IST R&D PROJECT

SHARED-COST RTD PROJECT

THEME: INFORMATION ACCESS AND INTERFACES
COMMISSION OF THE EUROPEAN COMMUNITIES
DIRECTORATE GENERAL DG INFSO

PROJECT OFFICER: DR PAT MANSON

Resource Selection and Data Fusion for Multimedia
International Digital Libraries

Prototype tools for resource selection

D3.2

October 29, 2002, UNIDO/WP3/Version 1

Henrik Nottelmann, Norbert Fuhr

IST Project Number	IST-2000-26061	**Acronym**	MIND
Full title	Resource Selection and Data Fusion for Multimedia International Digital Libraries		
EU Project officer	Dr. Pat Manson		

Deliverable	**Number**	D3.2	**Name**	Prototype tools for resource selection			
Task	**Number**	All	**Name**				
Work Package	**Number**	WP3	**Name**	Resource selection			
Date of delivery	**Contractual**		2002/04/30		**Actual**		2002/10/29
Code name	<codename>				**Version** 1	draft ☐	final ☑
Nature	Prototype ☐ Report ☑ Specification ☐ Tool ☑ Other:						
Distribution Type	Public ☑ Restricted ☐ to: Partners, Commission, Reviewers						
Authors (Partner)	Henrik Nottelmann, Norbert Fuhr (UNIDO)						
Contact Person	Henrik Nottelmann						
	Email	nottelmann@ls6.cs.uni-dortmund.de	**Phone**	+49 231 755 5319	**Fax**	+49 231 755 2405	
Abstract (for dissemination)	This is the description of the MIND resource prototype tools. It describes in detail the resource selection framwork used within MIND (which is already described in D3.1 but is modified in some details) and the components involved in resource selection. This document also contains descriptions of the MIND architecture, the SOAP-based communication and the installation.						
Keywords	Resource selection, decision-theoretic framework, logistic function, normal distribution, indexing weights distribution, score distribution, software.						

Contents

1 Introduction **3**

2 System overview **4**

2.1 System architecture 4

2.2 Communication between components 6

2.3 The registry 7

2.4 Communication with Java 7

2.5 Communication with C++ 8

3 System installation **9**

3.1 Java part 9

3.1.1 Overview over the Java part 9

3.1.2 Installation of the Java part 12

3.2 C++ Part 13

4 Resource selection framework and methods **14**

4.1 Decision-theoretic framework 14

4.2 Retrieval model 15

4.2.1 Indexing weights for different media types 15

4.2.2 Relationship between score and probability of relevance 17

4.3 Estimating retrieval quality 18

4.3.1 Method 1: Recall-precision-function 19

4.3.2 Method 2: Simulated retrieval 19

4.3.3 Method 3: Normal distribution 20

5 Resource selection components **22**

5.1 Resource selection in the media-specific score estimation components of the proxies 22

5.1.1 Resource selection mode 1 22

5.1.2 Resource selection mode 2 and 5 22

5.1.3 Resource selection mode 3 and 6 23

5.1.4 Resource selection mode 4 23

5.2 Resource selection in the media-independent proxy components 23

5.3 Resource selection in the dispatcher 24

Chapter 1

Introduction

This document describes the prototype tools for resource selection, their usage and their behaviour. Furthermore, it gives a brief overview over the underlying theory (the resource description framework) which is described in detail in deliverable D3.1 and in [17].

Chapter 2 gives an overview over the MIND system, the system architecture with the different components and the communication among the components via SOAP (as well as the SOAP communication implementation in the various programming languages).

Chapter 3 is the installation manual of the complete MIND system.

Chapter 4 described very briefly the resource selection framework. It presents the overall decision-theoretic model, the underlying retrieval model and three different resource selection methods.

Chapter finally described which components are involved in resource selection as well as their behaviour.

Chapter 2

System overview

This chapter gives an overview over the MIND system and describes the components and the communication between the components.

2.1 System architecture

The MIND system is broken down to several components which can distributed across different computers. This design as a distributed system has multiple advantages:

1. Development is easier because the partners can run their program code on their own computers without the problem of maintaining the sources and compiled programs.
2. The partners can use different existing platforms (hardware, operating systems) and programming languages.
3. Specialised program code can benefit from specialised hardware, software or data available only for one partner.
4. Load balancing and replication can improve the system scalability and reliability.

The disadvantage is that communication is more complex than in an homogeneous and centralised environment.

The current MIND system[1] consists of six component types:

1. The dispatcher is the mediating instance of the system. User interfaces (or other programs) send their queries only to the dispatcher. The dispatcher then selects relevant libraries, forwards the queries to the selected libraries, and performs a very simple merging w. r. t. their scores (as the scores returned by the DLs are not comparable, a sophisticated data fusion step is required in the user interface).
2. As some DLs are non-co-operative and do not return their resource description, there is one proxy per DL in the MIND system. The proxies are responsible for maintaining resource descriptions, estimating retrieval costs, querying the corresponding digital library, and for handling heterogeneous schemas.

   The proxy consists of several components out of 5 types:

   (a) The media-independent part of the proxy is called by the dispatcher and calls the other components.

[1] The original architecture specification [16] is outdated after a huge amount of changes.

(b) The media-specific resource description construction parts of the proxies are responsible for creating resource descriptions from (e.g., from a sample generated by query-based sampling).

(c) The media-specific score estimation parts of the proxies estimate the scores of the top-ranked documents before retrieval (used for resource selection).

(d) The media-specific schema mapping parts of the proxies handle heterogeneous schemas w. r. t. their media type, including query transformation (from the query schema into the DL schema) and document transformation (backwards from the DL schema into the query schema).

(e) The media-independent interface part of the proxies acts as a wrapper for the corresponding library. Thus, this component receives a transformed query, queries the DL, parses the results, and returns a result (w. r. t. the DL schema). In other words, the interface component provides a unique API for all connected DLs.

