, the absolute value operator can be omitted. The sum is normalized by the size of the approximate result set so that we get the same position error when the first object is missing in the result with 10 and 20 objects. We compute the *error on the position*, EP, as

$$EP = \frac{\sum_{i=1}^{|S_A|} S_E(o_i) - S_A(o_i)}{|S_A|}$$

Position error values are high if far objects are found instead of the near ones, which is also reflected by the aforementioned measures. However, for results with similar precision and relative error or distances, EP can penalize results with errors on the first positions (considered more important) from errors on further positions.

This measure is quite difficult to use since it requires the sorted list of all elements from the dataset (with respect to a given query object q). In large data collections it requires enormous amounts of time to obtain such lists. Fortunately, it is in fact not necessary to supply the entire lists; we only need a sorted list of all objects with distance lower or equal to d_A^{max} to find the difference between precise and approximate positions of all objects in the result set. Such list can be obtained in the MUFIN from a range query $R(q, d_A^{max})$ over the overlay with precomputed overall distances.

4.4.5 Improvement in Efficiency

All the previous measures have dealt with accuracy. However, we are using the approximate searching to gain efficiency, so we need a measure for efficiency as well. The *improvement in efficiency*, IE, is expressed as the costs ratio of the precise and the approximate algorithm:

$$IE = \frac{cost_E}{cost_A}$$

There are more strategies for measuring algorithm costs, we can consider the number of distance computations, disk accesses or messages sent through the network. However, the most interesting for users is the total time needed for query processing, so we will use it as our cost measure.

Unfortunately, to run the precise algorithm in order to get the precise times for comparison is very costly for a large set of experiments. To get the precise result sets, we use the precomputed distance layer with a single-metric query, which works in a different way and cannot be compared in costs to a complex query. Therefore, we can only watch the approximation costs in our experiments, and estimate the costs of the precise searching with the help of quality measures. From the ratio between the costs and the quality of a given query we can guess what time would be needed to find the precise answer.

4.5 Results Quality Estimations

In the previous section we have described various measures of quality that are used to evaluate average precision of a given approximation method by comparing its results to the exact ones. For example, we can find that an approximate algorithm gives $80\,\%$ recall on average. However, the recall can be very different for individual queries, being very high for some but rather poor for others. Of course, information about the quality of each query result is of more value to the users than just the average values. Knowing how precise the results are the user can decide whether he/she is satisfied or whether he/she wants to issue additional queries, perhaps choosing more precise (but slower) searching. Also the system itself can exploit the knowledge about the results quality to distribute the resources more effectively.

The problem is that a real-time information about the results quality cannot be obtained since to measure the quality we need to know the exact result. Any real-time quality reports can therefore only provide some upper/lower bounds (based on some knowledge of the algorithm properties) or estimates (supported by observations on the usual behavior of query processing).

In the following, we will discuss both the bounds and the estimates that can be used to gain some information about the results quality at the time of query evaluation. The reason why the bounds are insufficient is simple: bounds are always correct but often quite far from the real values. Therefore we would like to find some way of predicting the quality that would give better estimation with high probability. The proposed estimation method will be tested in experiments.

4.5.1 Bounds on Quality

When we look on the Threshold Algorithm more closely we find that certain theoretical limits exist for the approximate results quality. In each step the algorithm computes current threshold value which is provided by monotonous aggregation function. As the arguments for this function are supplied in increasing order (from sorted lists of each feature) also the threshold value in each iteration has to be equal to or greater than the previous ones. The same reasoning shows that all future values of the threshold must be at least equal to the last computed one. At the same time the threshold value t implies that no object with overall distance lower than t can be found in the future iterations. Therefore, in the last iteration of the approximate algorithm with threshold t_A^{last} we know that all objects in the result set with overall distance lower than or equal to t_A^{last} belong also to the precise result set.

Exploiting this knowledge we can define the *lower bound on recall*, R_{bound}, as a ratio of objects from the approximate result set with distance at most t_A^{last} to the total of objects in the result set:

$$R_{bound} = \frac{|\{o \in R_A | d(q,o) \leq t_A^{last}\}|}{|R_A|}$$

Similarly, the LQ measure can be upper-bounded by the original Fagin's θ parameter. d_E^{max} is always higher than or equal to t_A^{last} unless t_A^{last} is higher than d_A^{max} in which case the precise result was found because the TA has stopped and $d_A^{max} = d_E^{max}$.

$$LQ_{bound} = \frac{d_A^{max}}{min(t_A^{last}, d_A^{max})} - 1$$

Both these bounds can be easily computed in each TA iteration, enabling us to watch the progress of the results quality. Visualization of bounds is provided in Figure 4.8, colored areas show which objects from the current result set are bound to be in the exact result (orange background) and which are in the exact result but this is not known in current iteration (gray background).

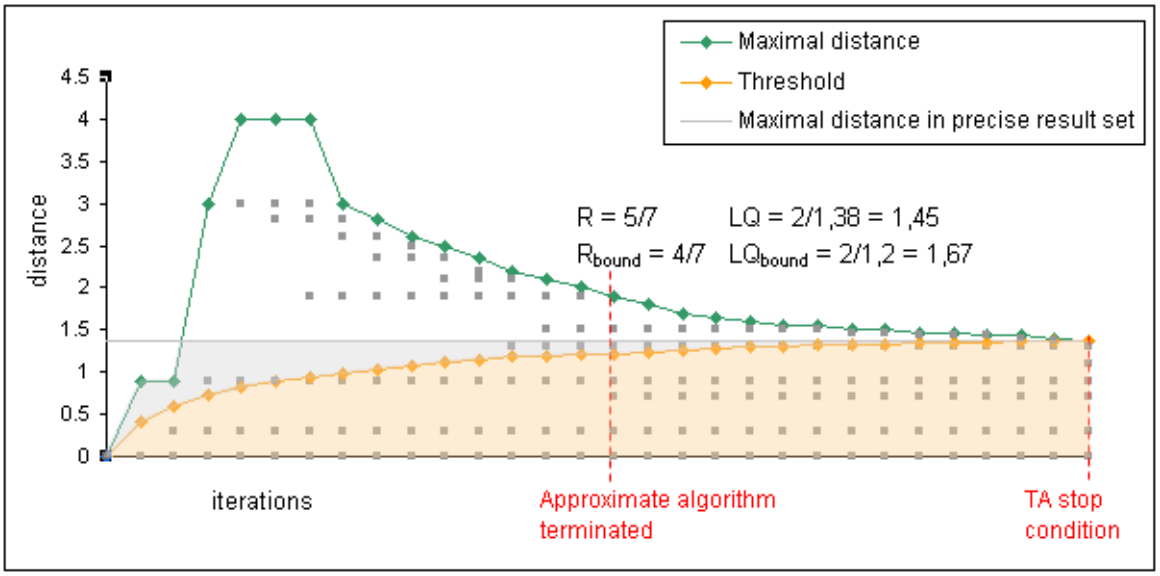

Figure 4.8: Quality bounds

4.5.2 Estimations

Average evaluation of results gives only limited information about the quality of a particular query result. Bounds which can be defined for some quality measures are better because they are computed with respect to the specific query. However, the information about the approximate results accuracy is still insufficient. We believe that better estimations of quality can be obtained by predicting the algorithm behavior in future iterations (i.e. those iterations that are performed by the standard TA but have been cut off by the approximation). More specifically, we are interested in the development of d^{max} and t^{last} values which can be viewed as functions of iteration i. We know their values up to a certain iteration i^{last}

where the approximate algorithm is terminated, and we want to guess their behavior in the next iterations until they intersect (see Figure 4.9). We are also familiar with their general behavior (t^{last} is non-decreasing, d^{max} non-increasing after the first couple of iterations) and know they have to intersect. The intersection must appear after a finite number of iterations, in the worst case after all objects from the dataset have been seen; the object with the highest overall distance must have all descriptor values lower than or equal to the last values in the list, and thus its overall distance can be at most equal to the threshold value computed over the last values in the sorted lists. If the x and y coordinates of the intersection, i.e. the number of iterations needed to reach the intersection and the maximal distance in the precise result, can be guessed, we may use them to better estimate the quality or the time needed for obtaining an exact answer.

