Enabling Federated Search with Heterogeneous Search Engines

Combining FAST Data Search and Lucene

Sergey Chernov[1], Bernd Fehling[2], Christian Kohlschütter[1], Wolfgang Nejdl[1], Dirk Pieper[2], and Friedrich Summann[2]

[1]L3S Research Center
University of Hannover
Expo Plaza 1
30539 Hannover

[2]Bielefeld University Library
University of Bielefeld
Universitätsstr. 25
33615 Bielefeld

March 22, 2006

Contents

Executive Summary

This report analyses Federated Search in the VASCODA context, specifically focusing on the existing TIB Hannover and UB Bielefeld search infrastructures. We first describe general requirements for a seamless integration of the two full-text search systems FAST (Bielefeld) and Lucene (Hannover), and evaluate possible scenarios, types of queries, and different ranking procedures.

We then proceed to describe a Federated Search infrastructure to be implemented on top of these existing systems. An important feature of the proposed federation infrastructure is that participants do not have to change their existing search and cataloging systems. Communication within the federation is performed via additional plugins, which can be implemented by the participants, provided by search engine vendors or by a third party. When participating in the federation, all documents (both full-text and metadata) stay at the provider side, no library document / metadata exchange is necessary.

The integration of collections is based on a common protocol, SDARTS, to be supported by every member of search federation. SDARTS is a hybrid of the two protocols SDLIP and STARTS. SDLIP was developed by Stanford University, the University of California at Berkeley, the California Digital Library, and others. STARTS protocol was designed in the Digital Library project at Stanford University and based on feedback from several search engines vendors. Additional advantages can be gained by agreeing on a common document schema, as proposed by the Vascoda initiative, though this is not a precondition for Federated Search.

Überblick

Dieser Bericht analysiert verteilte Suche (Federated Search) im VASCODA-Kontext, ausgehend von den Suchinfrastrukturen der TIB Hannover und der UB Bielefeld. Die Arbeit beginnt mit der Spezifikation grundsätzlicher Anforderungen für eine nahtlose Integration bestehender Volltext-Suchsysteme, im speziellen FAST Data Search (Bielefeld) und Lucene (Hannover), vergleicht deren Funktionalitäten und evaluiert mögliche Szenarien für den Einsatz der Systeme im Rahmen einer verteilten Suchinfrastruktur.

Der Bericht beschreibt eine verteilte Suchinfrastruktur, die aufbauend auf diesen bestehenden Systemen implementiert werden kann. Wichtig hierbei ist, dass alle Teilnehmer an dieser Föderation ihre bestehenden Such- und Katalogsysteme weitestgehend weiterverwenden können. Die Kommunikation innerhalb der Föderation erfolgt mittels zusätzlicher Komponenten, sogenannter Plugins, die durch den Suchmaschinen-Anbieter, den Teilnehmer selbst oder einem Drittanbieter implementiert werden können. Ein Austausch von Dokumenten / Metadaten zwischen den Teilnehmern ist hierbei nicht notwendig.

Die Integration der Dokumentsammlungen erfolgt über ein gemeinsames Protokoll, SDARTS, das von jedem Teilnehmer unterstützt wird. SDARTS setzt sich aus den zwei Protokollen SDLIP und SDARTS zusammen. SDLIP wurde von der Stanford University, der University of California at Berkeley, der California Digital Library und anderen entwickelt. Das STARTS Protokoll wurde im Digital Library Projekt in der Stanford University zusammen mit verschiedenen Suchmaschinenanbietern entwickelt. Die Nutzung eines gemeinsames Dokument / Metadatenschemas ist von Vorteil, aber keine Voraussetzung für verteilte Suche.

Chapter 1

Introduction

1.1 Vascoda Background and Motivation

The Vascoda strategy report[19] states the Vascoda vision as becoming "the corner stone for a common German digital library", the main goal being the integration of distributed scientific information in Germany and beyond.

The central Vascoda information portal currently provides a metasearch environment, which however does not meet the expectations of the customers who have become used to Google-like search engines and interfaces. In order to guarantee the success of the Vascoda project, the regulation board proposed to substitute the metasearch portal with up-to-date search engine technology. One possibility would be to build up an index of all Vascoda-relevant scientific content, based on the FAST Data Search search engine[1] operated by the HBZ. The HBZ intends to index about 60,000,000 documents[2] of about 300 collections, also and especially including the so called "Deep Web". While this is an ambitious project, it seems unrealistic that a single central provider will be able to index all information that is needed for a German Digital Library. In addition to the required capacity for such a central system, important publishers and content providers will not be able to make their data available for this central system because of licensing contracts they have with other publishers or providers. Also, since there already is a rich market of information providers as well as search engine systems, not every information provider will be able/interested to deal with the FAST Data Search engine used by HBZ.

[1] `http://www.fastsearch.com/`

[2] By *documents* we refer to documents, full-text or abstract, and their metadata annotation

Therefore, a cooperative environment built on an heterogeneous search infrastructure is preferrable. In such an *Federated Search* environment, the partners are able to keep their own search systems, and handle storage and indexing in a decentralized way, yet the results will look as if they were generated using a single centralized installation. The concept of such a federated search engine infrastructure is illustrated in Figure 1.1. Large boxes correspond to information providers, smaller embedded boxes represent search engines.

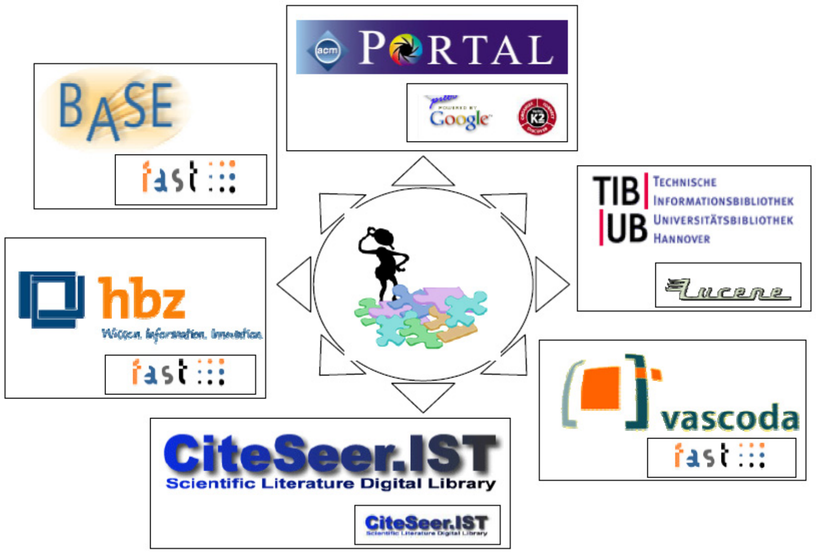

BASE
fast
PORTAL
Google
TIB UB
Technische Informationsbibliothek
Universitätsbibliothek Hannover
Lucene
hbz
Wissen. Information. Innovation.
fast
vascoda
fast
CiteSeer.IST
Scientific Literature Digital Library
CiteSeer.IST

Figure 1.1: How can we query all these document collections in a seamless way?

As the HBZ proposal did not make any arrangements for such a federated system [17], the Vascoda regulation board asked for examination of Federated Search in the Vascoda context [18]. Providing such a Federated Search infrastructure has the additional advantage that the participating institutions can then also integrate data from the central Vascoda system into their own information systems, e.g. in their subject portals or their own search engines.

Currently, two major members of Vascoda, the German National Library of Science and Technology (Technische Informationsbibliothek, TIB) and FIZ Technik, already have their own search engines based on the open-source Lucene search library [7, 9], a well-known alternative to commercial search engine software. Bielefeld University Library relies on a customized installation of Fast Data Search, focusing on search over OAI resources. All three members intend to keep and integrate their systems into a Federated Search Vascoda infrastructure.

This report discusses the requirements for such a Federated Search infrastructure, and focuses on Lucene- and FAST Data Search–based systems. It also provides suggestions for Federated Search in general. In the near future, each Vascoda project member will have to think about how to integrate other (especially commercial) scientific search engines, e.g. Google Scholar[3], Scirus[4], Scopus[5] etc., in order to provide a preferable portal for the end–user through the Vascoda federation.

1.2 Digital Libraries Background

Digital Libraries (DLs) have become an important part of educational, research and business processes. They provide up to date information to support decision making, serve as a source of learning materials and textbooks, and deliver patents and standard specifications. The success of the Open Archives Initiative (OAI), including the rise of academic repository systems for all types of digital information, provided an important starting point for the provision of scientific resources. Still, there is a strong need to include Web resources into scientic search scenarios, usually based on Web search engine technology.