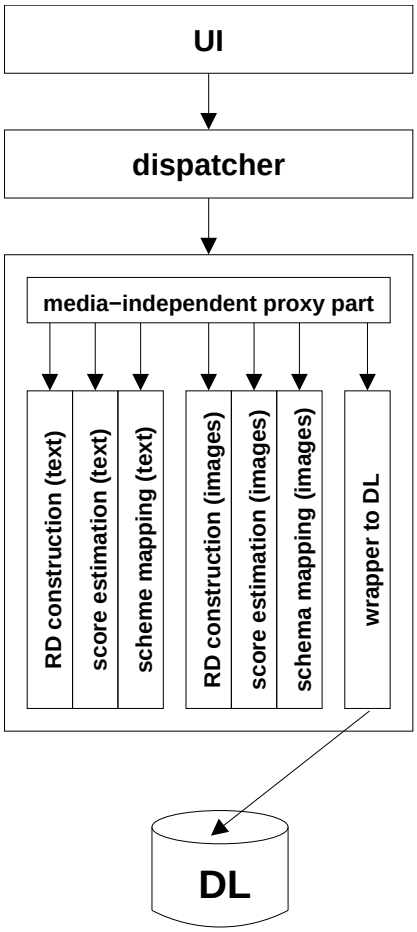


Figure 2.1: MIND communication hierarchy

Communication between the components is organised in a strictly hierarchical way: The user interfaces call the dispatcher. The dispatcher calls the proxies and processes their results. A proxy calls its different media specific parts and its wrapper and processes their results. Furthermore, there will be user interface clients which will to call the dispatcher. This structure is depicted in figure 2.1.

2.2 Communication between components

The MIND communication structure can be implemented with simple procedure calls in a centralised environment. But as the MIND components are distributed, this solution has to be extended to remote procedure calls (RPC).

Within the area of distributed systems, the MIND architecture is called a client-server-architecture: There are "services" running on dedicated computers ("servers"). Each service provides several "procedures", identified by a name and a list of parameter types. "Clients" can call service procedures and process the results returned by that service.

In the MIND system, the dispatcher and the proxies are clients, and all components are services. Thus, within the MIND system the terms "component" and "service" are synonyms. "Component" is used when referring to the core MIND architecture, and "service" when referring to the communication layer.

As communication protocol, the W3C standard SOAP [25] is used. A registry for service URLs is added to simplify system administration.

SOAP messages are encoded in XML, thus it is easy to connect different platforms and programming languages. Thus, the use of XML was a natural choice in the design process. The overhead of following the SOAP standard for XML messages (in contrast to defining an own protocol) is negligible (particularly if Java is used). The advantage is to have an application which follows web standards and, thus, can be called easily by other programs.

In MIND, SOAP will be used in combination with HTTP. To call a service procedure, the client sends (HTTP method POST) an XML document ("request") to the service's URL. The procedure has to process the request and to send back a "response" (another XML document containing the result).

The XML request document contains the envelope (the root element) and the body element (which contains the payload as its immediate child element). The payload element is identified by a fully qualified name, where the service name is the namespace URI and the procedure name is the local name. In MIND, the service name has the form "urn:NAME".

For the several MIND components, "NAME" is split into the component type and the local name:

"urn:dispatcher.Dispatcher" for the dispatcher,

"urn:proxy.Google" for the Google proxy,

"urn:proxy-construction-text.Google" for the text-specific resource description construction component of the Google proxy,

"urn:proxy-estimation-text.Google" for the text-specific score estimation component of the Google proxy,

"urn:proxy-mapping-text.Google" for the text-specific schema mapping component of the Google proxy, and

"urn:proxy-interface.Google" for the wrapper component of the Google proxy.

Besides "text", the media type can also be "facts", "images" and "speech".

Coding the service name into the XML message allows for attaching several services to one URL (handled by a router). E.g., this will be done for services which are implemented in Java.

2.3 The registry

To call a service procedure, the client need to know

1. the name and the URL of the service and
2. the signature (name and parameter list) of the procedure.

The name and the signature of services are known when a service has to be called, in contrast to its URL (which can change). If it is coded in the client's program code, that class has to be recompiled whenever the URL changes. If the URLs are stored in local configuration files, these files have to be changed on every machine. Both solutions leads to a notable administration overhead.

Thus, MIND has a centralised mapping from service names to URLs which can be used by all clients, on whatever computer they are running. The mapping component is called registry, and it is also a SOAP service. Its entries have the form

```
urn:proxy.Google=http://foor.bar.org:12345/google
```

If the URL of a service changes, only the registry has to be updated. Administration tools allow to register and unregister services. Clients can query the registry to retrieve the URL of services. Of course, SOAP will be used for these registry query calls.

More than one URL can be registered for one service. The client can call one or all URLs. This replication allows for load balancing. Furthermore, it provides usability even if one computer is shut down (because then another URL can be tried). Of course, then registry can also be replicated.

To improve efficiency, a client should not query the registry for every service call. Instead, the results should be cached locally for later use. As the URLs for a service can change during lifetime of a client using that service, the client has to maintain its internal mapping. This can be done in two ways:

1. The client memories – for each service – the time of its registry query. When the service is called again, the client queries the registry again if the difference between the current and the last registry query call time exceeds a certain limit ("expiry time").
2. If the client is also a service, it tells the registry its service name when submitting a query. The registry keeps track of such queries notifies (via another SOAP call) the retrieving client about changes of the URL of the service retrieved.

To improve reliability, a service should use both methods and cache its local mapping in RAM (this will be done e.g. in the Java classes).

2.4 Communication with Java

SOAP can be used in all programming languages which provide access to TCP/IP ports. Then, it is possible to listen for HTTP requests and to send HTTP messages. Thus, the requirements for using SOAP are very small and are met e.g. by C, C++, Perl and Java.

Apache SOAP [5] is a simple framework for Java which hides many details of the SOAP communication. This framework and its use within the MIND project is described in this section, as a large amount of the MIND program code is written in Java.

Apache SOAP contains an API used by the clients and some servlets (administration servlets and one router servlet) which run on a servlet server (e.g. Jakarta Tomcat [6]). Thus, all services running on one computer have the same URL – the URL of the router servlet. Incoming SOAP messages are dispatched by the router w. r. t. the service name. Each service (name) corresponds

to a service class (the mapping to be specified while "deploying" a service). Every class can be used as a service (without modification).

The router transforms the XML message into objects, calls the service, transforms the result into XML and sends it back to the client. For the transformation, specialised classes (serializers and deserializers) are used. Besides standard serializers attached to Apache SOAP, we implemented a large amount of other (de)serializers for the Java data structure classes.

To simplify development of clients, there is a "stub" for each service (component). A stub is a Java class implementing the service interface. The stub methods are responsible for the corresponding SOAP call. Thus, the SOAP communication is hidden (the service name just has to be specified as a stub constructor argument), and calling the stub is within a client the same as calling a service class directly.

The communication process is depicted in figure 2.2.

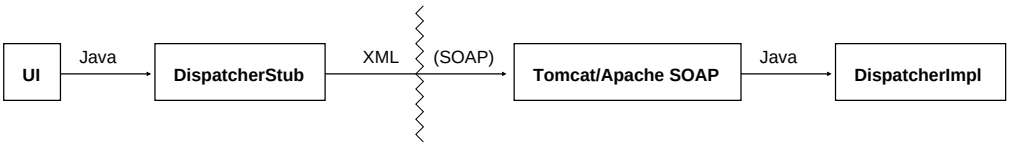


Figure 2.2: MIND communication process in Java

Of course, as SOAP is an interoperability protocol, Java clients can call services written in another programming language, and vice versa.