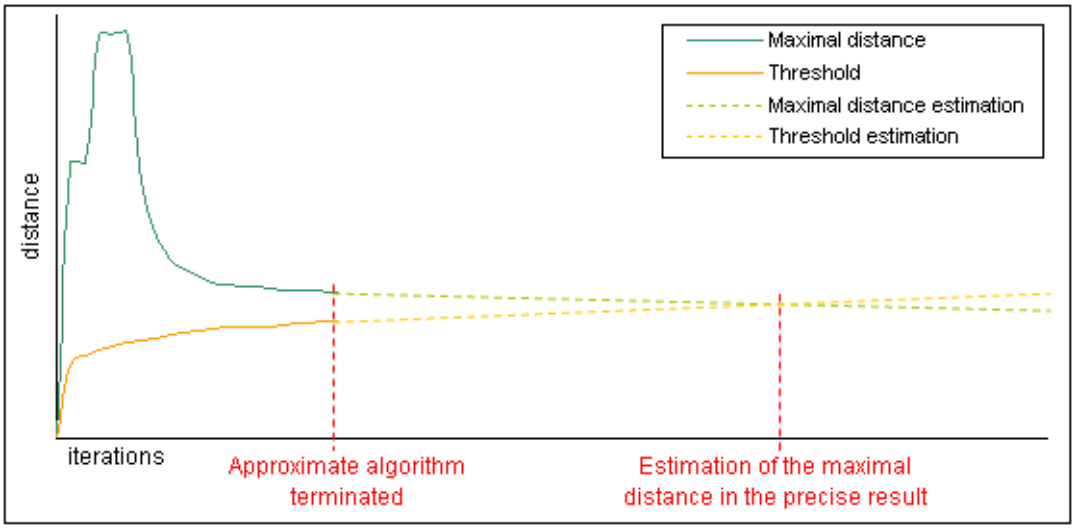

Figure 4.9: Maximal distance and threshold estimations

A possible way of determining t^{last} behavior is to compute threshold values for some selected iterations, e.g. $i^{last} + 100$, $i^{last} + 200$ etc., until the threshold value meets the maximal distance of the approximate result set. The computations are not time-consuming and resulting values enable us to follow the threshold curve development very precisely. The problem is that we cannot easily obtain the j^{th} object from the sorted lists for any j. We only use lists of predefined length $c.k$, and for $j > c.k$ finding the j^{th} object requires that a new query is executed. This could be very expensive as we do not know how many iterations will be needed for the threshold to intersect with the maximal distance. We would either need to do the sorted access for the whole dataset or repeatedly issue new queries.

However, we can try to estimate the quality without any additional accesses to the data. Using the values of the maximal distance and the threshold computed in the approximate TA iterations, we can extrapolate the two curves by standard numeric methods and then compute their intersection.

4.5.3 Extrapolation

Extrapolation is a process of constructing new data points outside a discrete set of known data points. Its results can be of variable quality depending on the choice of extrapolation method and the data used for extrapolation. Suitable extrapolation method should be chosen with respect to the known data points and general assumptions about their properties (assumed to be continuous, smooth, possibly periodic, etc.). Extrapolation methods range from simple ones such as linear extrapolation to more complex ones that are quite time-demanding.

For our problem we needed a simple method that can be easily used since we need quick responses. Fortunately, we know quite a lot about the curves we want to estimate, above all their general shape. Consultation with statistics experts (namely RNDr. Marie Budíková, Dr. and Mgr. Jiří Zelinka, Dr. from the Department of Mathematics and Statistics of MU Faculty of Science) confirmed that both curves can be extrapolated either as exponential or linear ones. Exponential curves express the shape more precisely and would probably give better results but are more difficult to compute. Therefore, linear curve was chosen as a base for the extrapolation, as it is simpler and in later iterations both curves draw to a line.

Now we have decided to use a linear function $f(i) = a.i + b$ to find the threshold values for iterations $i > c.k$ (the same applies to function $g(i) = a'.i + b'$ and the maximal distance). To identify the most suitable parameters a and b, the values of t_i (or d_i^{max}) computed in iterations $i \leq c.q$ can be used. However, we do not employ all of them as the curves are strongly non-linear at the beginning. Whenever the values are computed in an iteration they are compared to the values from the previous iterations; if the difference between the curve tangent from iteration i and several previous iterations is sufficiently small, we consider the curve linear and from this iteration on we use the values to find the parameters for function $f(i)$. This is done by the *least squares* statistical method [4]. Having the data pairs (i, t_i), we find a and b such that the function values $f(i)$ best fit the data. Formally, we minimize the functions

$$\sum_{i=1}^{c.k}(a.i + b - t_i)^2 \quad \text{and} \quad \sum_{i=1}^{c.k}(a'.i + b' - d_i^{max})^2$$

Differentiation is used to find the minimum, and the parameters a, b, a' and b' are computed from the resulting linear equations. The intersection of $f(i)$ and $g(i)$ is used as the estimation d_{estim}^{max} of the maximal distance in the exact result set.

Exploiting the value d_{estim}^{max} we can estimate the recall, R_{estim}, and the loss of quality, LQ_{estim}, using the same reasoning that we gave for R_{bound} and LQ_{bound} respectively:

$$R_{estim} = \frac{|\{o \in R_A | d(q,o) \leq d_{estim}^{max}\}|}{|R_A|} \qquad LQ_{estim} = \frac{d_A^{max}}{d_{estim}^{max}} - 1$$

4.5.4 Estimation Error

In the previous section, we have described a method for estimation of the development of two curves, t^{last} and d^{max}. We use their intersection to simulate the maximal distance (and at the same time the last threshold) in the precise result set and, exploiting this value, we estimate the quality of the approximation. The accuracy of our estimation therefore depends upon the accuracy of the extrapolation, which can be measured by the distance between the curves intersection and the real maximal distance in the precise result. The distance can be normalized by the maximal distance. Consequently, we define the *estimation error*, EE, as follows:

$$EE = \frac{d_E^{max} - d_{estim}^{max}}{d_{estim}^{max}}$$

A positive (or negative) EE value signifies that the estimation d_{estim}^{max} was higher (lower) than the real value d_E^{max}.

Chapter 5

Experiments

In the area of similarity searching, the data domains are often so large and variable that only very general algorithms can be proved to work for all of them with a guaranteed quality, usually at the cost of a low efficiency. If we want to create a system that processes large data efficiently we often have to fit the search methods to the specific data collection, i.e. to design them in such a way that for most of the data objects good results are found quickly. Worse results or longer response times are tolerated for less typical data objects in return for good performance in most cases.

In the previous chapter, we have proposed an approximation algorithm for combined queries in an image searching system. To obtain an efficient searching tool for this system, we need to set the algorithm's parameter c in such a way that it will work well for majority of images in the system (which are various and numerous enough to represent any general database of images). We aim to determine the suitable parameter through a series of experiments on randomly chosen objects and then test the resulting algorithm on a different set of objects. We also aim to examine the usability of the proposed estimations of results quality.

5.1 Experiment Settings

Our aim is to find such a value of c that the approximate TA using $c.k$ sorted accesses will, with high probability, find good results for the top-k complex queries with the query object q. To achieve this, it is useful to consider which queries are the most probable to appear, and to fit c to them in the first place. We can hardly say which images are more probable to be submitted to queries; however, we can make some guesses about the number k of objects required in the result set. It is probable that for similarity searching the user will not ask for a single object but will prefer to have more similar ones so that he/she can choose from them. On the other hand, a result set of 500 objects is too large for browsing through. The user will rather ask for fewer objects, select the best of them and eventually issue a new query. An optimal size of a result set that can be easily displayed on the screen is 5 to 100 previews.

Exploiting these observations, we have chosen the values of k for the experiments. Most of them have been taken from the optimal scope but several less probable result set sizes have been used as well to see how the algorithm behaves for them (see Table 5.1 for details).

Next, we needed to choose the values of c to be tested. Here, we wanted to select such settings that are likely to give good results in an acceptable time. The lower bound on c is provided by the fact that sorted accesses are the only way we can acquire objects, and at

Parameter Name	Parameter Values
k	1, 2, 5, 7, 10, 15, 20, 30, 50, 75, 100, 150
c	1, 2, 5, 10, 15, 20

Table 5.1: Parameter settings for experiments

least k objects are needed for the answer. We have issued the nearest neighbor query for $c.k$ objects to each descriptor overlay; since we have used five overlays, we could get up to $5.c.k$ objects. However, as some objects may appear repeatedly in the responses from several overlays, we needed c to be at least 1 to be sure that at least k objects would be retrieved by the sorted accesses. On the other hand, $20.k$ sorted accesses provide up to $100.k$ objects which should be enough to give good results. Moreover, it this case the computations may already be quite time-demanding for higher values of k. We have run the experiments for six values of c between 1 and 20 to find the optimal value.

We have not tried different settings of aggregation function in our experiments. The function we have used (see Section 3.3.3 for details) was chosen by experts; to find other aggregations suitable for image searching would require further study which was not part of this thesis. However, we intend to run experiments with different aggregation functions in future research.

The experiments have been run on a network described in Section 3.3.4, holding 2 million image objects. 50 objects were randomly picked from a larger dataset of 10 million images and for each of them a top-k query was run with $c.k$ sorted accesses for each combination of the parameters c, k from Table 5.1. The query results as well as the values of the threshold t_i and the maximal distance d_i^{max} in each TA iteration were stored for quality evaluation and estimations computing.

5.2 Evaluation of Results

For each of the 3.600 experiments, we have compared the results to the precise result sets and evaluated all the quality measures presented in Section 4.4. The precise result sets were obtained through a range query to the precomputed overall-distance layer (see Section 4.4 for details) for each query object. The range for each object was chosen as the maximal distance of all objects found in any of the 72 approximate queries executed for this object. Thus the precise result sets contain at least as many objects as the largest approximate result sets, and for all objects found by the approximation we can find out how many objects with lower distance from the query object exist. For individual experiments evaluation, we have used subsets of these precise result sets with the adequate number of objects (i.e. for queries with k = 5 we compared the results with the first 5 objects in the precise result set) with the exception of error on positions where the whole precise sets were needed.