Most DLs specialize in some area of interest: the Patent Documents collection of the TIB for example contains several million patent documents. Users access DLs via a Web-based query interface, one per library. The query interfaces run on top of local search engines, which can be custom-built for libraries, built upon open source projects or bought from search engine vendors. The vast majority of prospective users do not know precisely which DL interfaces they need to query in order to accomplish a particular task. Metasearch engines address this problem by providing a single unified query interface and combining search results from multiple DLs, usually implemented as a single, centralized metasearch broker. A system collects the final result lists from each underlying DL and re-ranks them to get a unified importance ordering of the results. To do this, it solely analyzes a list of search results, not the documents themselves nor any collection-wide statistics. Therefore, it cannot exploit the usual information retrieval measures which search engines apply internally. While this allows search even over uncooperative DLs, the re-ranking process unavoidably results in loss of rank quality.

[3]`http://scholar.google.com/`
[4]`http://www.scirus.com`
[5]`http://www.scopus.com`

A better approach is Federated Search. Here, brokers and search engines cooperate in order to deliver a common, consistent ranking, to provide search quality results compareable to search based on a single search engine. In general, this goal is not easy to achieve due to differences in search engine implementations and DL schemata. Factors like stemming algorithms, stop word lists, local collection statistics, ranking formulae, etc., should be unified across the federation. Usually, however, search engine companies do not provide detailed specifications of their products, citing intellectual property reasons.

In principle, a distributed search system using only one search engine brand could be developed relatively easy by the engine manufacturer, using proprietary protocols. Such a system is very valuable in a single-company search cluster (for example, Google has their own federated/distributed search infrastructure [2]). However, in a truly *federated* scenario, it is almost impossible and probably very undesirable to force every participant to use a single product, especially when another search system is already in use. In this case, the institutions have already invested much effort and money in customizing their systems, have defined complex workflows in order prepare documents for search, and of course, have their staff trained in the use of their system.

Therefore, instead of forcing libraries to use a completely new system, this reports describes a standardized Federated Search infrastructure where participants can continue using their existing search engines. Information exchange between these search engines is performed using open standard protocols. Technically, the approach is based on the use of *plugins*, providing the interface to different search engines, which we describe in detail in Section 4.3.1.

Specifically, we focus on enabling Federated Search over DLs like the TIB, which uses a search system based on the open-source search library Lucene, and the Bielefeld University Library, which uses the commercial FAST Data Search engine. The approach has to be scalable and new DLs should be easily connected to the infrastructure. The required search functionality includes both full-text and metadata search capabilities. In general, the approach is easily extendable to other search engines and to intranet search applications.

Combining the information from many DLs is a non-trivial task, which requires a common standard interface for query and result exchange. Instead of defining a new protocol, we re-use an existing protocol, SDARTS, which was proposed by Columbia University [11]. It is built on two complementary protocols for distributed search, namely SDLIP [13] and STARTS [10], which are described in detail in Chapter 2.

When integrating heterogeneous information, different data structures and entry formats used in different DLs cause additional problems. This is true for metasearch, for HBZ's proposal using a centralized system, and for Federated Search. The integration of available document schemas is therefore not a focus of this report, a solution could build on a common schema which has been specified in Vascoda [20] already.

Regarding document collections, this report considers text-oriented search over all covered documents, including document metadata (author, year of publication, abstract, etc.) and document full-text. When presenting result lists for queries, the shown documents should be ranked in the order of importance, the most relevant documents being presented first. Given that the Vascoda libraries already cooperate in various ways, we assume Vascoda participants will also cooperate to perform Federated Search and deliver appropriate information about ranking information for result merging. This substantially improves current metasearch systems, where ranking can only be performed in a very limited way due to lack of detailed statistics about documents and information sources.

1.3 Information Retrieval Background

1.3.1 Distributed Search Engine Architecture

The main unit in a distributed information retrieval environment is the broker, providing access to the distributed search engines. Users define a set of databases to be queried, the broker itself stores an appropriate set of statistics about every database/search engine participating in the search process. The broker receives queries from the user and propagate them to the available search engines, eventually restricting search to a subset of them, using database selection algorithms. The broker then collects, merges, reorganizes and displays results in a unified result list of ranked documents. Of course, the federation may be accessed by more than one broker, possibly having different functionalities and serving different user groups. A Federated Search architecture is shown in Figure 1.2.

The important notions and concepts relevant for distributed informational retrieval are described below.

1.3.2 Important Terms

Information retrieval deals with architectures, algorithms and methods used for performing search among internet resources, DLs, institutional repositories, and

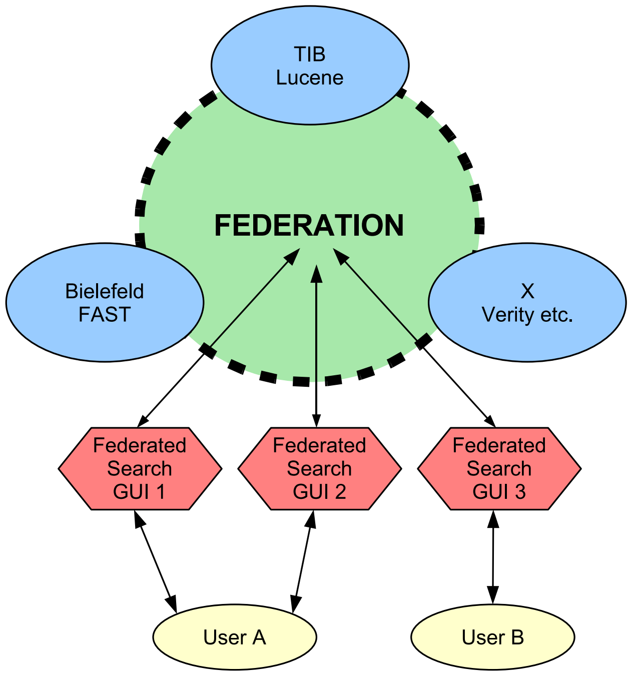


Figure 1.2: Federated Search Engine architecture

text and multimedia databases. The main goal is to find the relevant documents for a query from a collection of documents. The documents are pre-processed and placed into an index, a set of data-structures for efficient retrieval.

A typical *search engine* is based on the *single-database model*. In the model, the documents from the available document sources are collected into a centralized repository and indexed. The whole model is effective if the index is large enough to satisfy most of the user's information needs and if the search engine uses an appropriate retrieval system.

A *retrieval system* is a set of retrieval algorithms for different purposes: ranking, stemming, index processing, relevance feedback and so on. Many models are based on the *bag-of-words* model, which assumes that every document is represented by the words contained in the document. A well-known model extending *bag-of-words* is the *Vector Space Model*.

When search engine results are presented in a specific order, we call this a *result ranking*.

1.3.3 Distributed Information Retrieval

Distributed information retrieval appears when the documents of interest are spread across many sources. In this setup, it is possible to collect all documents on one server or establish multiple search engines, one for each collection of documents. The search process is then performed across the network, using distributed servers, this being a distinctive feature of distributed information retrieval.

Distributed search is based on the *multi-database model*, where several text databases are modelled explicitly. The multi-database model for information retrieval has many tasks in common with the single-database model, but also has to address additional problems [6], namely [6]:

- The resource description task;
- The database selection task;
- The query mapping task;
- The result merging task.

As these issues are the main driving factors for distributed information retrieval research, we briefly describe them below.

Resource description task

A full-text database provides information about its contents in a set of statistics, which is called "resource description" or "collection-wide information". It may include data about the number of specific term occurrences in a particular documents, in a collection, or a number of indexed documents. Resource description data is obtained during the index creation step. The availability of collection-wide information depends on the level of cooperation in the system. The STARTS standard [10] is a good choice for a Federated Search environment, where all search engines present their results in a unified resource description format.[7]

[6]We enlarged this list with the "Query mapping task"

[7]Still, when they are not willing to cooperate, one can infer some statistics from query-based sampling [15].

Database selection task

The collected resource descriptions are used for database selection and query routing tasks, as we are usually not interested in querying those databases which are unlikely to contain relevant documents. Rather, we want to be able to select only the relevant databases to our query, according to their resource descriptions. One simple way to do this is to ask the user to manually pick the set of interesting databases. Another option is to calculate automatically a "usefulness measure" of each database. Usually, this measure is based on the vector space model.

An intermediate approach between completely manual and automatic selection is a subject-oriented grouping of retrieval resources. Based on additional metadata information (by data providers or secondary sources) for the database descriptions, we can dynamically compose the set of relevant databases for each query. For example, the DBIS application of Regensburg University Library collects such information in a distributed way from the member libraries. Currently, this system includes about 5,000 resource descriptions contributed by 73 libraries.