2.5 Communication with C++

Similar to Apache SOAP for Java, gSOAP [22] provides a useful utility for creating SOAP clients and servers. gSOAP provides a utility that writes SOAP serializers and deserializers based on C++ class and function definitions provided as input. The resulting output is code that can be used by a C++ client or server for networked SOAP communication.

In order to provide support for a remote method call, one must provide an appropriate function definition corresponding to the signature of the SOAP call. The programmer may also have to provide class definitions for arguments and return values. The gSOAP utility can then be used to create a serializer and a deserializer for the function. When writing the client/server, the programmer then uses the generated (de)serializer that was created by gSOAP.

As only the query-based sampling utility currently uses C++, only the SOAP calls and structures needed by the sampling utility were created for input to gSOAP. The complete MIND API could be added to the C++ support if needed.

Chapter 3

System installation

This chapter describes the files and the directory structure of the MIND system.

3.1 Java part

This section covers the Java part of MIND. It starts with an overview over the implementation, and finishes with detailed installation notes.

3.1.1 Overview over the Java part

The following parts of the MIND system are implemented in Java:

1. The registry is entirely written in Java (package `mind.registry`).
2. The dispatcher is entirely written in Java (package `mind.dispatcher`).
3. The media-independent part of the proxy is written in Java and can be reused by all proxies without big changes (only the proxy ID as to be set).
4. For most of the other components of the proxies, there are basic classes written in Java which have to be subclasses for concrete proxies. These components can also be reused.

All Java services run on a Tomcat server [6] using Apache SOAP [5]. The URL of a service is `http://{host}:{port}/soap/servlet/rpcrouter`.

The Java part of the MIND system resides in one directory, specified by the environment variable `$MIND_HOME`. This directory is structured as follows:

`$MIND_HOME/.bashenv` initialisation file for Unix, Solaris, Linux etc.

`$MIND_HOME/build.xml` build file for ant

`$MIND_HOME/bin/` executables and utility files for the executables

`$MIND_HOME/conf/` configuration files

`$MIND_HOME/conf/mind/` configuration files for MIND (see below)

`$MIND_HOME/conf/soap/` configuration files for Tomcat and Apache SOAP

`$MIND_HOME/doc/` JavaDoc documentation

`$MIND_HOME/lib/` libraries (MIND Java classes, Java libraries required by Tomcat and Apache SOAP)

`$MIND_HOME/man/` man pages for the executables in `$MIND_HOME/bin/`

`$MIND_HOME/src/` source files

It is possible to have more than one installation on a computer, e.g. one development system and one productive system.

The initialisation file `$MIND_HOME/.bashenv` should be called during system initialisation, as it sets some required environment variables. It is only tested under bash (Solaris and Linux), but it is easy to modify it for other systems.

The Java part of MIND heavily relies on Jakarta Ant [1], a Java-based substitution for Make. The file `Makefile` used by make is replaced by an XML document file `$MIND_HOME/build.xml`. Like Make, Ant has to be called from the directory which contains `$MIND_HOME/build.xml` (alternatively, the file has to be specified as an argument).

The `$MIND_HOME/build.xml` files defines some targets for

- compiling all changed Java files (or all files) in `$MIND_HOME/src/mind/` and putting them into `$MIND_HOME/lib/mind/`,
- creating the JavaDoc documentation in `$MIND_HOME/doc/`,
- starting, stopping and restarting Tomcat servers,
- deploying and undeploying special proxies, the dispatcher and the registry,
- several operations with the CVS server holding the project files, and
- installing the system from the CVS repository (see below).

The directory `$MIND_HOME/conf/mind/` contains configuration files for MIND itself:

`$MIND_HOME/conf/mind/config` configuration, containing names Java properties:

`registry-request.expire-seconds` expiry time (in seconds) for registry query call results: if a service is called and the last registry query call for this service was carried out before this number of seconds, a new registry query call should be issued

`http.proxyHost` the host of the proxy (optional)

`http.proxyPort` the port of the proxy (optional)

`urn:Registry` the URL of the MIND registry (see below)

`gnuplot.bin` path and name of the Gnuplot tool[1], used for learning parameters and for evaluation (probably does not work under Windows)

`trec.path` path of the raw TREC data (ony required for constructing indices for TREC databases)

`trec.files` path of the converted files for the TREC "wrappers" (actually, they are small IR engines)

`bibdb.path` path of the converted files for the BIBDB "wrappers" (actually, they are small IR engines)

`tmp.path` path for temporary files

`database.uri` complete JDBC-URI for the resource description database (not used for all proxies)

[1] `http://www.ucc.ie/gnuplot/gnuplot.html`

`database.driver` class of the JDBC driver for the resource description database (not used for all proxies)

`database.user` user name for the resource description database (not used for all proxies)

`database.password` password name for the resource description database (not used for all proxies)

`database.log` name of log file for the resource description database (has to be the same for all databases using the same "host")

`database.proxy id.uri` complete JDBC-URI for the resource description database for the specified proxy (the proxy id is the part after `urn:proxy.`)

`database.proxy id.driver` class of the JDBC driver for the resource description database for the specified proxy (the proxy id is the part after `urn:proxy.`)

`database.proxy id.user` user name for the resource description database for the specified proxy (the proxy id is the part after `urn:proxy.`)

`database.proxy id.password` password name for the resource description database for the specified proxy (the proxy id is the part after `urn:proxy.`)

`database.proxy id.log` name of log file for the resource description database for the specified proxy (the proxy id is the part after `urn:proxy.`); this entry has to be the same for all databases using the same "host"

Furthermore, the configuration files contains LOG4J [4] configuration statements, but these are subject to change and thus not described here.

`$MIND_HOME/conf/mind/registry` service mapping file for the registry, each entry has the form

```
urn:proxy.Reuters=http://foor.bar.org:12345/reuters
```

`$MIND_HOME/conf/mind/schemas/` schemas for wrapper components of proxies in XML format (part of the proxy id before an optional dot plus `.xml`, e.g. `Google.xml` for `urn:proxy.Google`, `TREC` for the proxy `urn:proxy.TREC.ap88_1`), loaded automatically by the corresponding service

Configuration files for Tomcat and Apache SOAP are stored in `$MIND_HOME/conf/soap/`. It is possible to install several Tomcat servers on one machine (or on several computers which share the same file system). Then, it is necessary that each server instance has its one configuration files. Thus, `$MIND_HOME/conf/soap/` contains one subdirectory for each server. The name of the directory is the name of the server instance and can be used by the administration tools.