Since we are more interested in the general behavior of our algorithm than in the individual cases, we evaluated the approximation quality measures for given settings of c and

k as the average of the values computed for individual queries with the same setting. For each of the chosen measures we constructed a graph showing the dependence between the respective measure values and the approximation parameter c for all values of k under examination. In addition, average values have been computed from all experiments for a given c, and depicted in a small embedded graph. In the graphs, we distinguish between the more and less typical values of k – the boundary values are represented by dashed lines. Similarly, in the small graphs we can find two curves, the solid line representing the average computed without the boundary values, and the dashed line representing the overall average.

5.2.1 Loss of Quality

The LQ measure describes the ratio between the maximal distances in the approximate and precise result sets (see Section 4.4.1 for exact definition). Evaluation of results is depicted in Figure 5.1 (note that the scale of the main graph is logarithmic, the scale in the embedded graph). We can observe that the worst values have been achieved for k = 1 and k = 2. This shows that the objects with low overall distances from the query object are often not in the first places in the sorted lists for individual descriptors. However, if the very best object has overall distance 0, it has to be first in all the lists since the overall distance is computed as a sum of values for individual descriptors with positive coefficients. From our set of query objects, only 6 out of 50 were present in the searched data, therefore the best result was not always found for k = 1.

With the growing number k of objects in the result set the chance of finding good results among objects retrieved by the $c.k$ sorted accesses increases. This indicates that although the good objects are not necessarily in the first places of the sorted lists, they tend to appear quite soon.

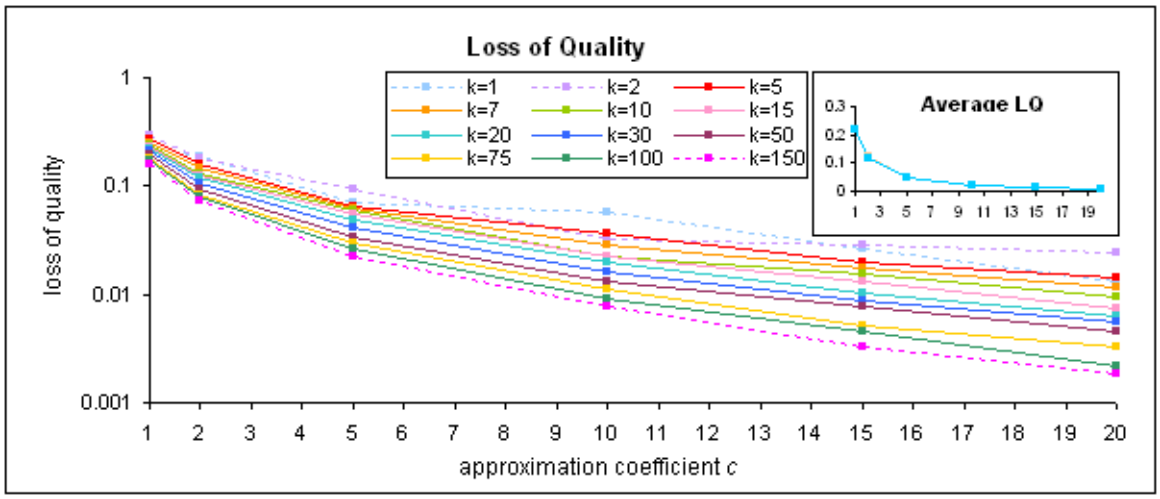

Figure 5.1: Loss of quality

5.2.2 Recall

Recall R is defined as the proportion of objects from the precise result that are found by approximate searching. We can see that about $30\,\%$ of the precise result is found even for c = 1. Specifically, in the case of the point query (i.e. query with k = 1) with 1 sorted access the recall is $30\,\%$. In each sorted access up to five different objects are retrieved from the five descriptor overlays. We know that in $12\,\%$ of our experimental queries the query object itself was sent from each overlay. In the other $18\,\%$ of experiments, the best match was found among the five objects, i.e. the best object was on the first position in at least one sorted list.

For the maximal value of c, most of the queries give recall of $90\,\%$ and better. Worse results are only retrieved by queries with small response sets, where the number of objects examined by the algorithm is far lower – as we have already observed, good objects tend to appear soon, but not immediately in the beginning of the sorted lists. We can also see that the recall rises quickly at the beginning but for large values of c the improvements are very small so it would require many more sorted accesses to reach the $100\,\%$ recall, i.e. to find the precise result.

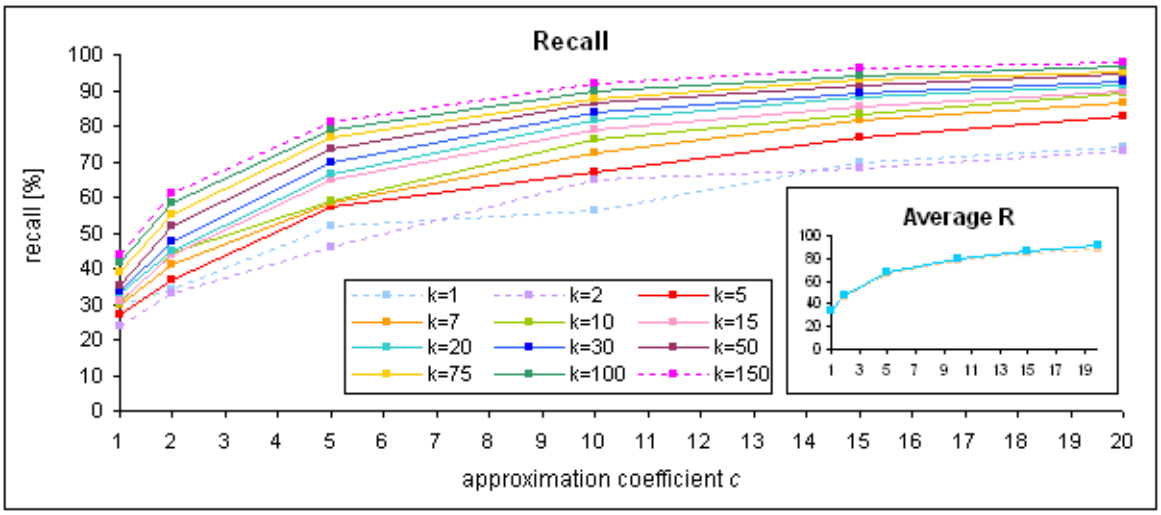

Figure 5.2: Recall

5.2.3 Relative Error on Distances

The RE measure is similar to the LQ one but compares distances of all objects from the approximate and precise result sets whereas LQ compares only the maximal distances from each result set. The graph in Figure 5.3 (using the logarithmic scale again) shows that the RE values are a little lower, which can be easily explained. Some of the distances used for the RE computation are the same in the precise and approximate responses since the same objects appear in both sets (see the Recall graph). This reduces the average error on distances. On

the other hand, LQ works with the worst objects which are always different in the precise and approximate results unless the recall is $100\,\%$. As an example, consider an approximate response set with 10 objects and $90\,\%$ recall. The only object that does not belong to the precise response has to be the worst object from the approximate result set. Let the object distances be 1, 2, ..., 10 for the precise result set, and 1, 2, ..., 9, 11 for the approximate one. Then LQ = 11/10 - 1 = 1/10 and RE = 56/55 - 1 = 1/55.

In the previous example, the disproportion between the RE and LQ values is much higher than in our experiments where the errors are distributed more uniformly in the whole response set. For k = 1 the RE and LQ values are naturally the same.

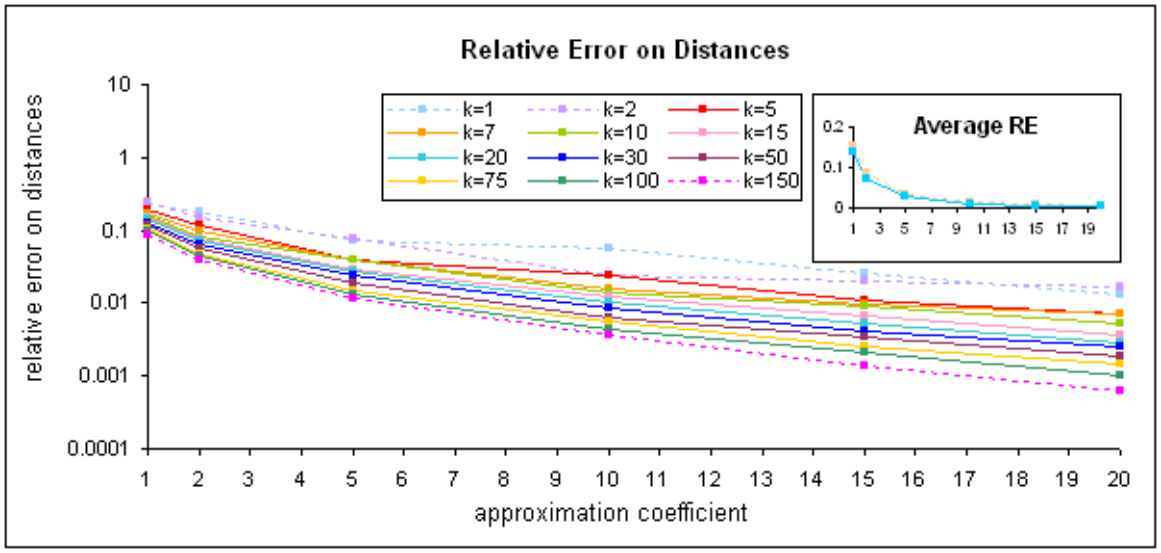

Figure 5.3: Relative error on distances

5.2.4 Error on the Position

The EP measure (depicted in Figure 5.4, also in logarithmic scale) expresses the average difference between the positions of a given object in the precise and approximate results. It is interesting to see that in terms of position, some results are really bad, especially for c = 1 where there exist several hundreds of objects better than the ones retrieved. In the special case of c = 1 and k = 1 we can even observe that the curve rises above the graph logaritmic scale, the error value being 1.255. This is caused by a single query which returned 57941th object instead of the first one. Still, from the previous measures we know that the retrieved objects are not substantially different from the precise ones in terms of similarity. In other words, the metric space is so dense that we can afford to miss several hundreds of objects without any great effects on the results quality. This is very advantageous for the approximate searching.