Query mapping task

An important and difficult task is the handling of different document schemata and query languages. The document schema mostly depends on the coverage and origin of data. Some databases cover basic reference data only, while other resources contain enriched information with subject or classification information, abstract or additional description parts. Other databases contain both abstract and full-text.

In practice, we have to work with data of different quality. The same documents will have different representations in different database environments. For example, Anglo-American collections often lack European special characters and person names only cover first names in abbreviated form. In some databases, metadata is split into fine-grained parts: a title could contain several subtitles. Also, search engine products have many query languages, with different syntax and complexity. These facts influence the database structure, query and result representation. These differences call for upstream query transformation and a downstream result display reorganization, they are not a problem of distributed search per se, but rather a problem caused by lack of standard for data entry and description. Therefore, this problem cannot be solved purely by technical means (i.e. automatic mapping), but rather by agreements between the institutions participating in the federation.

Result merging task

The result merging problem arises when a query is performed over several selected databases and we want to create one single ranking out of these results. This problem is not trivial, since the computation of the similarity/importance scores for documents requires collection-wide statistics. The scores are not directly comparable among different collections and the most accurate solution can only be obtained by global score normalization, based on a a cooperative Federated Search environment.

Metasearch and Federated Search environments

An important property of distributed search engine is the degree of cooperation. We can divide distributed search infrastructures into two categories:

- Metasearch (uncooperative, isolated environment);
- Federated Search (cooperative, integrated environment).

Metasearch has no other access to the individual databases than a ranked list of document descriptions in the response to a query. Federated Search has access to document and collection-wide statistics like TF and DF (these notions are described in details in Subsection 3.1.2). The result rankings produced by metasearch strategies are thus less appropriate than Federated Search rankings.

In this report we assume that all collections in the proposed system provide the necessary statistics.

1.4 Report Overview

The remainder of this report is organized as follows. In Chapter 2, we describe the requirements for a Federated Search infrastructure. We start with a description of the existing SDARTS protocol and continue with a discussion of how to implement SDARTS within FAST Data Search and Lucene. In Chapter 3, we present details of document indexing and ranking algorithms. The implementation details for Lucene-based and FAST-search engines are given in Chapter 4. Chapter 5 contains results of a preliminary evaluation of our proposed Federated Search infrastructure using a first prototype. Chapter 6 provides a short summary of this report.

Chapter 2

Existing Components for Federated Search

In this chapter, we describe current efforts for a Federated Search infrastructure. In Section 2.1, we start with the analysis of several protocols for search interoperability in DLs context. Section 2.2 and Section 2.3 contain descriptions of the two most relevant protocols, SDLIP and STARTS, composing the SDARTS protocol for distributed search, which we have chosen as foundation for our federated search infrastructure. In Section 2.4, we provide some background information about the FAST Data Search search engine and the Lucene information retrieval library; their capabilities with respect to the STARTS protocol are analyzed in the Appendix.

2.1 Protocols for Distributed Search

Search in DLs is performed using online search interfaces. A common search interface is important, as it eliminates the need to enter the same query several times, once for every library of interest. This common interface should automatically convert a query into an appropriate format and send it to a number of selected DLs. It can also provide benefits like increased search coverage and a unified document ranking. Better usability is another advantage, as users only have to use a single graphical user interface.

The main challenge for such an interface is to guarantee proper query conversion and consistent merging of results from different DLs. A single query interface is often collection-specific. On various search engines, the queries may be executed quite differently, document relevance is computed using inhomogeneous statistics, and final result lists are incomparable. In the following we review several protocols addressing at least some aspects of federated search.

2.1.1 Z39.50-2003

Z39.50[3] is an ANSI Standard for querying different information systems via a unified interface, which is already used by a number of DLs. The Z39.50 ZDSR profile (http://www.loc.gov/z3950/agency/profiles/zdsr.html) is a specification similar to STARTS for merging homogeneous search engines (also see http://www.lub.lu.se/tk/demos/DGD97.html). However, ZDSR just specifies what STARTS does (while requiring the full payload of Z39.50), and therefore is just too complex for our task.

2.1.2 ZING

ZING, "Z39.50-International: Next Generation"[1], covers a number of initiatives by Z39.50 implementors to make the intellectual/semantic content of Z39.50 more broadly available. This protocol is more attractive to information providers, developers, vendors, and users because of lower implementation barriers, while preserving the existing intellectual contributions of Z39.50 accumulated over nearly 20 years.

Current ZING initiatives are SRW (including SRU), CQL, ZOOM, ez3950, and ZeeRex. Some (e.g. SRW/U) seek to evolve Z39.50 to a more mainstream protocol, while others (e.g. ZOOM) try to preserve the existing protocol but to hide its complexity. Most of these approaches are either too implementation-specific or not powerful enough with respect to Federated Search requirements.

For example, the Common Query Language (CQL)[2] is a formal language for representing queries to information retrieval systems such as web indices, bibliographic catalogs and museum collection information. The design objective is that queries are human readable and writable, and that the language is intuitive while maintaining the expressiveness of more complex languages. CQL tries to combine simplicity and intuitiveness of expression for simple every-day queries, with the richness of more expressive languages to accomodate complex concepts when

[1] `http://www.loc.gov/z3950/agency/zing/`
[2] `http://www.loc.gov/standards/sru/cql/`

necessary. It is not possible, though, to provide additional statistics with the query needed for homogeneous ranking.

2.1.3 OAI-PMH

The Open Archives Initiative Protocol for Metadata Harvesting [12] is another well-known and widely used protocol. It provides an application-independent interoperability framework based on metadata harvesting. The metadata may be in any format that is agreed upon by a community (or by any discrete set of data and service providers), although unqualified Dublin Core is specified to provide a basic level of interoperability. Metadata from many sources can be gathered together in one database, and services can be provided based on this centrally harvested (aggregated) data. The protocol is not useful for search activity, though. Since data should be stored in distributed indices and not be collected in one place, OAI-PMH does not help to provide federated search per se. It can help by unifying document schemata to a certain degree. Different search engines already started to integrate this information into their index (Google Scholar, Scirus, OAIster etc.)

2.1.4 SDARTS

The SDARTS protocol [11] is a hybrid of two separate protocols: the **S**imple **D**igital **L**ibrary **I**nteroperability **P**rotocol [13] (SDLIP) and the **Sta**nford Protocol Proposal for Internet **Ret**rieval and **S**earch [10] (STARTS).

SDLIP is a protocol jointly developed by Stanford University, the University of California at Berkeley and at Santa Barbara, the San Diego Supercomputer Center, the California DL, and others. SDLIP is an abstraction of lower-level metasearch operations. It proposes a layered, common query interface for many document collections, and also provides access to source metadata.

STARTS defines a protocol which enables search engines to provide necessary information for interoperability. This protocol was developed in the DL project at Stanford University and based on feedback from several search engine vendors. The protocol defines the set of statistics and metadata about document collections, which allows the creation of a consistent document ranking across several search engines.

SDARTS provides both the basic search operations from SDLIP and a set of statistics defined in STARTS, which are necessary for homogeneous result ranking. Another advantage is that a reference open source implementation of SDARTS is available, which makes it a very good choice for our federated search infrastructure. We therefore describe the SDLIP and STARTS protocols in more detail in Section 2.2 and Section 2.3.

2.2 SDLIP Protocol Description

2.2.1 SDLIP History

SDLIP stands for Simple DL Interoperability Protocol (SDLIP; pronounced S-D-Lip)[13][3]. It evolved from the CORBA-based DL Interoperability Protocol (DLIOP) in Stanford University's DL project. The protocol was developed jointly with the Universities of California at Berkeley and Santa Barbara, the San Diego Supercomputer Center (SDSC), and the California DL (CDL), trying to find a trade-off between a full-fledged search middleware like Z39.50, and the typical ad hoc search protocol for the Web.

SDLIP is used to request search operations over information sources, with special focus on scalability. Its design allows DL applications to run both on PC and handheld devices. The protocol is implemented in two variants, on top of CORBA or HTTP protocols. It supports both synchronous and asynchronous modes. In synchronous mode, the user waits until the full set of search operations is finished, while in asynchronous mode partial results can be accessed as soon as they come in.

2.2.2 Interfaces and Architecture

The main concept in SDLIP is the *Library Service Proxy* (LSP), which can be seen as a "broker". It has a back end and a front end. The back end is a socket for the collections to be connected, its implementation is specific for every single collection. The front end supports SDLIP and is used by clients to issue the queries to the sources. When a user poses a query, it is translated into SDLIP syntax and sent to selected DLs. Each library uses a corresponding implementation of the backend, for example a "FAST-SDLIP" proxy, "Lucene-SDLIP" proxy, and so on (assuming that such proxies are available).