There are two major tools for MIND. Both are written in Java, both have a small shell wrapper script in `$MIND_HOME/bin/`, and both have a integrated help screen (option `-h`) and a man page in `$MIND_HOME/man/`.

The program `tomcat` is responsible for handling the Tomcat servers for the MIND services. It allows for creating, deleting, starting, stopping and restarting Tomcat servers (identified by their name), deploying and undeploying services (and, as an option, for calling the registry program which is described below) and displaying the status of the servers. For installation, `tomcat` is not directly called by implicit by Ant.

The second program is `registry`, and – as the name says – is responsible for handling the registry service (which is also a SOAP service). It allows for creating, deleting, starting, stopping and restarting the corresponding Tomcat server (the server instance name is also `registry`), registering and unregistering services for a URL (for external servers) or a Tomcat server instance (for internal Java services), for displaying the status of the registry, and for querying with a sub-language of regular expressions (as it can also be called with the registry stub). For installation, `registry` is not directly called by implicit by Ant.

3.1.2 Installation of the Java part

MIND is delivered with many software components it relies on (Jakarta Tomcat, Apache SOAP, Xerces etc.). Nevertheless, there are a few programs which have to be installed before MIND can be installed:

1. MIND requires an installed virtual machine [3] (at least J2SE 1.2.2, but J2SE 1.3 is recommended, J2SE 1.4 is not yet tested). For only running MIND, the runtime environment (JRE) is sufficient. For development, the JDK is required.
2. Ant [1] must be installed.
3. A CVS client [2] must be installed as well.

This installation manual does not cover the installation of Java, Ant and CVS, as these programs will be already installed on most machines.

The environment variable `$CVSROOT` has to be set to the correct value (at the moment, it is `:pserver:{loginname}@devweb.cis.strath.ac.uk:/var/network/cvs_repositries/mind`, where `{loginname}` denotes the login name which can be delivered by the USG system support (`mailto:support@cis.strath.ac.uk`).

To install MIND, the following steps have to be executed:

1. Create a directory and go into that directory.
2. For the first time using the CVS server, you have to log in:
   `cvs login`
   Now, you have to type your CVS password.
3. Check out the Ant build file:
   `cvs co build.xml`
4. Install the MIND files from the CVS directory:
   `ant install`
5. In the file `.bashenv`, set the domain s.th. `‘hostname‘.${MIND_DOMAIN}` is the complete host name for all servers. For Dortmund, this is e.g.
   `export MIND_DOMAIN=cs.uni-dortmund.de`
6. Edit `~/.bash_profile` (or the appropriate init file if sh, csh, tcsh etc. is used) by adding
   `. {install}/.bashenv`
   where of course `{install}` has to be replaced by the installing directory path.
7. Maybe you have to set the WWW proxy host and port in the file `conf/mind/config`. For Dortmund, this is e.g.
   `http.proxyHost=fbi-www.cs.uni-dortmund.de`
   `http.proxyPort=3128`
8. Log out and in again to make the changes visible.
9. Change to the MIND directory.
10. Compile all source files:
    `ant compileall`
11. Create the newest JavaDoc documentation:
    `ant javadoc`

Now, the MIND files are installed from the CVS repository. The next two steps cover the installation of MIND servers. Currently, we are using two servers, one for the registry (called `registry`) and one for all other services (called `services`).

1. Create and start both Tomcat servers on host {`host`} with port 15300 for `registry` and port 15310 for `services`:
   `ant -Dhost={host} install-server`
   You can set other ports by
   `ant -Dhost={host} -Dregistryport={port of registry} -Dport={port of services} install-server`

2. Deploy and register the registry on server `registry` and all other services (at the moment, the dispatcher and the proxies for Google, respectively):
   `ant deploy-all`

Furthermore, for running most proxy components where the schema contains text or fact attributes a relational database management system has to be installed, and the file `$MIND_HOME/conf/mind/config` has to configured accordingly.

3.2 C++ Part

The C++ part consists of the query-based sampling utility. It is a component of the Lemur [9] toolkit for information retrieval. Directions for installation and use of the Lemur toolkit [8] should be followed for installation of the query-based sampling utility.

Chapter 4

Resource selection framework and methods

This chapter describes briefly the resource selection framework and methods. Further information is given in D3.1 and in [17].

4.1 Decision-theoretic framework

Our methods are based on the decision-theoretic framework presented in [13]. The basic assumption is that we can assign specific retrieval costs $C_i(s_i, q)$ to each digital library DL_i when s_i documents are retrieved for query q. The term "costs" is used in a broad way and also includes other cost factors (beside money) like time and quality as discussed below.

If the user specifies (together with her query) the total number n of documents which should be retrieved, the task then is to compute an optimum solution, i.e. a vector $\vec{s} = (s_1, s_2, \ldots, s_m)^T$ with $|\vec{s}| = \sum_{i=1}^{m} s_i = n$ which minimises the overall costs:

$$M(n, q) := \min_{|\vec{s}|=n} \sum_{i=1}^{m} C_i(s_i, q). \tag{4.1}$$

Different cost sources should be considered:

Effectiveness of the library: Probably most important, a user is interested in getting many relevant documents. Let $r_i(s_i, q)$ denote the number of relevant documents in the result set when s_i documents are retrieved from library DL_i for query q. We assign library-independent costs C^+ for viewing a relevant document and C^- for viewing an irrelevant document (with $C^+ < C^-$). The resulting cost function is

$$C_i^{rel}(s_i, q) := r_i(s_i, q)C^+ + [s_i - r_i(s_i, q)] \cdot C^-.$$

Estimating the number $r_i(s_i, q)$ of relevant documents in the result is the most challenging task for resource selection, and thus the major part of this paper.

Time: There is computation time at the library site and communication time for delivering the result documents over the network. In many cases, a linear function can be used, where we have initial constant costs for querying and costs linear in the number of documents retrieved:

$$C_i^{time}(s_i, q) := C^{time,init} + s_i, q \cdot C^{time,doc}.$$

It is rather simple to estimate the time cost factors by sending probe queries to the digital library and measuring the response time [10].

Money: Many digital libraries will be for free, but some DLs may charge. For some users, monetary costs may be one of the major issues. In many cases, there are simple per-document-charges, thus a linear function. The initial constant costs are zero in most applications, but modelled anyway:

$$C_i^{money}(s_i, q) := C^{money,init} + s_i, q \cdot C^{money,doc}.$$

Monetary costs have to be specified manually, are there is no easy way to derive them automatically.