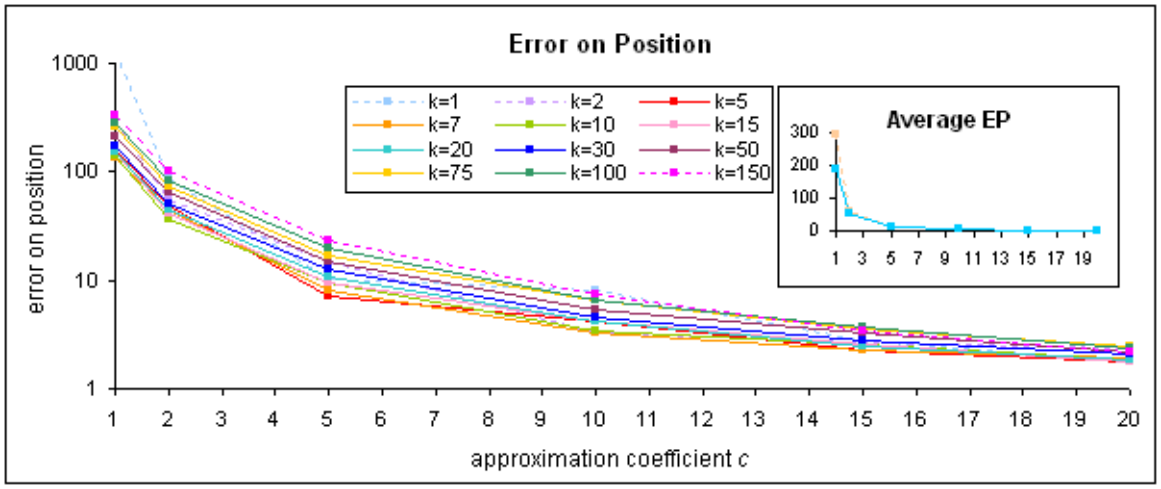

Figure 5.4: Error on the position

5.2.5 Search Costs

As explained in Section 4.4.5, we cannot compare the approximate search costs to the costs of the precise searching because these cannot be easily obtained. Therefore, we only watch the development of the cost curves for individual experiments settings. As anticipated, the approximate algorithm costs grow with the number of objects to be examined. The more objects are visited, the better is the quality of results but the more time is needed.

We can very roughly estimate the improvement in efficiency, i.e. the ratio between the precise and approximate algorithm costs, from the costs and recall graphs: the average recall (without boundary values) for $c = 15$ is $87\,\%$, for $c = 20$ it is $91\,\%$. If the recall curve continued to grow with the same speed, it would reach $100\,\%$ for $c = 54$. The curve would probably grow slower so $54.k$ is a lower bound on the number of the sorted accesses needed to get $100\,\%$ recall. From the costs graph we can estimate that with $c = 54$ the time needed for the approximation would be about 13 seconds. Thus the efficiency might be about 11 times better for $c = 1$, 4.3 times better for $c = 10$ and 2.5 times better for $c = 20$.

The comparison between the costs of the sorted and random accesses shows that, in general, more time is needed for the random accesses and the costs grow more steeply than in the case of sorted accesses. When we look on the costs more closely for specific values of k (as depicted in Figure 5.6) we see how the proportion of time consumed by the sorted and random accesses changes with the growing values of c. Detailed costs graphs for all values of k can be found in Appendix A.

5.2.6 Results Quality Conclusion

All the measures confirm that our hypothesis about the dependence between the number of the sorted accesses and results quality was correct. We can also see that in all graphs,

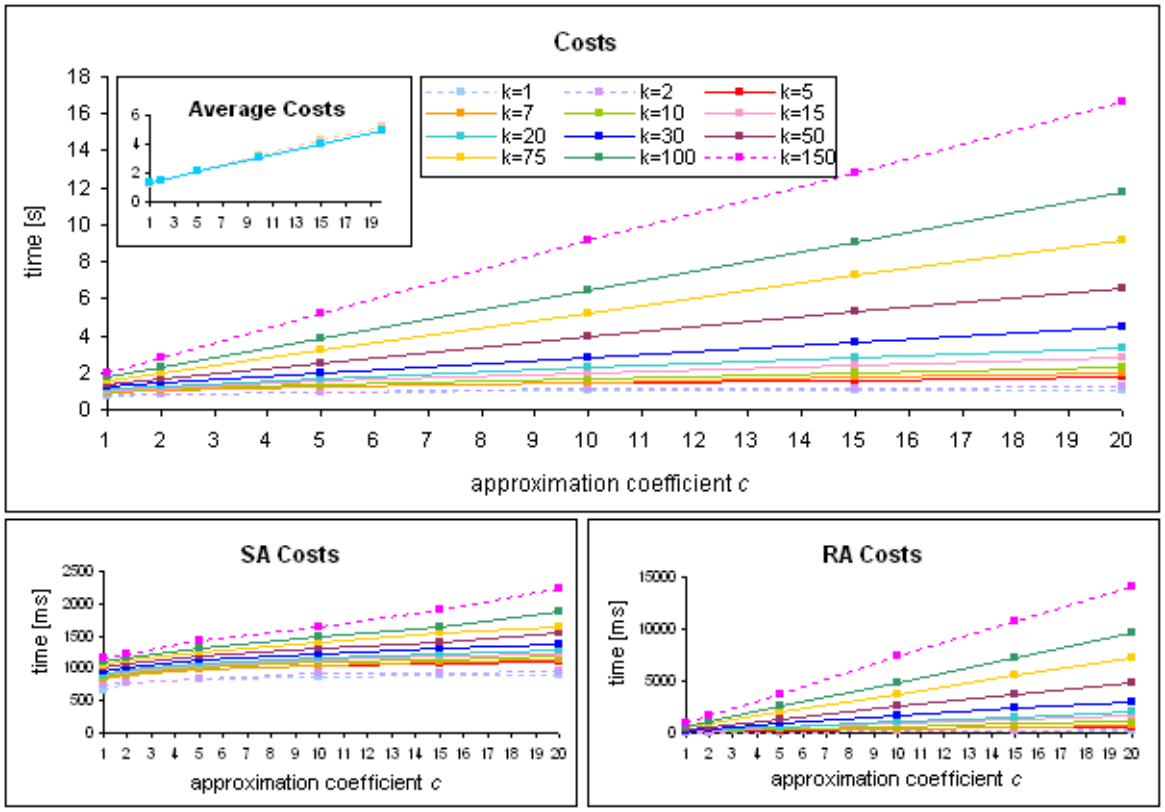

Figure 5.5: The total response time is the sum of sorted accesses cost, random accesses cost and time needed for processing of sorted access results (this is insignificant)

the quality rises more steeply for lower values of c, later the progress is not so fast. The breakpoint moves between $c = 5$ and $c = 10$. On the other hand, costs rise linearly. Thus it is more advantageous to use $10.k$ sorted accesses than $20.k$ because in the former case we gain more quality for the same costs. Altogether, $c = 10$ seems to be the best choice of approximation parameter. Using this, we can get query response in about 5 seconds, which is not suitable for highly-interactive applications but acceptable for browsing, and several times better than in the case of precise searching. Average result set will provide $80\,\%$ recall with low relative error on distances, i.e. $80\,\%$ of the objects from the precise answer, and $20\,\%$ very similar to the precise objects.

5.3 Estimations Evaluation

As we have discussed in the previous chapter, it is very desirable to complete the approximate result with the information about its quality. Since it is not possible to evaluate the quality precisely we can try to estimate it using the knowledge about the typical TA be-

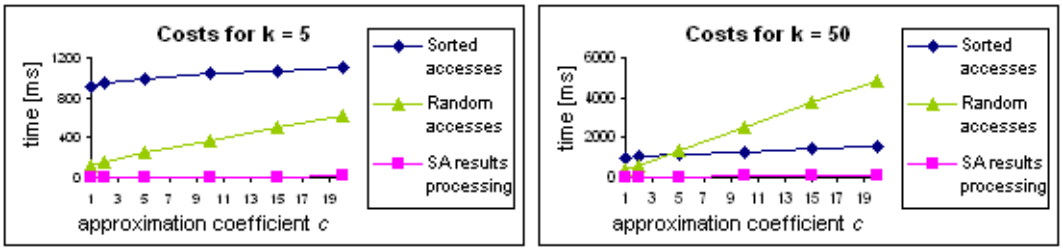

Figure 5.6: Approximation costs for two different settings of k

havior. To test the usability of the proposed extrapolation method we applied it on the t_i and d_i^{max} values acquired from our experiments, computed the intersection d_{estim}^{max} and compared it with the real value d_E^{max} in the precise result. In addition to the estimation error we also evaluated estimations of two quality measures, the loss of quality and the recall, and compared the results with the exact values and bounds defined for these measures.