The operations in SDLIP are grouped into three interfaces (see Fig. 2.1). The search interface receives the queries, the result access interface provides access to the set of result documents, the source metadata interface provides resource descriptions. This separation allows quite complicated interconnection setups. For

[3]See also `http://dbpubs.stanford.edu:8091/~testbed/doc2/SDLIP/`

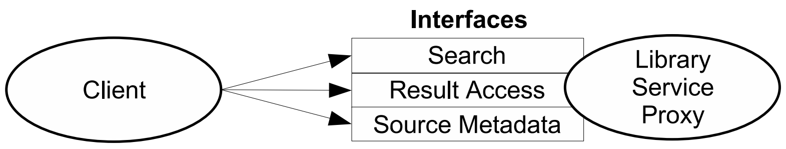


Figure 2.1: SDLIP Interfaces

example, the first resource can provide a search interface, while the result access interface can be implemented in second resource and after query execution the first resource delegates the responsibility to return results to the second resource. The protocol also supports a "state parking meter" notion, which is similar to "time-to-live", but is applied to the time of not deleting the previous search results at the collection side. This parameter can be used to control caching strategies. Also, there are server session ID and client request ID, which are used to distinguish between requests.

The main SDLIP implementation architecture is presented at Fig. 2.2. SDLIP aims at easy development of client applications, and construction of LSPs/brokers that wrap arbitrary sources. Only the client application and the LSP have to be implemented, the transport layer is provided by standard libraries. The transmission of queries and results is system-independent and can be done through CORBA, HTTP, or some other means. More details about the SDLIP protocol can be found in [13].

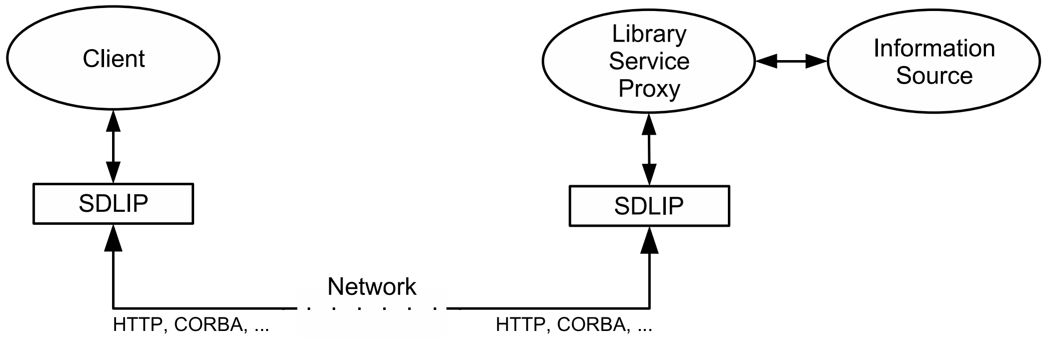


Figure 2.2: SDLIP Implementation Architecture

2.3 STARTS Protocol Description

The following section is based on the STARTS description in [10]. In the context of the Stanford DL project, a number of search engine companies and other big IT companies participated in the design of a protocol for search and retrieval of documents from distributed DLs. The key idea was that Stanford's database group could act as an unbiased party, it could analyze requirements from different search engine vendors and propose a balanced protocol, acceptable for all parties. This procedure was intentionally done at an informal level, in order to avoid time delays unavoidable for official standards.

Eleven companies participated in defining STARTS: Fulcrum, Infoseek, PLS, Verity, WAIS, Microsoft Network, Excite, GILS, Harvest, Hewlett-Packard Laboratories and Netscape. The Stanford group met individually with each company and discussed the possible solutions for each problem in detail. After the first draft of the STARTS proposal was released, two more revisions followed, using feedback from the companies and organizations involved. The revised draft was then discussed at a special workshop with all participants, where they discussed the remaining open issues and produced feedback for the final draft.

2.3.1 Architecture and Assumptions

The STARTS architecture can be applied to a large number of resources. Every resource may accomodate one or several document sources. The information how to contact each source is exported by the resource. In our setup, this means that every DL can have many subcollections; smaller portals can provide query interfaces to many single DLs. Every source has an associated search engine, number of documents per source is not limited.

STARTS can be compared with the Z39.50 [3] standard, which provides most of the STARTS functionality. But STARTS is much simpler and easier to implement. The protocol is designed for automatic communication between query interfaces, not for end users. It is simple in the sense that communications are sessionless, and information sources are stateless. The normal usage scenario assumes querying several sources, which requires two types of regular activity:

- Periodical extraction of the list of available sources from resources (to find out what sources are available for querying)
- Periodical extraction of metadata and content summaries from the sources

(to be able to decide, given a query, what sources are potentially useful for the query)

When the user submits a query, a Federated Search interface (or search broker) should perform the following:

1. Issue the query to a source at a resource (Source 1 in Fig. 2.3), possibly specifying other sources at the resource where to evaluate the query (Source 2 in Fig. 2.3);
2. Issue the query to other promising resources (serially, or in parallel to 1.).
3. Get the results from the multiple resources, merge them, and present them to the user.

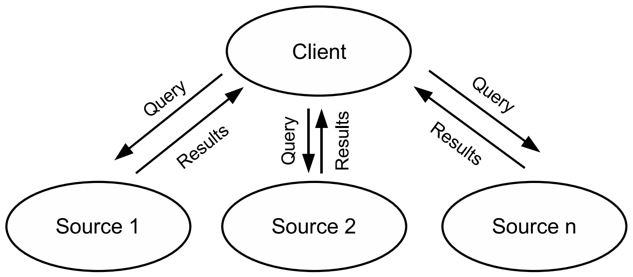


Figure 2.3: STARTS Distributed Search

The protocol defines how to query sources and what information the sources export about themselves. It does not describe an architecture for distributed search (in our case this is done by SDLIP). STARTS primarily describes which information is needed for distributed search, in order to retrieve and to combine results. The protocol keeps the common requirements for implementors to a minimum, while it enables optional sophisticated features for advanced search engines.

2.3.2 Query Language/Interface

Every search engine implementing STARTS has to support several basic features. First, any query should consist of two parts:

- a filter expression;

- a ranking expression.

A filter expression defines the documents that qualify for the answer, usually using boolean operators. The ranking expression associates a score to these documents, ranking them accordingly. For example, consider the following query with filter expression:

```
((author "Norbert Fuhr") and (title "Daffodil"))
```

and a ranking expression:

```
list((body-of-text "digital") (body-of-text "libraries")).
```

This query returns documents having "Norbert Fuhr" as one of the authors and the word "Daffodil" in their title. The documents that match the filter expression are then ranked according to how well their text matches the words "digital" and "libraries".

If the ranking expression is empty, the documents are ranked according to the parameters given in the filter expression. In principle, a filter expression can also be empty; then, all documents qualify for the answer. Sources also might only partially support such expressions, e.g. only filter expressions. Hence, they must indicate what type of expressions they support as part of their metadata.

The number of keywords in both the filter and the ranking expressions is not limited by STARTS. Terms can be combined with "AND" and "OR" operators. The ranking expressions also can use "LIST" operator, for grouping a set of terms. The terms in a ranking expression may have an associated weight, indicating their relative importance in the ranking expression. Both types of expressions are described in more detail later in this section.

L-strings

In STARTS an *L-string* is defined as either a string (e.g., `"Norbert Fuhr"`, or a string qualified with its associated language or language and country. The language–country qualification follows the format described in standard RFC 1766[1]. For example, the L-string `[en-US "handy"]` states that the string "handy" is a word in American English. The L-string is a Unicode sequence encoded using UTF-8, so all relevant languages and character sets are supported.

Atomic terms

When an L-string is enriched with attributes, e.g. `(author "Norbert Fuhr")`, it becomes a term. Attributes can be of two types, *field* or *modifier*. For example, in the expression `(date-last-modified > "2003-10-01")`, the string `date-last-modified` is a field and > is a modifier. STARTS defines the "Basic-1" set of attributes. This list has several predecessors, some of its field were defined in Z39.50-1995 Bib-1 use attributes, and later extended in GILS attribute set.