For resource selection, the costs have to be known *before* the libraries are queried. As we cannot know the *actual* number $r_i(s_i, q)$ of relevant documents retrieved by the library in advance, we can only use its expectation $E[r_i(s_i, q)]$. This leads to *expected* costs

$$EC_i(s_i, q) := EC_i^{sys}(s_i) + EC_i^{rel}(s_i, q).$$

The task then is to minimise the expected overall costs, thus we have to replace formula 4.1 by

$$EM(n, q) := \min_{|\vec{s}|=n} \sum_{i=1}^{m} EC_i(s_i, q).$$

For the cost function described above, the expected costs $EC_i(s_i, q)$ are increasing with the number s_i of documents retrieved. Thus, the algorithm presented in [13] can be used for computing an optimum solution.

4.2 Retrieval model

In MIND, a query q in this model is a set of conditions c_j with query term weight $Pr(q \leftarrow c_j)$. A condition always refers to one of the attributes in the corresponding schema, one of the attributes search predicates, and has a comparison value. Furthermore, we consider the probability $Pr(c_j \leftarrow d)$ (the indexing weight of document d w. r. t. condition c_j).

As usual in modern IR, we view information retrieval as uncertain inference [23] and consider the probability $Pr(rel|q, d)$ that document d is relevant to query q. Here, we split the query q according in subqueries q_p w. r. t. the different predicates p:

$$Pr(\text{rel}|q, d) := \sum_{p \in q} Pr(\text{rel}|q_p, d).$$

Furthermore, the relationship between this probability of relevance and the probability $Pr(q_p \leftarrow d)$ that d logically implies q_p (probability of inference) is modelled with a predicate-specific function $f_p : [0, 1] \rightarrow [0, 1]$:

$$Pr(\text{rel}|q_p, d) = f_p(Pr(q_p \leftarrow d)).$$

In addition, the widely used linear retrieval function [21, 24] is applied for computing the probability of inference:

$$Pr(q_p \leftarrow d) \quad := \quad \sum_{c_j \in q_p} \underbrace{Pr(q_p \leftarrow c_j)}_{\text{query term weight}} \cdot \underbrace{Pr(c_j \leftarrow d).}_{\text{indexing weight}}$$

4.2.1 Indexing weights for different media types

All methods rely on the probability $Pr(c \leftarrow d)$ that a document d implies a query condition c. In the next paragraphs the meaning and computation of this probability (the indexing weight of document d w. r. t. condition c) is described for the different media types.

Indexing weights for text

For text, the comparison value of a condition c_j is a term t_j, and only the predicate `contains` is considered. Then, $Pr(t_j \leftarrow d)$ is the indexing weight of the term t_j in document d, e.g. a BM25 weight [18]:

$$Pr(c \leftarrow d) := \frac{tf(t,d)}{tf(t,d) + 0.5 + 1.5 \cdot \frac{dl(d)}{avgdl}} \cdot \frac{\log \frac{N}{df(t)}}{\log |DL|}.$$

Here, the *idf* component is normalised with $(\log |DL|)^{-1}$ (the size of the actual DL) to obtain values in $[0, 1]$. In addition, $tf(t, d)$ denotes the frequency of term t in document d, $dl(d)$ the length of document d (sum of term frequencies), $avgdl$ the average document length, N the number of documents in the collection to index (the complete DL or a sample) and $df(t)$ the document frequency of term t (the number of documents containing term t).

Indexing weights for facts

For facts, the computation of $Pr(c \leftarrow d)$ relies on the datatype of the attribute the condition refers to and the predicate:

Datatype "name": This datatype has the two certain predicates `=` and `soundex`, thus we get:

$$Pr(c \leftarrow d) \in \{0, 1\}$$

Other, vague predicates like `edit-distance` or `n-grams` could be considered as well; they will be integrated in later stage.

Datatype "year": This datatype has the certain predicates `=`, `<`, `>`, `<=`, `>=` and `in` (the latter one having two numbers as comparison value), thus we get for these predicates:

$$Pr(c \leftarrow d) \in \{0, 1\}$$

Furthermore, three different vague predicates $\sim$`=`, $\sim$`<` and $\sim$`>` are defined. For $\sim$`>`, we have:

$$Pr(c \leftarrow d) := \begin{cases} 1 - \frac{c.v - d.v}{d.v} & , \quad c.v > d.v \\ 1 & , \quad else \end{cases}$$

For $\sim$`=`, we define:

$$Pr(c \leftarrow d) := \begin{cases} 1 - (\frac{c.v - d.v}{d.v})^2 & , \quad c.v > d.v \\ 1 & , \quad else \end{cases}$$

Indexing weights for images

For images, different similarity functions $Sim(c, d)$ for comparing the condition value c and the image stored in document d are considered (e.g. for comparing colour histograms or textures):

$$Pr(c \leftarrow d) := Sim(c, d).$$

Generally, the similarity is defined with respect to a distance measure. Possible distance measures for color histograms include Minkowski-form distance, Histogram intersection, Kullback-Leibler divergence, χ^2 statistics or a quadratic-form distance [20], [14], [15].

Indexing weights for speech

In principle, indexing weights for speech are computed in the same way as for text, e.g. using BM25. The primary difference between resource selection for speech transcripts and normal text files lies in the effect of word error rate (WER) on speech recognised documents. It is observed that a lower tf value of words in recognised documents occurs due to recognition errors and this causes retrieval models to favour normal text files (or hand transcripts of speech) to speech recognised ones [19], or prefer speech documents with lower WER to those with higher WER. To compensate this unbalancing in resource selection, some smoothing techniques need to be sought to increase the tf value in speech documents according to the level of their WER.

4.2.2 Relationship between score and probability of relevance

The relationship between this probability of inference (the score) and the probability $Pr(rel|q,d)$ of d being relevant to q is modelled with a function $f : [0,1] \rightarrow [0,1]$:

$$Pr(\text{rel}|q,d) = f(Pr(q \leftarrow d)).$$

According to [23], we can compute the probability of relevance as

$$\begin{aligned} Pr(\text{rel}|q,d) &= Pr(\text{rel}|q \leftarrow d) \cdot Pr(q \leftarrow d) + \\ & \quad Pr(\text{rel}|\neg(q \leftarrow d)) \cdot Pr(\neg(q \leftarrow d)). \end{aligned}$$

If we assume $Pr(\text{rel}|\neg(q \leftarrow d)) \approx 0$, we obtain

$$Pr(\text{rel}|q,d) = Pr(\text{rel}|q \leftarrow d) \cdot Pr(q \leftarrow d).$$

Thus, the relationship between the probabilities $Pr(\text{rel}|q,d)$ and $Pr(q \leftarrow d)$ can be modelled with the linear function

$$f : [0,1] \rightarrow [0,1], f(x) := Pr(\text{rel}|q \leftarrow d) \cdot x.$$

Our experiments show that in practice there is no linear relationship. So, we considered other functions as well. In addition, we found that this mapping function is predicate-specific. Thus, we form attribute/predicate-specific subqueries $q_p \subseteq q$, and map the score $Pr(q_p \leftarrow d)$ onto a probability of relevance.