5.3.1 Estimation Error

The estimation error describes how much the estimation missed the real value (see Section 4.5.4 for details). In Figure 5.7, estimation errors are depicted for several values of k; the whole set of graphs for all experiments can be found in Appendix A.

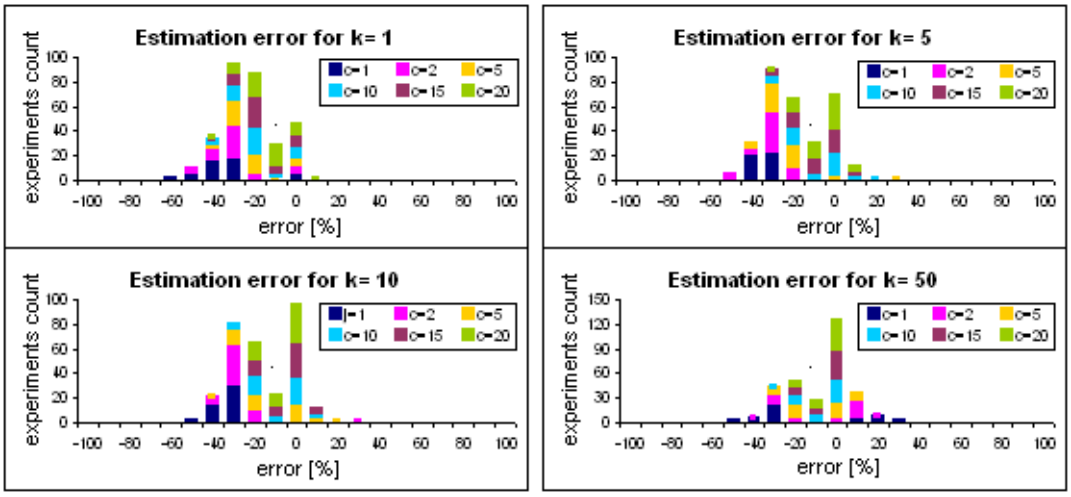

Figure 5.7: Estimations error

We can see that estimation is worse for low values of both k and c. This is natural since in this case we only know the values of the threshold and the maximal distance from a few iterations so the extrapolations perform poorly. The more iterations have been run, the better chance there is for good extrapolation and, consequently, good estimation. The good results

for higher values of c are also influenced by the fact that after the last TA iteration the difference between the threshold value and the maximal distance in the result set is small so the estimation cannot be seriously wrong.

5.3.2 Loss of Quality Estimations

The loss of quality estimations, together with the LQ bounds and the real LQ values computed in comparison with the precise results are depicted in Figure 5.8 for several values of k (more graphs can be found in Appendix A). The graphs show that with the growing number of the sorted accesses (and TA iterations) the difference between the real LQ and the bounds decreases. As for the estimations, we can see that better results are obtained for higher values of k and c where the intersection of the threshold and maximal distance curves can be estimated more accurately.

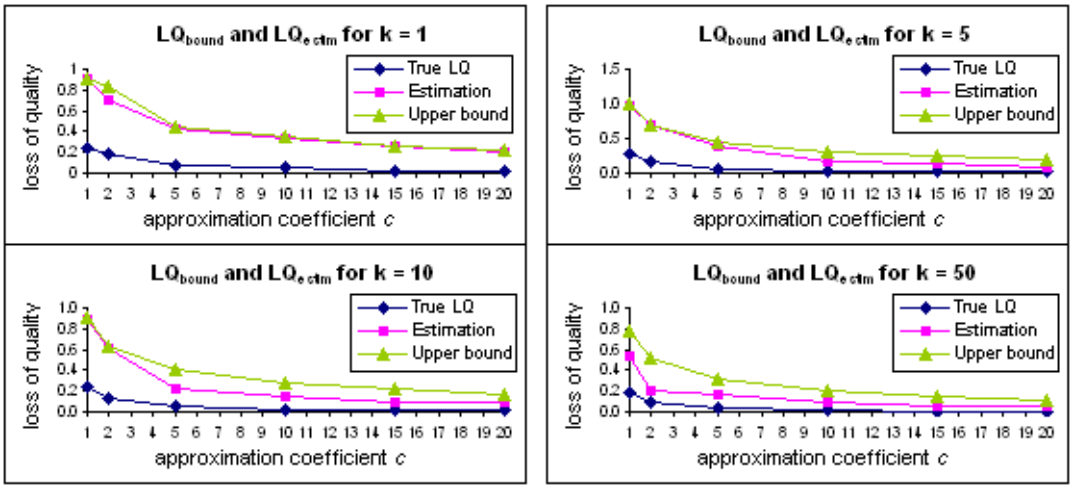

Figure 5.8: Loss of quality estimations and bounds compared to the real loss of quality

5.3.3 Recall Estimations

The evaluation of recall estimations (see Figure 5.9) confirms findings from the LQ estimations. Again, estimations work well if more sorted accesses have been used. However, estimations are more useful here since the bounds are quite far from the real values even for higher values of k. To compare the R estimations for all values of k, see Appendix A.

5.3.4 Estimations Conclusion

Even though the estimations are by far not exact, still they can give better information about the results quality than bounds, especially for higher values of k. As a future work, we will

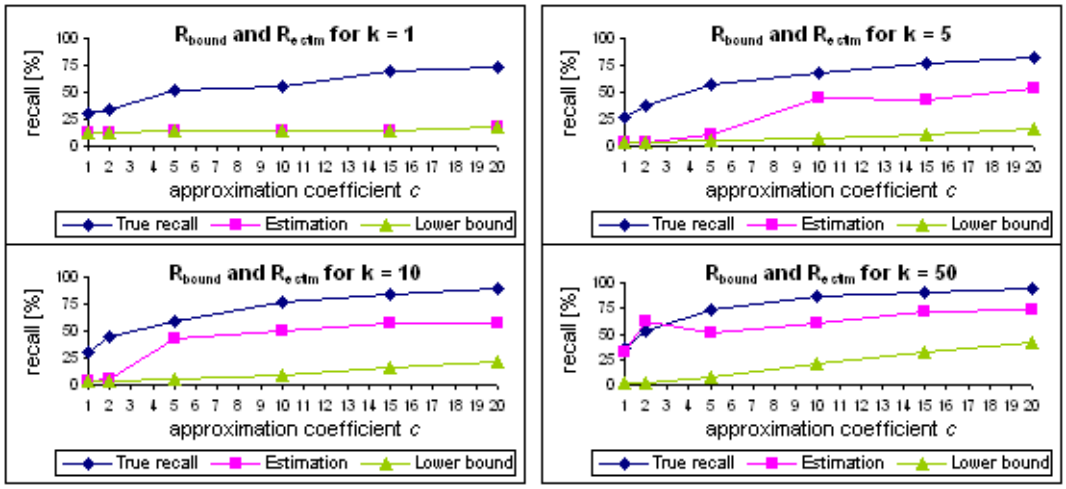

Figure 5.9: Recall estimations and bounds compared to the real recall

try to tune the estimations to give even better results, which may be affected by the values submitted to the extrapolation.

5.4 Results Verification

Based on the previously described set of experiments, we decided that the most suitable value for the approximate TA parameter c is 10 as it gave the best results for the chosen test queries. However, we needed to verify this hypothesis on another set of objects to find whether it is true in general, not only in the selected set of objects (called the *baseline set* in the following text). Therefore, we have randomly chosen another set of 50 objects, run all the experiments again and evaluated the results. For this *verification set*, all query objects have been chosen from the indexed collection.

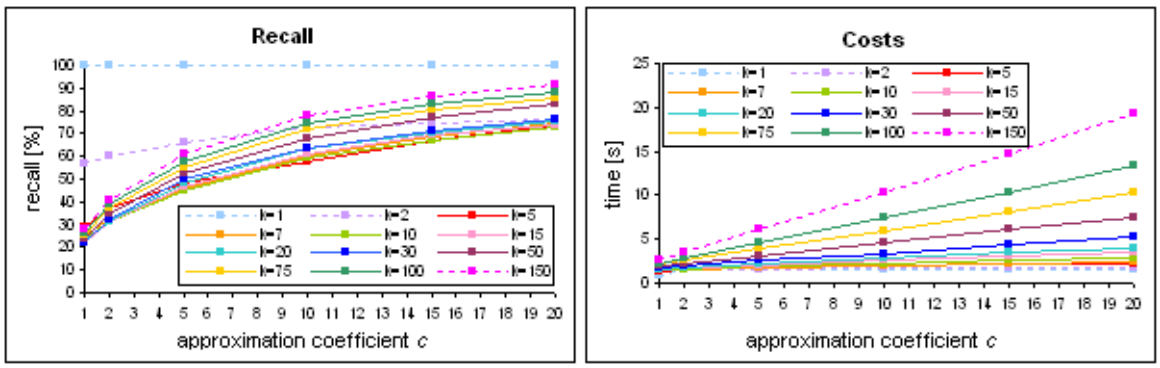

Figure 5.10: Quality and costs evaluation for the verification set of experiments

In Figure 5.10, we can see the evaluation of quality in terms of recall (the other measures give similar information as the baseline case and can be found on the enclosed DVD). The results are very similar to the baseline experiments, the only significant exception is the behavior of experiments with $k = 1$, where we find $100\,\%$ recall (and consequently zero errors). This is caused by the fact that all query objects belonged to the indexed collection and therefore had to be found in the first TA iteration. We can also observe that the recall is a bit lower than for the baseline experiments, which can be affected by the distribution of objects in metric space. However, the general behavior of approximate TA is the same – quality rises more steeply for lower values of c.