Fields

The *fields* correspond to a metadata or a body fields, associated with the term, e.g. `author`, `year`, etc. Each term can have one associated field, if the field assignment is empty, term is associated with `any` field. Fields are divided into *supported* and *optional*. Supported fields must be recognized by any source, while interpretation can still vary among sources. We listed STARTS fields in Table 2.1

Field	Required	Description
Author	yes	
Body-of-text	no	
Document-text	no	For relevance feedback.
Date/time-last-modified	yes	Formatted according to the International Standard ISO 8601 (e.g., "1996-12-31").
Linkage	yes	URL of the document.
Linkage-type	no	MIME type of the document.
Cross-reference-linkage	no	List of URLs in document.
Language	no	The language(s) of the document, as a list of language tags as defined in RFC 1766. For example, a query term (language "en-US") matches a document with value for the language field "en-US es". This document has parts in American English and in Spanish.
Free-form-text	no	A string (maybe representing a query in some query language not in the protocol) that the source somehow knows how to interpret.

Table 2.1: Fields

Modifiers

The *modifiers* indicate, what values are represented by a term, e.g. treat the term as a stem, as its word's phonetics (soundex) representation etc. Zero or more modifiers can be specified for each term. In the STARTS the set of modifiers corresponds to the Z39.50 "relation attributes", see Table 2.2.

Modifier	Default Value
<, <=, =, >=, >, !=	If applicable, e.g., for fields like "Date/time-last-modified", default: =
Phonetic (soundex)	no soundex
Stem	no stemming
Thesaurus	no thesaurus expansion
Right-truncation	the term "as is", without right-truncating it
Left-truncation	the term "as is", without left-truncating it
Case-sensitive	case insensitive

Table 2.2: Query Modifiers

Complex filter expressions

For building complex *filter expressions* from the terms, STARTS uses simple filter expressions, listed in Table 2.3. The first three operators are well-known Boolean operators. The *PROX* operator is also widely used in information retrieval tasks. For example, consider two terms `t1` and `t2` and the following filter expression:

AND
OR
AND-NOT
PROX, specifying two terms, the required distance between them, and whether the order of the terms matters.

Table 2.3: Filter Expressions

```
(t1 prox[3,T] t2)
```

The documents that match this filter expression contain `t1` followed by `t2` with at most three words in between them. "T" (for "true") indicates that the word order matters (i.e., that `t1` has to appear before `t2`).

Complex ranking expressions

The *ranking expressions* from the Table 2.4 are used for building complex ranking expressions from the terms. If a source supports ranking expressions, it must support all these operators. Each term in a ranking expression may have a weight (a real number between 0 and 1) associated with it, indicating the relative importance of the term in the query.

Global settings

The information to be supplied with a query is presented in Table 2.5.

AND
OR
AND-NOT
PROX, specifying two terms, the required distance between them, and whether the order of the terms matters.
LIST, which simply groups together a set of terms.

Table 2.4: Ranking Expressions

Field	Description	Default value
Drop stop words	Whether the source should delete stop words from the query or not.	Drop the stop words
Default attribute set	Default attribute set used in the query, optional, for notational convenience.	Basic-1
Default language	Default language used in the query, optional, for notational convenience, and overridden by the specifications at the L-string level.	No default
Additional sources	Sources in the same resource, where to evaluate the query in addition to the source where the query was submitted.	no other source
Returned fields	Fields returned in the query answer	Title, Linkage
Sorting fields	Fields used to sort the query results, and whether the order is ascending (“a”) or descending (“d”).	Score of the documents for the query, in descending order.
Documents returned Min	Minimum acceptable document score	No default
Documents returned Max	Maximum acceptable number of documents	20 documents

Table 2.5: Information propagated with a query

Result Merging

After receiving a query, the source reports the number of documents in the result set. Since the source might modify the given query before processing it, the source also reports the query that it actually processed. Besides that, for every item in the result set, the source provides the information presented in Table 2.6:

Optionally, a set of test queries can be specified. The results for these queries from different sources can be used for “calibrating” the final scores. The mechanism for handling duplicates is not static in STARTS, one has the choice to present only one document in the result or several identical documents having the same URL.

2.3.3 Source Metadata

In order to only select the most relevant sources for the query, we need additional information. With respect to this task, each source exports two types of information: metadata-based description (a list of metadata attributes) and statistics-based

Field	Description
Score	The unnormalized score of the document for the query
ID (of source)	The id of the source(s) where the document appears
TF (for every term)	The number of times that the query term appears in the document
TW (for every term)	The weight of the query term in the document, as assigned by the search engine associated with the source, e.g., the normalized TFxIDF weight for the query term in the document, or whatever other weighing of terms in documents the search engine might use
DF (for every term)	The number of documents in the source that contain the term, this information is also provided as part of the metadata for the source
DSize	The size of the document in bytes
DCount	The number of tokens, as determined by the source, in the document

Table 2.6: Information propagated with a result

description (a content summary of the source). The resource also exports resource descriptions with metadata for the covered sources.

Source metadata attributes

The information about every source is wrapped in metadata fields, which we list in Table 2.7. These source metadata attributes show the source capabilities, i.e. whether it supports the required functionality or not. This information is also helpful for reformulating the query according to a specific implementation of a source's search engine.

Field	Required	Description
FieldsSupported	yes	What optional fields are supported in addition to the required ones. Also, each field optionally is accompanied by a list of the languages that are used in that field within the source. Required fields can also be listed here with their corresponding language list.
ModifiersSupported	yes	What modifiers are supported. Also, each modifier is optionally accompanied by a list of the languages for which it is supported at the source. Modifiers like stem are language dependent.
FieldModifierCombinations	no	What field-modifier combinations are supported. For example, stem might not be supported for the author field at a source.
QueryPartsSupported	no	Whether the source supports ranking expressions only (“R”), filter expressions only (“F”), or both (“RF”). Default: “RF.”
ScoreRange	yes	This is the minimum and maximum score that a document can get for a query; we use this information for merging ranks. Valid bounds include -infinity and +infinity, respectively.
RankingAlgorithmID	yes	Even if we do not know/understand the actual algorithm used, it is useful to know that two sources use the same algorithm.
TokenizerIDList	no	E.g., (Acme-1 en-US) (Acme-2 es), meaning that tokenizer Acme-1 is used for strings in American English, and tokenizer Acme-2 is used for strings in Spanish. Even when we do not know how the actual tokenizer works, it is useful to know that two sources use the same tokenizer.
SampleDatabaseResults	yes	The URL to get the query results for a sample document collection.
StopWordList	yes	
TurnOffStopWords	yes	Whether we can turn off the use of stop words at the source or not.
SourceLanguage	no	List of languages present at the source.
SourceName	no	
Linkage	yes	URL where the source should be queried.
ContentSummaryLinkage	yes	The URL of the source content summary; see below.
DateChanged	no	The date when the source’s metadata was modified last.
DateExpires	no	The date when the source metadata will be reviewed, and therefore, when the source metadata should be extracted again.
Abstract	no	The abstract of the source.
AccessConstraints	no	A description of the constraints or legal prerequisites for accessing the source.
Contact	no	Contact information of the administrator of the source.

Table 2.7: Source metadata attributes

Source content summary

Another type of information about the source is the *content summary*. This kind of data is used to decide the degree of source relevance to the query. It is also crucial for normalizing the relevance scores of results and making them comparable across many sources. The required information is presented in Table 2.8. The recommended default parameters for content summary creation are: the words listed are not stemmed, stop-words are not removed, the words are case sensitive, each word is accompanied by corresponding field information.

Field	Description
Stemming	on/off
Stop-words	included/not included
Case sensitive	on/off
Total number of documents in source	
List of words	List of words that appear in the source.

Table 2.8: Source content summary

The list of words that appear on the source also has several subfields per every word, see Table 2.9.

Field	Description
Index field	Index field corresponding to where in the documents they occurred, e.g., (title “databases”), (abstract “databases”), etc.
Language	In addition, the words might be qualified with their corresponding language, e.g., [en-US “behavior”].
Number of postings	Total number of postings for each word, i.e., the number of occurrences of a word in the source.
DF	Document frequency for each word, i.e., the number of documents that contain the word.

Table 2.9: Information provided for each word in content summary word list

Resource definition

The resource is in charge for exporting the information from the contained sources. It should provide a list of provided sources and URLs to metadata attributes. This information is sufficient for brokers to prepare the query, retrieve and merge the results.

2.4 FAST and Lucene: General Description

2.4.1 FAST Data Search

FAST Data Search is a commercial software system by the Norwegian company Fast Search & Transfer (FAST) with a sound history as a competitive solution on the rapidly grown search engine market. Their software was the foundation for the AllTheWeb search engine (www.alltheweb.com). The core of the FAST system contains a full-text engine accompanied by a broad set of the Web-oriented search engine features and tools. It is designed to accomodate the typical search engine characteristics: speed, linguistic features and ranking methods which fulfill users needs for result weighting. The ranking functionality is a configurable set of values with some hidden algorithms and parameters behind.