Indexing weights for text

Figure 4.1 shows the actual relationship between the probability of inference and the probability of relevance for an real-world example library and query (using a BM25 "contains term with stemming and stopword removal" predicate), and two fits (linear and logistic).

In an optimum situation, exactly the top-ranked documents (where score exceeds a specific threshold) are relevant. In practice, this is rarely ever the case. Instead, we want a continuous function f as an approximation. One good candidate is a logistic function [11, 12]

$$f_p : [0,1] \rightarrow [0,1], \; f(x) := \frac{\exp(b_0 + b_1 x)}{1 + \exp(b_0 + b_1 x)}$$

with the two parameters b_0 and b_1.

For resource selection, we do not have query-specific relevance data to compute the optimum parameters b_0 and b_1. Thus, we compute a global function f_p (for each digital library) with a learning sample.

Our experiments showed that in some cases we can obtain a better quality if we normalise the scores on a per-query-basis:

$$Pr(\text{rel}|q,d) = f_p\left(\frac{Pr(q \leftarrow d)}{max_{d' \in DL} Pr(q \leftarrow d')}\right).$$

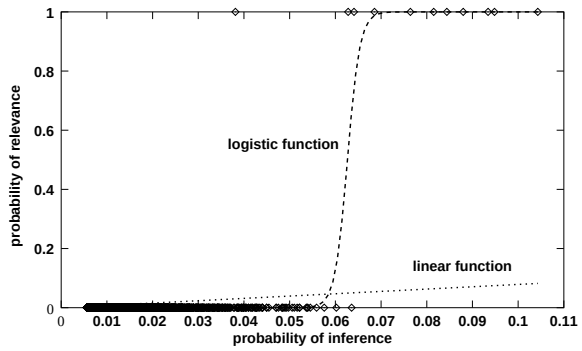


Figure 4.1: Example query results and fit with linear and logistic function

Indexing weights for facts

For the certain predicates, we simply use the identity mapping function:

$$f_p \equiv id \ , \ f_p(x) := x.$$

For the vague predicates, we apply a logistic function.

$$f_p : [0,1] \rightarrow [0,1], \ f(x) := \frac{\exp(b_0 + b_1 x)}{1 + \exp(b_0 + b_1 x)}$$

with the two parameters b_0 and b_1.

Indexing weights for images and speech

For speech and images, we will also use a logistic function.

4.3 Estimating retrieval quality

Resource selection accuracy in this model heavily depends on good approximations of the number of relevant documents in the result set.

Three methods for estimating the number $r_i(s_i, q)$ of relevant documents in the result set of library DL_i for query q (where s_i documents are to be retrieved) are described in the rest of this section.

All estimation methods require some information about the underlying library, stored in *resource descriptions* (see D2.2), e.g.:

- the number of documents $|DL_i|$, for which no automatic derivation method is available so far,
- average indexing weights $avgweight(c_j, d)$, acquired (at least for text) with the technique of query-based sampling [7], or
- query-independent approximation of $Pr(\text{rel}|q \leftarrow d)$, b_0 and b_1, learned from some relevance judgements.

4.3.1 Method 1: Recall-precision-function

Using the linear function f and the linear retrieval function for $Pr(q \leftarrow d)$, the expected number $E(\text{rel}|q, DL_i)$ of relevant documents in DL_i can then be computed as:

$$E(\text{rel}|q, DL_i) = |DL_i| Pr(\text{rel}|q \leftarrow d) \cdot \sum_{c_j \in q} Pr(q \leftarrow t_j) \cdot avgweight(c_j)$$

with the average probability of implication (stored in the resource description)

$$avgweight(c_j) := \frac{1}{|DL_i|} \sum_{d \in DL_i} Pr(c_j \leftarrow d).$$

For text, this is the average indexing weight.

For computing the expected number of relevant documents in the retrieval result, we get

$$E[r_i(s_i, q)] = s_i \cdot EP_i(s_i, q).$$

The expected precision $EP_i(s_i, q)$ is derived from the recall-precision curve of DL_i, which we approximate by the linear function

$$P_i : [0, 1] \rightarrow [0, 1],\ P(R) := P_i^0 \cdot (1 - R)$$

with the library-specific constant P_i^0 (also stored in the resource description). Then, we get:

$$E[r_i(s_i, q)] = \frac{P_i^0 R_i s}{R_i + P_i^0 s}.$$

As described in deliverable D2.2, using sampled resource descriptions for facts imposes the problem of missing values. To overcome this problem, a (low) standard average value can be chosen (implemented in the prototype). Alternatively, n-grams or author networks could be used for overcoming this problem for names, and a smoothing technique for years.

For images, the average indexing weight of c_i can be estimated using the information stored in resource descriptions. In particular, this is computed as:

$$avgweight(c_j) := \frac{\sum_{cc_i \in RD(g)} N_i * Sim(c_i, cc_i)}{\sum_{cc_i \in RD(g)} N_i}$$

Here, cc_i a cluster centre, N_i its cardinality, $Sim(c_i, cc_i)$ the similarity between the query condition c_i and cluster cc_i and $RD(g)$ the set of clusters stored in the resource description at granularity g.

4.3.2 Method 2: Simulated retrieval

Instead of only storing statistical metadata like average indexing weights $avgweight(t_j, d)$ in the resource descriptions, we can index the complete sample. Then, all document-term pairs together with the indexing weights are stored in the resource description. In the resource selection phase, retrieval is simulated with the same query q on this small sample (300 documents), obtaining a probability of relevance $Pr(\text{rel}|q, d)$ for each sample document. The preliminary result is the empirical, discrete density p of the corresponding distribution.