When we compare the costs (depicted in Figure 5.10) to the cost of the baseline experiments, we find that the second set of experiments took longer to process. Both sorted and random accesses needed more time, in case of sorted accesses the increase is more significant. Again, this may be affected by the distribution of objects in metric space, so that more peers need to be accessed and searched to find the $c.k$ objects for each of the sorted lists. Still, the costs rise linearly so we can use the same reasoning as for the baseline experiments and confirm that $10.k$ is the most suitable number of sorted accesses for reasonably effective and efficient searching.

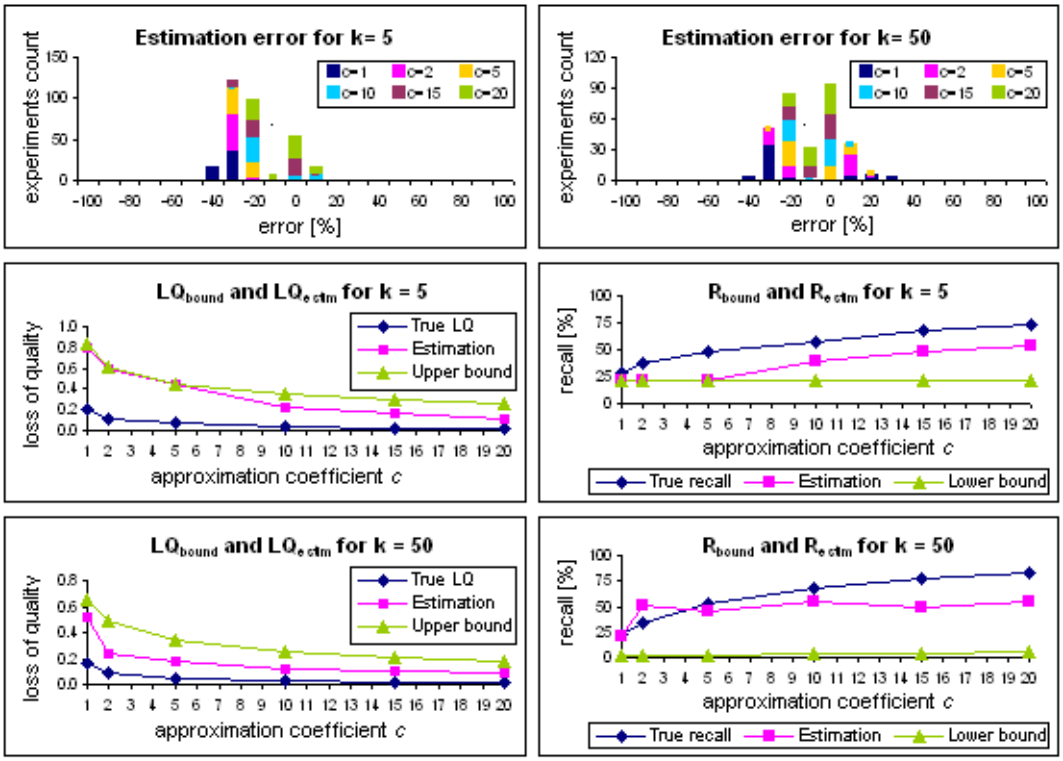

Figure 5.11: Estimations evaluation for the verification set of experiments

In the last set of graphs in Figure 5.11 we have evaluated the quality of estimations for the second set of experiments. The results are similar to the baseline experiments, again we can see that in most cases, estimations can provide better information about results quality than we would obtain with the use of the bounds. Naturally, also in this case the estimations are more accurate when more data is available for the extrapolations, i.e. when more sorted accesses have been allowed and more iterations run.

5.5 Summary

In the experiments, we have found that our approximate algorithm for complex similarity searching works well for the chosen data collection. We are able to reach $80\,\%$ recall in several seconds and the results are very good in terms of similarity compared to the precise results. Based on the experimental results, we propose to use $c = 10$ as the basic setting of approximate searching.

We are also able to compute estimations of the results quality at the query processing time. The estimations are not accurate but give useful information and will be subject to further research.

All the results were verified by an independent set of experiments which produced similar results. The entire statistics obtained from both sets of experiments as well as their visualization in graphs are available on the enclosed DVD.

Chapter 6

Conclusions

The first three chapters of the thesis provide an introduction to the topic of similarity searching and describe the theoretical background. In Chapter 1, we have discussed the need for new storing and searching techniques suitable for modern data and presented similarity searching in metric spaces as a promising solution. Chapter 2 covers the principles of similarity searching as well as the basics of distributed processing, which is necessary for large data collections effective management. Chapter 3 describes the multi-purpose similarity searching system MUFIN, for which we have designed an algorithm for effective processing of combined queries.

6.1 Realization of Objectives

The contributions of this thesis are embraced in Chapters 4 and 5. In Chapter 4, we have presented the baseline algorithm for combined queries, proved its correctness for our searching model and discussed its implementation. We have described advantages and disadvantages of each proposed method and one of them has been chosen as our prototype implementation. As the time costs of the precise algorithm have been too high, we have proposed an approximate algorithm for the complex queries. We have also presented measures for approximate results quality evaluation. Furthermore, we have discussed the need for real-time information about approximate results quality and propose a method for real-time quality estimations based on statistical properties of the algorithm behavior.

Both the proposed approximate algorithm and the method for quality estimations have been thoroughly tested with many different settings of search parameters. Chapter 5 embraces the evaluation of experimental results, in particular, detailed graphs and the descriptions of the quality measures. The quality estimations have been compared to the theoretical bounds and real values computed with the knowledge of precise results. All the experiments have been run twice for different sets of query objects to verify the outcomes.

Specifically, the objectives set for this thesis have been realized as follows:

- Several possible ways of combined query implementation have been discussed in Section 4.2 and the most promising one has been chosen. Furthermore, in Section 4.3 we have proposed a modification of baseline algorithm for approximate searching to obtain user-friendly response times.

- Quality measures for evaluation of approximate results have been presented in Section 4.4. In Section 5.1, we have designed a set of experiments to test our implementation of combined queries.
- We have evaluated and discussed the results of approximate search in Section 5.2.
- A method for real-time estimation of approximate results quality has been proposed in Section 4.5 as well as a measure of estimation accuracy.
- The estimation method has been tested and evaluated in Section 5.3.

Furthermore, both the proposed approximate algorithm and the estimation method as well as the results from experiments have been presented in a paper at SISAP'08 Workshop on Similarity Search and Applications [11].

6.2 Future Work Outline

In the course of work on complex queries implementation, many challenges have been recognized which could not be realized within the scope of this thesis but will be subject to further research:

- We would like to try the other TA implementations described in Section 4.2 and evaluate their efficiency.
- In the area of aggregation functions, we plan to run the experiments again with different weights of individual descriptors for the overall distance comptutations and compare the results to experiments described in this thesis. We would also like to try another types of aggregation functions than sum.
- A wide space for improvements is open in the area of estimations.
- To get a feedback from users, we intend to submit our implementation to extensive testing by a set of volunteers. This will enable us to find out more about the actual behavior of users in the course of searching – how many objects are usually required, how many times is the query repeated to get better results etc.
- We would also like to explore the possibilities of re-ranking the results once they are delivered to the user (as proposed in [5]) and automatic tuning of the approximation.

Appendix A

Detailed Graphs

In Chapter 5, we have presented the average values of quality measures which illustrate the dependence between the approximation coefficient c and the results quality regardless of the number k of objects in the result set. However, it is also interesting to see how the approximation behaves for different values of k, and to compare the effectiveness and efficiency of our algorithm in answering the queries. Therefore, we present several groups of detailed graphs in this appendix that enable comparison of the quality of the results and its estimations and the costs between the approximate combined kNN queries for all the values of k used in our experiments (as described in Section 5.1).

In each graph, we show the investigated measure depending upon the values of the parameter c. The same six values of c were used for all experiments so we can observe the effect of the additional sorted accesses on the results of queries for the particular k. Each graph presents average values computed from the results of 50 experiments using the same values of parameters c and k for different objects. Only the results for baseline experimental set are depicted here; the results for verification set can be found on the enclosed DVD.