FAST software plays an important role in the Vascoda project because major Vascoda members use FAST as their basic system. Besides, some prospective Vascoda member institutions will use FAST to present their portal or database–oriented activities. Vascoda is Web-based and there is a strong need to integrate search engine functionality into the main and subject portals. This covers search engine functionalities, like advanced user interfaces, fast retrieval and result display, crawling of scientific information (e.g., based on the common link lists) and a high quality ranking.

FAST interfaces and APIs have to be further investigated. This includes low-level interfaces like direct HTTP calls, XML-based HTTP communication and Web service based architecture. Apart from the technical layer, the ideal scenario of integrating FAST resources with external search environments should cover all FAST search engine features with focus on the topic-based ranking support.

STARTS functions can be delivered either by using FAST API functions (supported: HTTP GET, C++, Java, .NET) or by specific scripts for analyzing internal resources. A complete comparison between STARTS and FAST capabilities is given in Appendix A.

2.4.2 Lucene

Lucene is an information retrieval library, which was created in 1997 by Doug Cutting, an experienced search engine developer. In 2000, Lucene became an open-source project and was released at SourceForge; since 2001, Lucene is an official Apache project. Over time, several active developers joined Cutting's initiative, now providing a solid developer and user base. Originally written in Java, Lucene has also been ported to programming languages like C++, C#, Perl, and Python. Major advantages of Lucene are its simplicity, efficiency, well-written code and good documentation. Recently, the book "Lucene in Action" [9] has been published, which makes it easy to understand Lucene core functionality and gives a good overview of its additional capabilities.

The application area of Lucene is wide, it is used for search on discussion groups at Fortune 100 companies, bug tracking systems, and Web search engines (such as Nutch[4]). As any other open source product, it can be extended by everyone, a feature especially appreciated by developers. This is also important for us, since we are building evolving infrastructure and need to keep control over the code, to be able to extend federated search capabilities in future. The Lucene license is very unrestrictive, its integration does not require any fees to be paid to the original authors, for example. In addition, the Lucene mailing list acts as a free source of technical support. Meanwhile, several companies now also offer enterprise-level support for Lucene.

A proof-of-concept implementation of SDARTS for Lucene has been released by the SDARTS authors[8], although this implementation is somewhat outdated now. The implementation provides an SDLIP-to-Lucene query translator, means for maintaining (or automatically generating) collection-wide information, and for querying and retrieving documents. The version is based on a very early version of Lucene, and still requires significant effort to make it work with recent (recommended) Lucene versions.

A complete comparison between STARTS and Lucene capabilities is given in Appendix A.

[4]`http://lucene.apache.org/nutch/`

Chapter 3

Query Processing and Ranking

This chapter addresses search engine developers and people with information retrieval background. We discuss document ranking in federated search, possible problems and solutions. In Section 3.1, we describe necessary components of the document indexing process. Starting with document pre-processing steps, we then briefly introduce two most appropriate retrieval models and continue with a description of query processing in distributed search. In Section 3.2, we describe a general distributed search scenario, the possible ranking inconsistencies and problems are discussed in Section 3.3.

3.1 Document Indexing and Query Processing

3.1.1 Document Pre-processing

Natural languages inavoidably contain information which is redundant or overspecified for search. This includes punctuation, filler words or groups of words with the same basic meaning. Thus, in a preparation step prior to indexing, input text consisting of *words* is being converted into a stream of *tokens*, this is called *tokenization*; such a stream is the consolidated form of continuous text. A *term* basically is a token, but with the addition of a *field* (or zone), such as "author = John Doe" would map to "author:John author:Doe".

The conversion step usually prepares the text in the following ways:

- **Stopwords.**
  The most frequent words in a language, like "the", "and" or "is" in English, do not have rich semantics. They are called "stopwords" and get removed

from the document representation. The set of specified stopwords is usually stored in a list, it can also be determined after a first indexing step by counting the number of occurrences.

- **Stemming.**
  Word variations with the same stem like "run", "runner" and "running" are mapped into one term, corresponding to a particular stem, a stemming algorithm performs this process. In this example the term is "run". There are several algorithms available, most of them being focused on one language and, moreover, incompatible to each other, as they might produce different terms for the same input.

- **Anti-Phrasing.**
  Besides a simple stop words detection, many search engines, including FAST Data Search, provide extensive Natural Language Support, in particular a so-called anti-phrasing function. Based on a standard dictionary supporting several languages, this function removes redundant phrases from the query. Those phrases are parts of a query that do not contribute to the query's meaning, such as "Where do I find ..." or "I am searching everything about ...".

The same preparation steps are performed on a user's query, again transforming a list of words into a list of terms.

3.1.2 Retrieval Models

An important characteristic of an information retrieval system is its underlying retrieval *model*. This model specifies the procedure of estimation the probability that a document will be judged relevant. The final document ranking being presented to the user is based on this estimate.

Vector Space Model

The *Vector Space Model* is most widely used in search engines. In this model, a document D is represented by the vector $\vec{d} = (w_1, w_2, \ldots, w_m)$, where m is the number of distinct terms, and w_i is the weight indicating the "importance" of term t_i in representing the semantics of the document. For all terms that do not occur in the document, corresponding entries will be zero, making the full document vector very sparse.

If a term occurs in the document, two factors are of importance in weight assignment. The first factor is the term frequency (TF). It is the number of the term's occurrences in the document.[1] The term weight in the document vector is proportional to TF. The more often the term occurs, the greater is its assumed importance in representing a document's semantics.

The second factor affecting the weight is the document frequency (DF). It is the number of documents containing a particular term. The term weight is proportional to the inverse document frequency IDF. The IDF measure is computed as the logarithm of the ratio N/DF, where N is the overall number of documents in the collection. The more frequently the term appears in the documents, the less is its importance in discriminating between the documents containing the term from the documents not having it.

A common standard for term weighting is the combination of TF and IDF, namely the TFxIDF product and its variants. A simple query Q (being a set of keywords) is also transformed into an m-dimensional vector $\vec{q} = (w_1, w_2, \ldots, w_m)$ using the same preparation steps applied to the documents. After creation of $\vec{q}$, a similarity measure between the query's vector and all documents' vectors $\vec{d}$[2] is being computed. This estimate is based on a similarity function, which can be the Euclidian distance or the angle measure inside the vector space. The most popular similarity function is the cosine measure, which is computed as a scalar product between $\vec{q}$ and $\vec{d}$.

For our Federated Search infrastructure, we will use the Vector Space Model. It is effective, simple and well-known in the DL community. For developers who are interested in advanced models, we also provide details for an alternative Language Modeling approach:

Language Modeling-based Model

Another popular approach that tries to overcome a heuristic nature of term weight estimation comes from the so-called probabilistic model. The *Language modeling* approach [14] to information retrieval attempts to predict a probability of a query generation given a document. Although details may be different, the main idea can be described as follows: every document is viewed as a sample generated from a special language. A language model for each document can be estimated during indexing. The relevance of a document for a particular query is formulated

[1]In information retrieval terminology, the term "frequency" is used as a synonym for "count".

[2]At least for those documents which contain all required terms at least once

as how likely the query is generated from the language model for that document. The likelihood for the query Q to be generated from the language model of the document D is computed as follows [16]:

$$P(Q|D) = \prod_{i=1}^{|Q|} \lambda \cdot P(t_i|D) + (1-\lambda) \cdot P(t_i|G) \tag{3.1}$$

Where:
t_i — is the query item in the query Q;
$P(t_i|D)$ — is the probability for t_i to appear in the document D;
$P(t_i|G)$ — is the probability for the term t_i to be used in the common language model, e.g. in English;
λ — is the smoothing parameter between zero and one.
The role of $P(t_i|C)$ is to smooth the probability of the document D to generate the query term t_i, particularly when $P(t_i|D)$ is equal to zero.

3.1.3 Distributed Query Processing

A schema for query processing in distributed information retrieval scenario is presented on Figure 3.1 [5].