Figure 4.2 shows how we can estimate the number of relevant documents in the result set of s documents. The grey area (the area below the graph from a to 1) denotes the fraction $s/|DL|$ of the documents in the library which are retrieved. Thus, we have to find a point $a \in [0, 1]$ such that

$$s = |DL| \int_a^1 p(x)\ dx.$$

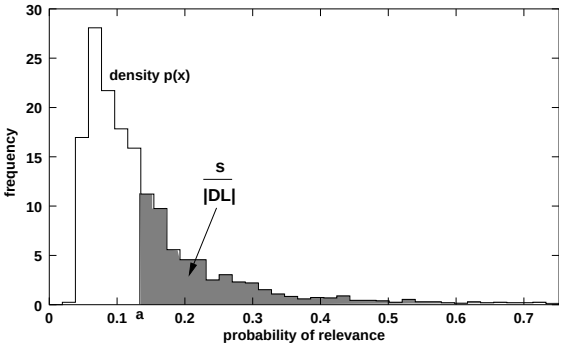


Figure 4.2: Density of probabilities of relevance and computation of $E[r(s,q)]$

In other words, we have to find the smallest probability of relevance among the s retrieved documents.

Then, the expected number of relevant documents in the result set can be computed as the expectation of the probabilities of relevance (defined by the density p) in this area:

$$E[r(s,q)] = |DL| \int_a^1 p_i(x) \cdot x \; dx.$$

More research is required for adapting this method for facts. When a complete resource description is used, retrieval will be done for for every selected library (once for resource selection, once for retrieval). In a sampled resource description, there is the known problem of missing values (see D2.2).

4.3.3 Method 3: Normal distribution

Like the previous methods, we try to estimate the distribution of the probabilities of relevance $Pr(\text{rel}|q,d)$. Instead of a pure empirical approach (as before), here we develop a new theoretic model for the relationship between the desired distribution and the distribution of the indexing weights.

We conducted some initial experiments and found – for text indexed with BM25 – that the empirical, discrete distribution of the indexing weights in the collection can be approximated best by a continuous normal distribution

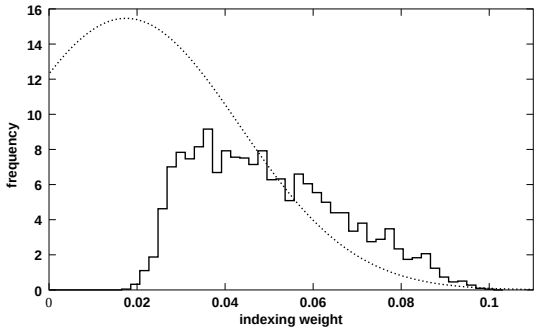
$$p(x,\mu,\sigma) := \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

which is defined by two parameters, the expectation μ (average indexing weight) and the variance σ. These parameters are also stored in the resource description. One example is displayed in figure 4.3(a). Here, we left out a huge peak at zero for improved readability, corresponding to the large amount of documents which do not contain the term.

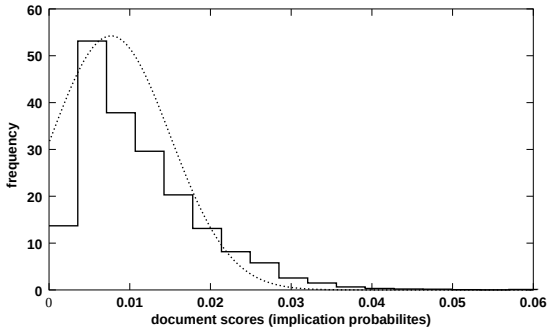
As we consider a linear retrieval function, the distribution of the scores can also be approximated by a normal distribution (see figure 4.3(b)), where the parameters can be computed efficiently:

$$\mu = \sum_{i=1}^{l} a_i \cdot \mu_{t_i}, \quad \sigma = \sqrt{\sum_{i=1}^{l} (a_i \cdot \sigma_{t_i})^2}.$$

For facts, there is no distribution which could really be modelled (e.g. certain predicates, only a few comparison values with non-zero weights). This makes this approach infeasible for facts.



(a) Indexing weight distribution, normal distribution fit



(b) Score distribution, normal distribution fit

The same is true for images.

For speech, more research (i.e., experiments) are required for testing if the indexing weight distribution can be modeled e.g. with a normal distribution. In principle, the indexing weight for texts is used, but the word error rate is used for smoothing the tf and idf values.

Chapter 5

Resource selection components

Three different component types are responsible for resource selection:

1. the media-specific score estimation components of the proxies, being responsible for either estimating the average score of all documents in the library or for estimating the scores of the top-ranked documents,
2. the media-independent proxy components, who compute costs for the different cost sources (at the moment, this is "relevance", "money" and "time"), and
3. the dispatcher who has the task to normalise the costs returned by the proxies, to compute total proxy costs, and to compute an optimum solution.

All three resource selection methods are implemented (defined by the "costs parameter" `rs.mode`). In addition, mode 4 denotes using retrieval on a complete resource description (when available, only for evaluation purpose), mode 5 corresponds to mode 2 with normalised scores and mode 6 corresponds to mode 3 with normalised scores.

5.1 Resource selection in the media-specific score estimation components of the proxies

The media-specific score estimation components of the proxies `DLScoreEstimation` (or one of the media-specific subclasses) are responsible for either estimating the average score of all documents in the library or for estimating the scores of the top-ranked documents. The actual work depends on the resource selection method selected by the user.

5.1.1 Resource selection mode 1

For mode 1, the method `DLScoreEstimation.getAverageScore(PropQuery)` is called (via SOAP) by the media-independent proxy component.

Then, the average score of all documents in the DL is computed based on the average indexing values stored in the resource description and the formula presented in section 4.3.1.

5.1.2 Resource selection mode 2 and 5

For modes 2 and 5, the method `DLScoreEstimation.getScores(PropQuery)` is called (via SOAP) by the media-independent proxy component.

Then, retrieval is simulated on the sample stored in the resource description. The discrete score distribution is computed, and then the scores of the top-ranked documents in the complete library are extrapolated (see section 4.3.2).

5.1.3 Resource selection mode 3 and 6

For modes 3 and 6, the method `DLScoreEstimation.getScores(PropQuery)` is called (via SOAP) by the media-independent proxy component.

Then, expectation μ and variance σ of the score normal distribution are computed based on the indexing weight expectations and variances stored in the resource description and the formula presented in section 4.3.3.

This method cannot be used yet for facts and images (see section 4.3.3).

5.1.4 Resource selection mode 4

For mode 4, the method `DLScoreEstimation.getScores(PropQuery)` is called (via SOAP) by the media-independent proxy component.

Then, retrieval is performed on the complete resource description (containing an index of all documents in the library), computing scores of the top-ranked documents.

This mode (which is mode 2 for complete resource descriptions) is only useful for evaluation, as usually we don't have complete resource descriptions.