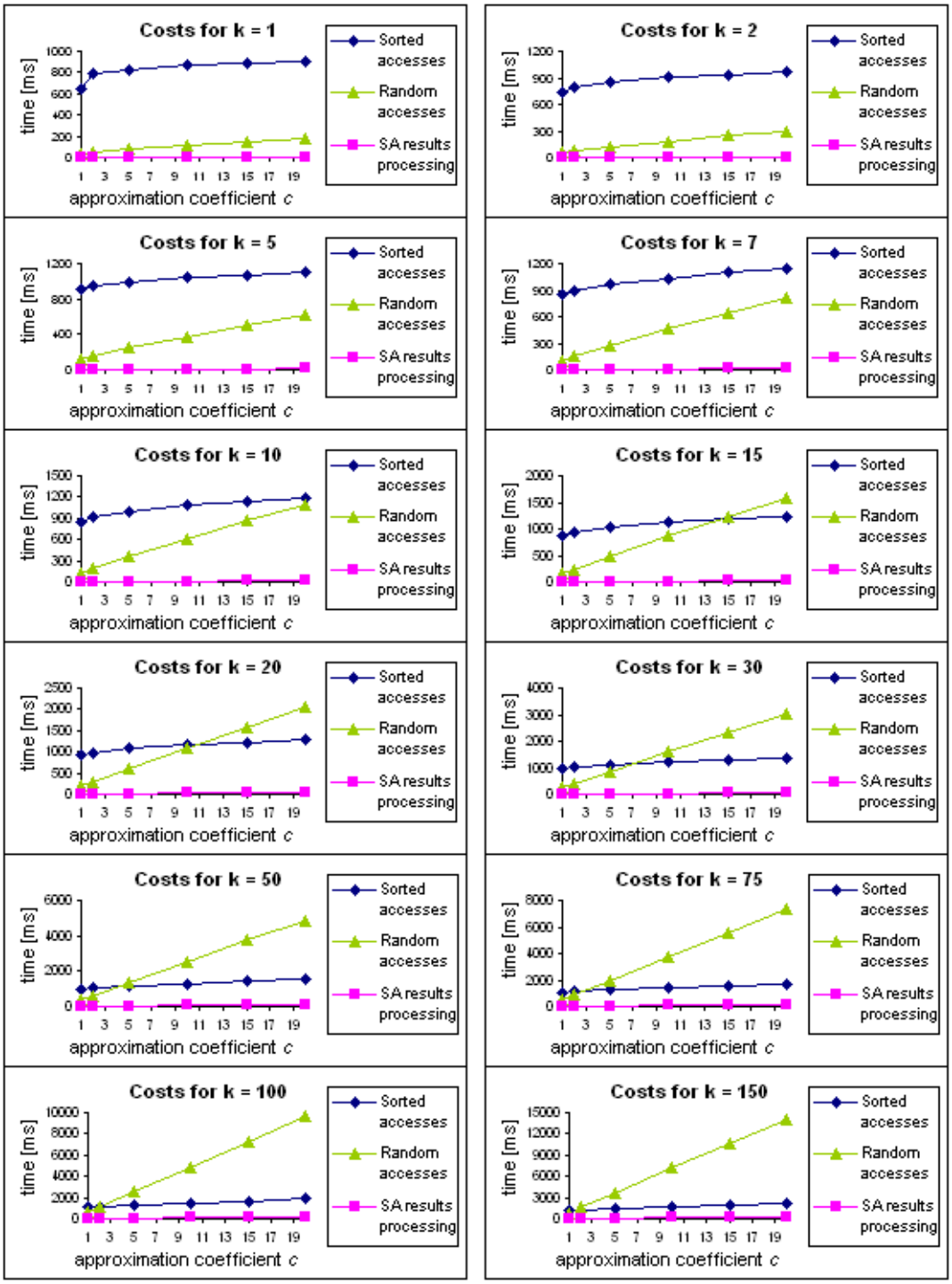

Figure A.1: Approximation costs for different values of k

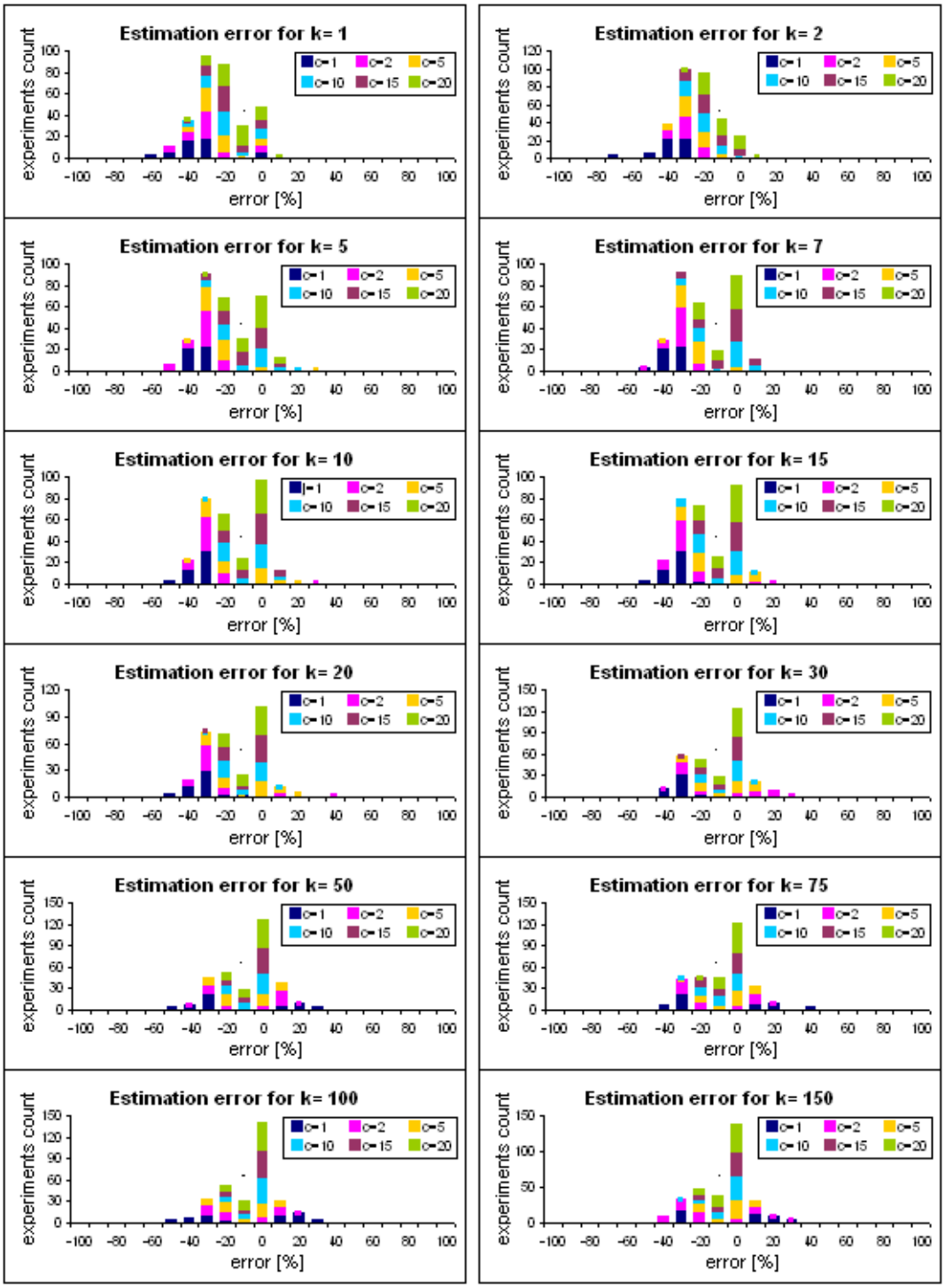

Figure A.2: Estimation errors for different values of k

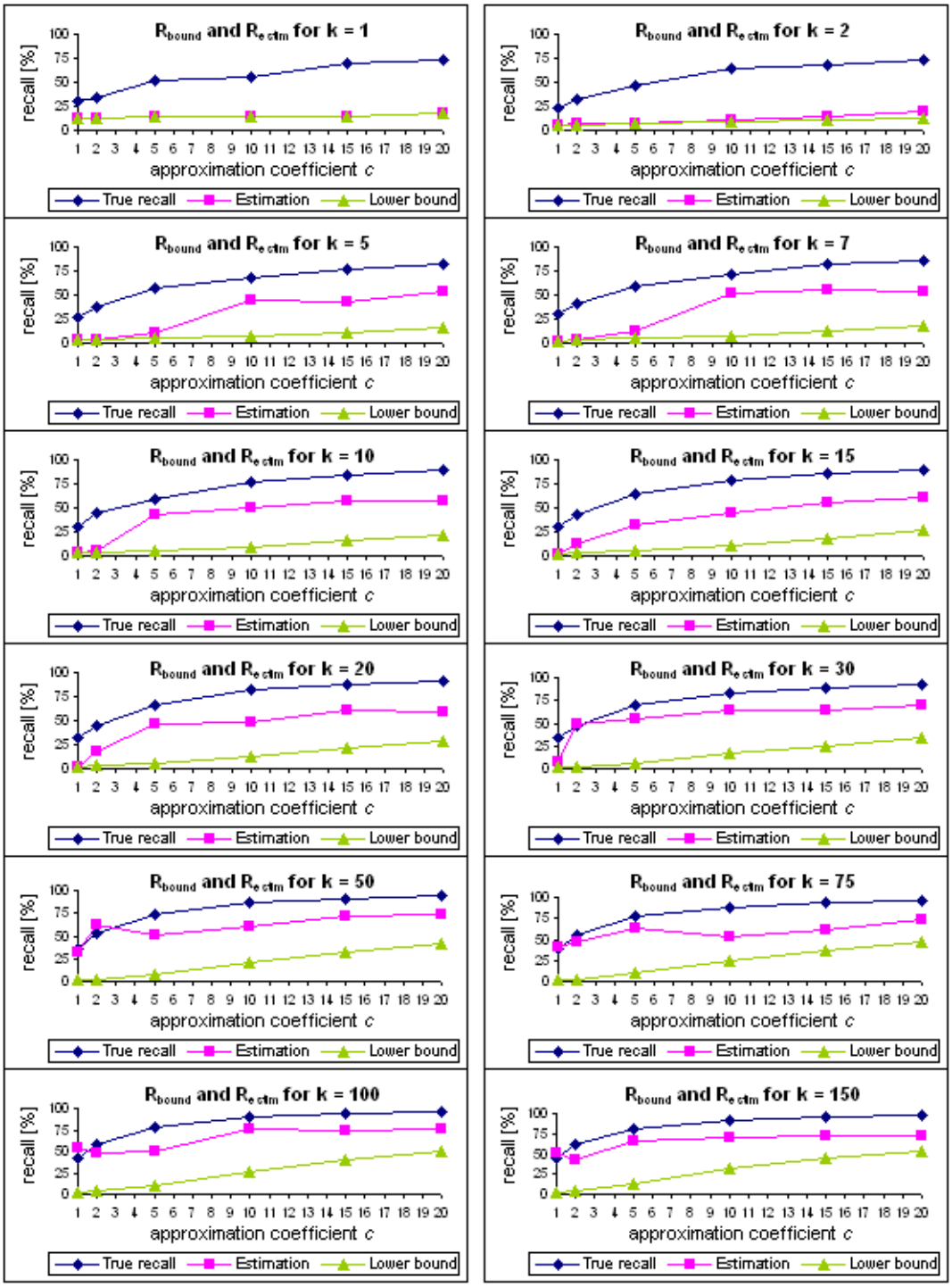

Figure A.3: Recall estimations and bounds compared to the real recall computed with the knowledge of the precise result

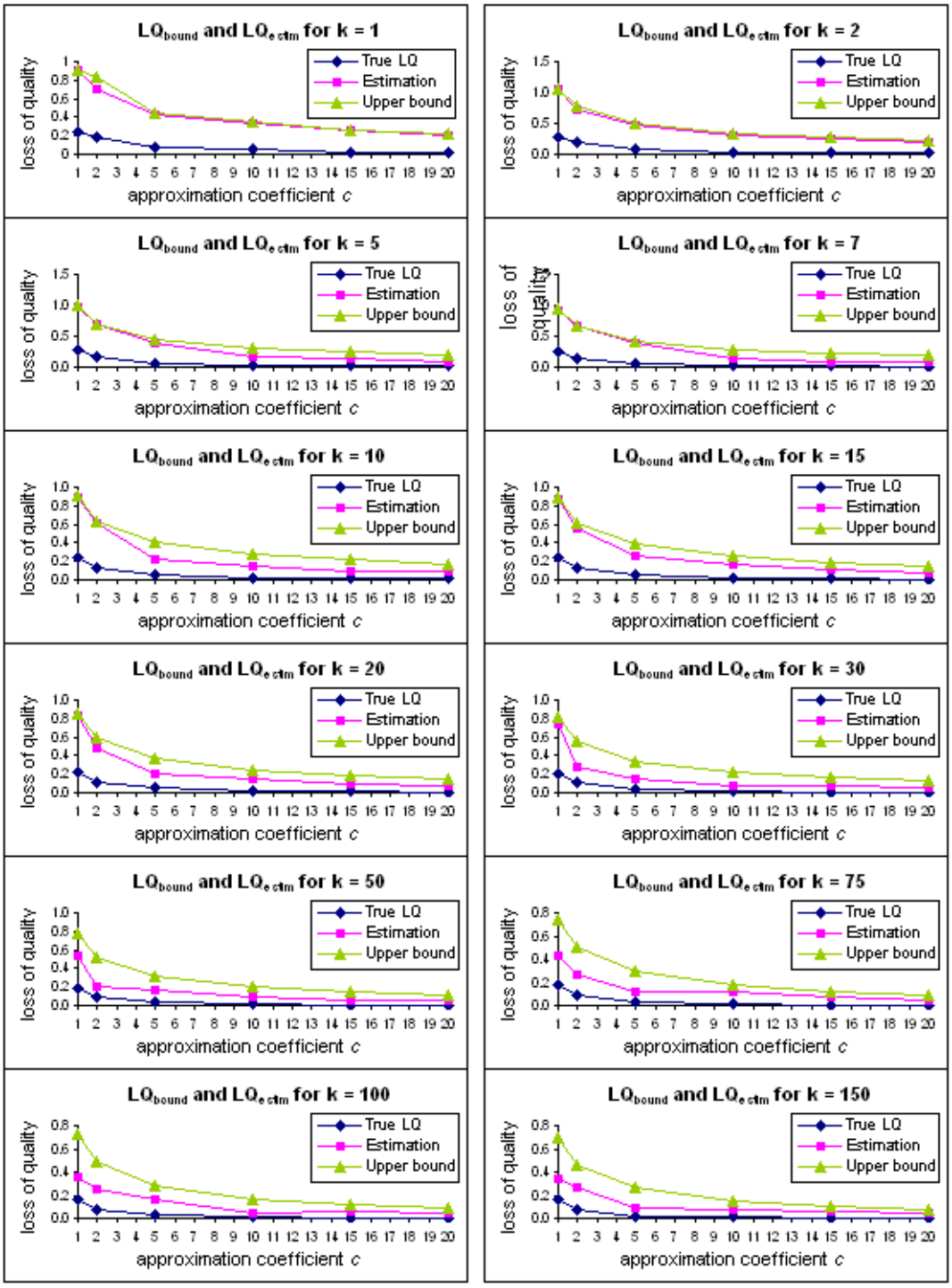

Figure A.4: Loss of quality estimations and bounds compared to the real loss of quality computed with the knowledge of the precise result

Appendix B

Contents of the Enclosed DVD

The enclosed DVD contains the following items:

- the text of this thesis in XML, TEX and PDF formats, and images presented in the thesis (directory `/thesis`);
- MUFIN implementation in Java (directory `/mufin`);
- Java classes for processing the data retrieved from experiments (directory `/utils`);
- results from the two sets of experiments (baseline and verification) as produced by MUFIN, a text file with results evaluation and quality estimations for each set of experiments, and graphs depicting the experimental results and various measures of their quality (directory `/results`).

Bibliography

[1] R. Fagin, A. Lotem and M. Naor: *Optimal aggregation algorithms for middleware*, in Proc. of the 20th ACM PODS'01, pages 102–113, 2001. 4.1, 4.4, 4.4.1

[2] J. Aspnes and G. Shah: *Skip graphs*, in Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 384–393, 2003. 2.7, 3.3.4

[3] D. Novák and P. Zezula: *M-Chord: A scalable distributed similarity search structure*, In Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 – June 1. IEEE Computer Society, 2006. 2.7

[4] K. F. Riley, M. P. Hobson and S. J. Bence: *Mathematical methods for physics and engineering*, Cambridge University Press, Second edition, 2002. ISBN 0 521 89067 5. 4.5.3