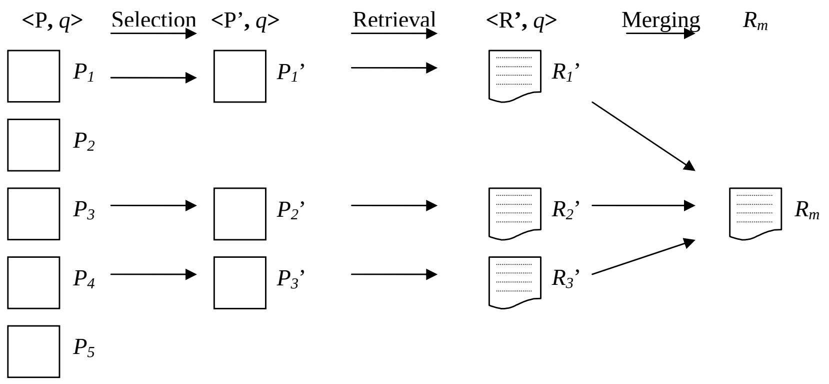


Figure 3.1: A query processing scheme in distributed search

A query q is posed on the set of search engines that are represented by their resource descriptions P_i. A search broker selects a subset of servers P', which are the most promising ones to contain the relevant documents. The broker routes the query q to each selected search engine P'_i and obtains a set of document rankings

R' from the selected servers. In practice, a user is only interested in the best "top-k" results, where k usually is between 10 and 100. For this, all rankings R'_i are merged into one rank Rm and the top-k results are presented to the user. Text retrieval aims at high relevance of the results at minimum response time. These two components are translated into the general issues of effectiveness (quality) and efficiency (speed) of query processing.

3.2 Distributed Ranking

3.2.1 Scenario

For a simple, yet easily extendable scenario, assume two document collections A and B. They do not contain duplicate documents and each is indexed by its own search engine. Both search engines have identical retrieval systems, i.e. identical stemming algorithms, stopword lists, metadata fields supported, query modifiers and Term Frequency (TF) - Inverted Term Frequency (IDF) ranking formulae:

$$SimilarityScore = \sum_{i=1}^{|q|} TF_i \cdot IDF_i \tag{3.2}$$

$$TF = \frac{TO}{|D|} \tag{3.3}$$

$$IDF = \log \frac{N}{DF} \tag{3.4}$$

TO – the number of term occurences in the document
$|D|$ – the document length measured in terms
N – the number of documents in the collection
DF – the document "frequency", the number of documents with the term.

As a simple query, the user poses a query consisting of two keywords q_1 and q_2. It is not specified, whether they should appear in metadata or in the document's text body. We also assume that environment is cooperative and we can obtain any necessary information from any collection. Our goal is to achieve the same result ranking in the distributed cases as produced by the same search engine on the single collection C, which contains all the documents from A and B. Since the retrieval systems are identical, TF values are directly comparable on both

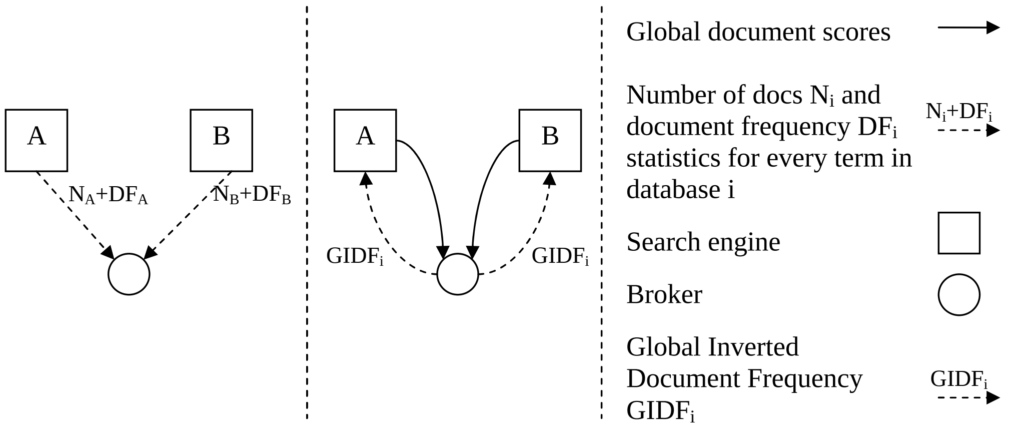

A
B
N_A+DF_A
N_B+DF_B
A
B
GIDF_i
GIDF_i
Global document scores
Number of docs N_i and document frequency DF_i statistics for every term in database i
N_i+DF_i
Search engine
Broker
Global Inverted Document Frequency GIDF_i
GIDF_i

Figure 3.2: Statistics propagation for result merging

resources. For collection-dependent statistics N and DF, we compute the global IDF, $GIDF$, and use the following normalized ranking formula:

$$DistributedSimilarityScore = \sum_{i=1}^{|q|} TF_i \cdot GIDF_i \tag{3.5}$$

$$GIDF = \log \frac{N_A + N_B}{DF_A + DF_B} \tag{3.6}$$

For this setup, the distributed similarity score in 3.5 is equal to the similarity scores computed on single collection C.

Most notably, the necessary N and DF values need only to be computed once (preferably according to the scheme from [4]), before query time, and regulary after changes in the collections (document additions, deletions, and updates). The communication flow for such an aggregation is presented on Fig 3.2. During query execution, the search engines compute results with comparable scores, since they use common global inverse document frequency $GIDF$, which is sent with the query. The global ranking is then achieved by merging each sub-results list in descending order of global similarity score.

3.3 Additional Issues in Distributed Ranking

In the cooperative environment, where all search engines provide necessary statistics, we can achieve the consistent merging as produced by a non-distributed system, also known as the perfect merging. In practice, it is difficult to guarantee exactly the same ranking as that of the centralized database with all documents from all databases involved. We enlarged the list from [10] of several issues, which can reduce search quality:

- Some relevant documents are missed after the database selection step;
- Database selection may be poor, if required statistics are not provided;
- Some collections may not provide N and DF values for normalization;
- Different stemmers influence both TF and IDF values;
- Different stopword lists influence TF and IDF values;
- Overlap affects globally computed IDF values;
- Query syntaxes may be incompatible;
- Unknown features of the proprietary algorithms cannot be removed;
- Document schemata (Metadata fields etc.) on resources do not match.

Another point to be considered is the case when several distinct documents yield the same similarity score. Whenever there is an additional overlap between collections, document duplicates may occur at not directly succeeding positions of the ranking. A viable solution to overcome this problem is to not only take the score as the only ranking criterion, but to additonally sort documents with the same score by their document identifier (URI, etc.).

In chapter 2, we defined specifications which are recommended for DLs in a Federated Search infrastructure. If the requirements from the specification are satisfied by the participating libraries, the system will produce the result ranking as described in general scenario in Section 3.2.

Most of aforementioned problems are relevant for both full-text and metadata search. Therefore, the solutions are also applicable to both types of search. The only special problem, which occurs in metadata search is a problem of collection-schema mapping. There is no universal solution for it so far, but the most important set of fields is defined in STARTS protocol as mandatory for all participants. Additional search fields besides this set are optional and can be queried as well, but the system does not guarantee a perfectly consistent ranking for them.

Chapter 4

Combining Lucene and FAST

In this chapter, we describe search engines based on Lucene and FAST Data Search, with respect to Federated Search. In Section 4.1, we present the details of Lucene-based search engines. A preliminary implementation of an interface to the FAST index is described in section 4.2.1. Section 4.3 describes the integration of Lucene and FAST search engines into the federation.

4.1 Distributed Search with Lucene

4.1.1 Background

Lucene notably differs from other systems like FAST Data Search because it is *not* a complete search engine. In fact, it is a programming library, which allows the easy creation of a search engine with desired properties and functionality. Lucene has been implemented in several programming languages, we are considering its original (and main-line) Java implementation.

Lucene provides many required core components for search, in a high-quality, well performing and extensible way. Functionalities not directly related to search, such as crawling, text extraction and result representation, are not directly focused by Lucene. Even though a few such examples exist, it is still up to the developer to correctly provide data for building up the index and to correctly retrieve and display results from a search, using additional components. While it is reasonably easy to set up a small application for full-text search, for a full understanding of the library and especially for creating high-performance search applications good technical skills as well as consolidated knowledge in information retrieval are recommended.

In table 4.1, we define the steps to be considered for indexing with Lucene in general, and within our Federated Search scenario.

Stage Description	**Our Setup**
1. Specify what should be indexed (plain full-text, text with metadata,binary data etc.)	We index full-text with additional metadata fields. Some metadata is expressed by tokenized text, some other by keywords.
2. Specify what kind of queries should be supported (term queries, phrase queries, range queries etc.)	We want to support at least following query types: terms, phrases, boolean clauses.
3. Specify the index schema (name the fields, define the indexing strategy and specify how to translate input data into terms.)	Our input data consists of several *document collections*, which can be accessed separately. Field names and values are defined according to a fixed schema and the values are pre-processed. For term creation, each portion of the document (full-text or metadata field), its text contents only have to be split into tokens separated by whitespace. Each token is then assigned to the current field to form a term.
4. Specify how search will be performed (query formulation, result display etc.)	Searches are performed via the SDARTS API (queries passed to Lucene, results passed back to the caller). SDARTS queries have to be converted into the Lucene API counterpart, Lucene results back to SDARTS format.
5. Specify how and where the index should be stored (one index, several distributed ones etc.)	Distribution on physical level is not necessary; each index can be stored on one hard disk. In case of performance issues, we can consider partitioning and distribute them over multiple servers.