5.2 Resource selection in the media-independent proxy components

The task of the media-independent proxy components is to compute costs for the different cost sources (at the moment, this is "relevance", "money" and "time").

Such a component is called by the dispatcher via its `DLProxy.calcExpectedCosts(Query)` method.

Firth, the media-independent proxy component calls (via SOAP) all media-specific score estimation components whose media type occurs in at least one query condition for score estimation:

- For resource selection method 1, `DLScoreEstimation.getAverageScore(PropQuery)` is called, returning the average score of all documents in the library. Then, a linear function is used for mapping it into the average probability of relevance, and multiplication with $|DL|$ yields the expected number of relevant documents in the DL. Finally, the recall-precision-function has to be applied for computing for every $0 \leq s \leq n$ the number of relevant documents in the first s top-ranked documents.
- For the resource selection methods 2 and 3, `DLScoreEstimation.getScores(PropQuery)` is called, returning a list of scores of the top-ranked documents. Now, a logistic function is applied for mapping the scores onto the probability of relevance, and the number of relevant documents in the first s top-ranked documents is derived (for $0 \leq s \leq n$).

Then, the costs for relevance are computed based on the number of relevant documents in the first s top-ranked documents (for $0 \leq s \leq n$), where relevant documents have costs 0 and irrelevant documents have costs 1.

In addition, the costs for money and time are computed (both with a linear function), where the cost factors (for initial costs and costs linear in the number of documents) are extracted from the resource description.

5.3 Resource selection in the dispatcher

The dispatcher has the task to normalise the costs returned by the proxies, to compute total proxy costs, and to compute an optimum solution.

From outside, the dispatcher method `Dispatcher.calcResourceSelection(Query)` is called (via SOAP) for resource selection.

First, the dispatcher calls (via SOAP) all media-independent proxy components for their estimated costs of retrieval (`DLProxy.calcExpectedCosts(Query)`). Each proxy returns a list of cost arrays (expected costs $EC_{source}(s)$, $0 \leq s \leq n$), one cost array per cost source (at the moment, this is "relevance", "money" and "time").

Then, for each cost source the maximum value in the cost arrays of all proxies is computed, and all cost values are normalised with this maximum. As a consequence, all values in the cost arrays are in $[0, 1]$.

In the third step, one array with total costs (the sum of the costs for all cost sources) is computed for each proxy:

$$EC(s) := \sum_{source} c_{source} \cdot EC_{source}(s), 0 \leq s \leq n.$$

Here, c_{source} denotes the user-defined cost parameters `rs.costs.relevance`, `rs.costs.money` and `rs.costs.time`.

Finally, an optimum solution $\vec{s}$ is computed, where s_i denotes the number of documents which have to be returned by that proxy. If the "cost parameter" `rs.numDL` is available and not zero, then exactly `rs.numDL` libraries are selected (this is not the overall optimum in the general case, but useful for comparing the results with resource ranking algorithms with a fixed number of selected libraries).

Afterwards, only the proxies DL_i with $s_i > 0$ are queried (calling `DLProxy.query(Query,int)` via SOAP).

List of Figures

2.1 MIND communication hierarchy 5
2.2 MIND communication process in Java 8

4.1 Example query results and fit with linear and logistic function 18
4.2 Density of probabilities of relevance and computation of $E[r(s,q)]$ 20

Bibliography

[1] Ant 1.4. http://jakarta.apache.org/ant/.

[2] CVS - Concurrent Versions System. http://www.gnu.org/software/cvs/.

[3] Java 2 standard edition. http://java.sun.com/j2se.

[4] LOG4J. http://jakarta.apache.org/log4j.

[5] Apache Software Foundation. Apache SOAP. http://xml.apache.org/soap/.

[6] Apache Software Foundation. Jakarta Tomcat. http://jakarta.apache.org/tomcat/.

[7] J. Callan and M. Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems*, 19(2):97–130, 2001.

[8] CMU. Installing and running lemur (version 1.9). `http://www-2.cs.cmu.edu/~lemur/1.9/install.html`.

[9] CMU. The lemur toolkit for language modeling and information retrieval. `http://www-2.cs.cmu.edu/~lemur/`.

[10] N. Dushay, J. French, and C. Lagoze. Predicting indexer performance in a distributed digital library. In *Research and Advanced Technology for Digital Libraries*, Berlin et al., 1999. Springer.

[11] S. Fienberg. *The Analysis of Cross-Classified Categorial Data*. MIT Press, Cambridge, Mass., 2. edition, 1980.

[12] D. Freeman. *Applied Categorial Data Analysis*. Dekker, New York, 1987.

[13] N. Fuhr. A decision-theoretic approach to database selection in networked IR. *ACM Transactions on Information Systems*, 17(3):229–249, 1999.

[14] S. Kullback. Information theory and statistics. Dover, New York, NY, 1968.

[15] W. Niblack, R. Barber, E. W., M. Flickner, G. E.H., D. Petkovic, P. Yanker, C. Faloutsos, G. Taubin, and Y. Heights. Querying images by content, using color, texture, and shape. In *Proceedings SPIE Conference on Storage and Retrieval for Image and Video Databases*, 1993.

[16] H. Nottelmann. D1.1: Test-bed architecture specification.

[17] H. Nottelmann and N. Fuhr. Predicting retrieval quality for resource selection in distributed information retrieval. Technical report, Universität Dortmund, Dortmund, Germany, 2002. `http://ls6-www.cs.uni-dortmund.de/bib/fulltext/ir/Nottelmann_Fuhr:02.pdf`.

[18] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, M. Gatford, and A. Payne. Okapi at TREC-4. In *Text REtrieval Conference 4*, pages 73–96, 1996.

[19] M. Sanderson and F. Crestani. Mixing and merging for spoken document retrieval. In *Proceedings of the 2nd European Conference on Digital Libraries; Heraklion, Greece, September 1998, pp397-407. Lecture Notes in Computer Science N. 1513, Springer Verlag, Berlin, Germany*, 1998.

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

[21] H. Turtle and W. Croft. Efficient probabilistic inference for text retrieval. In *Proceedings RIAO 91*, pages 644–661, Paris, France, 1991. Centre de Hautes Etudes Internationales d'Informatique Documentaire (CID).

[22] R. van Engelen. gSOAP: SOAP C++ web services. `http://www.cs.fsu.edu/~engelen/soap.html`.

[23] C. J. van Rijsbergen. A non-classical logic for information retrieval. *The Computer Journal*, 29(6):481–485, 1986.

[24] S. Wong and Y. Yao. On modeling information retrieval with probabilistic inference. *ACM Transactions on Information Systems*, 13(1):38–68, 1995.

[25] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/.