[5] A. K. H. Tung, R. Zhang, N. Koudas and B. C. Ooi: *Similarity search: a matching based approach*, VLDB '06: Proceedings of the 32nd international conference on Very large data bases, 2006, pages 631–642, Seoul, Korea. 6.2

[6] P. Zezula, G. Amato, V. Dohnal and M. Batko: *Similarity Search: The Metric Space Approach*, volume 32 of Advances in Database Systems. Springer-Verlag, 2006. 2.1, 2.2, 2.3.1, 2.4, 2.7, 3.3.4, 4.4

[7] M. Batko, D. Novák and P. Zezula: *MESSIF: Metric similarity search implementation framework*, in Digital Libraries: Research and Development. Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 978-3-540-77087-9, pages 1–10, Lecture Notes in Computer Science, Vol. 4877. 3.1

[8] *MPEG-7. Multimedia content description interfaces. Part 3: Visual.*, ISO/IEC 15938-3:2002, 2002. 3.3

[9] D. Novák, M. Batko, V. Dohnal and P. Zezula: *Scaling up the image content-based retrieval*, in Second DELOS Conference – Working Notes. Pisa, Italy: DELOS Network of Excellence, 2007, ISBN 2-912335-36-1, pages 1–10. 5. 12. 2007, Pisa, Italy. 3.3

[10] M. Batko and P. Zezula: *Scalable index structures in distributed environment*, in Second DELOS Conference – in DATAKON 2004. Brno: Masarykova Univerzita, 2004, ISBN 80-210-3516-1, pages 85–97. 23. 10. 2004, Brno. 2.6.1

[11] M. Batko, P. Kohoutková and P. Zezula: *Combining metric features in large collections*, in First International Workshop on Similarity Search and Applications (SISAP 2008), 2008, ISBN 978-0-7695-3101-4, pages 79–86. 6.1

[12] M. Batko, D. Novak, F. Falchi and P. Zezula: *On scalability of the similarity search in the world of peers*, in Proceedings of INFOSCALE 2006, Hong Kong, May 30 – June 1, 2006, pages 1–12, New York, NY, USA, 2006. ACM Press. 2.7