Table 4.1: Lucene Indexing Stages

Depending on these specifications, the complexity of the setup can be estimated as well as the required components. Our scenario implies a medium complexity level of the Lucene installation (large number of documents, full-text and metadata search with simple query types).

Based on these requirements, the developer then has to implement the following components in addition to using the Lucene standard API:

1. A custom Parser and Translator for importing documents into Lucene.
2. A Lucene representation of document collections (for indexing and search.)
3. The SDARTS wrappers for Lucene (probably re-using parts from the SDARTS distribution.)

4.1.2 Core Concepts for Indexing within Lucene

Lucene supports a broad scope of how to prepare data for full-text search. A central notion is the *Document*, which is a container for both indexed/searchable and stored/retrievable data. A *Document* may consist of several fields, each of them containing text, keywords or even binary data. A *Field* may be indexed into terms or simply be stored for later retrieval. A *Term* represents a token (word, date, email address etc.) from a text, annotated with the corresponding *Field* name.

The *Documents* are converted field-wise into terms by an *Analyzer*. This covers tokenization, converting to lowercase, stemming/lemmatization, stop-word filtering, etc. By using a custom *Analyzer*, the developer has full control over the term creation process.

The *Documents* can be made searchable by writing them into an index stored in a *Directory* (harddisk- or memory-based), using an *IndexWriter*. Internally, Lucene creates very compact index structures upon the given terms, such as inverted term-vector indices (in order to find documents similar to another one) as well as positional indices in order to support phrase queries. The TF and DF values are computed and stored as well as a general normalization factor for each *Document*, which, for example, contains a boost value for additional up- or down-ranking specific results (like PageRank).

The *Document* updates are only supported indirectly, by deleting the old *Document* and adding a new one. A deletion is performed in two steps. First, the *Document* is *marked* as deleted (search will indeed keep ranking using the old N and DF values, but simply ignore such documents in the result set). Upon request or when enough documents are marked as deleted, the index is re-created without these deleted *Documents*.

This re-creation step is very fast, since the index itself consists of several *Segments* of *Documents*. Each *Segment* basically is a small, immutable index. New *Documents* are added as one-*Document Segments* and are then merged with other *Segments* to a bigger one. The way how merging is performed can be configured and heavily influences indexing performance. Search performance can be improved by merging all *Segment*s to one single *Segment* again.

4.1.3 Search in Lucene

Lucene not only provides means for centralized search, but also for a distributed setup using the same concepts. Its rich API has a simplistic definition of a *Searchable* collection, providing methods for retrieving:

- the number of documents in the collection (N);
- the document frequency (DF) of a given *Term* in the collection;
- the search results *Hits*, where *Hit* is internal, numerical *Document id* plus score for a given query, eventually sorted and filtered to given criteria;
- the stored document data for a given *Document id*.

Such a *Searchable* can simply be a local index (accessible by an *IndexSearcher*), a single index on a remote system (providing a *RemoteSearchable*), or a combination thereof, which may also be combined again, and so on. There are two ways of combining *Searchers*, using the *MultiSearcher* and the *ParallelMultiSearcher*. The former performs a search over *Searchables* one after each other, whereas the latter queries all collections in parallel[1].

The different query types (term-, phrase-, boolean-query etc.) are represented as derived forms of an abstract *Query*, such as *TermQuery*, *PhraseQuery*, *BooleanQuery*, *RangeQuery* and so on, which may be aggregated (e.g., a *BooleanQuery* may contain several other queries of type *TermQuery* as well as *BooleanQuery* etc.).

Any search, distributed or on a single index, consists of the following steps:

- Specify the TFxIDF normalization formula (*Similarity*). Usually, Lucene's default implementation, *DefaultSimilarity* is used.
- Specify the *Query*, eventually being a complex composition of several others. A *QueryParser* can transform query strings into this representation.
- Rewrite the *Query* into more primitive queries optimized for the current index and compute boost factors.
- Create the query *Weight* (normalization factor based upon the query and the collection)[2]
- Retrieve the *Scorer* object, which is used to compute ranking scores, based upon the query weight and the *Searchable*'s similarity formula[3].

[1]While the idea of querying collections serially may sound inefficient, it may be used for search scenarios where only a certain maximum number of results from a specific list of prioritized collections is required. Still, in our scenario we will prefer the parallel implementation.

[2]Depending on the structure of the *Query*, this may be an aggregate of several sub-*Weight*s.

[3]As with the *Weights*, this may be an aggregate of several sub-*Scorers*

- Collect *Hits* object from the *Searchable*, sorting them in descending order according to the score computed by the given *Scorer*.
- Provide access to the stored *Document* data for the collected *Hits*, using the *Document id* as the key.

It is remarkable that Lucene clearly distinguishes between remote search and parallelized search facilities. The latter can also be useful on a single server when multi-threading is available. On the other hand, Lucene's remote search feature currently only exists as a Java RMI-based implementation, a rather Java-centric standard for remote method invocation, which may connect to any other *Searchable* on another server. However, the creation of another interface is simple (for SOAP, only a few *Stub* classes have to be created.)

Lucene's *MultiSearcher* implementation does not need to care about all these details. At instantiation time, a *MultiSearcher* sets up very little collection-wide information — a mapping table for converting sub-*Searchers'* (local) *Document id*s to a global representation. As Lucene's *Document ids* are integer numbers, and guaranteed to be gap-less (apart from documents marked as deleted), this is coded as offsets derived from the maximum document id of each source.

At query preparation time, the *MultiSearcher* retrieves the local DF values for all terms in the query and aggregates them to a set of global document frequencies. This global DF, along with the global maximum document id, is then passed to all sub-*Searchers* as the *Weight*.

Currently, there seems to be a small conceptual mistake in this implementation, as the DF values get refreshed for each distinct query, whereas the local-to-global id mapping is set up only once at instantiation time. While this saves some setup time, as DF values only need to be computed as necessary, any change (update/deletion of documents) in the downstream collections requires the creation of a fresh *MultiSearcher*. What is currently missing is the interaction between *MultiSearcher* and its sub-*Searchers* in such cases.

Also, the ranking formula used in Lucene by default (*DefaultSimilarity*) differs a little bit from the one shown in 3.2:

$$
\begin{aligned}
score(q,d) \;=\; & \sum_{(t\;in\;q)} (\sqrt{TF_{t\;in\;d}} \cdot (\log \frac{N}{(DF_t+1)}) + 1.0) \cdot boost(t.field\;in\;d) \cdot \\
\cdot\; & \sqrt{|terms \in t.field\;in\;d|}^{-1}) \cdot \\
\cdot\; & \frac{|terms \in q \cap d|}{|terms \in q|} \cdot \\
\cdot\; & \sqrt{sumOfSquaredWeights\;in\;q}^{-1}
\end{aligned}
$$

Besides looking a little bit more complex, it does obey the same principles, with the notable difference that the square-root of DF is used. However, this can be seen as an optimization for normalizing scores, just as the logarithm for $N \cdot DF^{-1}$.

In cases where this ranking formula might not fit, Lucene provides the option to specify a custom ranking algorithm (extending *Similarity*).

4.2 Distributed Search with FAST

4.2.1 Background

The FAST system can retrieve information from the Web using a *crawler*, from relational databases using a *database connector* and from the file system using a *file traverser*.

The FAST crawler scans the specified Web sites and follows hyperlinks based on configurable rules, extracts the desired information and detects duplicates. The document processing transfers the HTML content into structured data as defined by the Web representation. The Crawler supports incremental crawling, dynamic pages, entitled content (cookies, SSL, password), HTTP 1.0/1.1, FTP, frames, robots.txt and META tags. Additionally, FAST supports the handling of JavaScript parts, especially indexing dynamic content generated by JavaScript on the client side.

Its database connector currently provides Content Connectors for the most popular SQL databases, including Oracle and Microsoft SQL server.

The File Traverser scans the local file system and retrieves documents of various formats in a similar way as the crawler.

XML content can be submitted directly via the Content API or the File Traverser. FAST Data Search supports Row-Set XML and conversion from custom XML formats. The XML conversion is performed as part of the document processing stage, and is controlled using an XSL Style Sheet or an XPath-based configuration file. The sketch on Fig. 4.2.1 shows the data workflow for BASE, the Bielefeld Academic Search Engine:

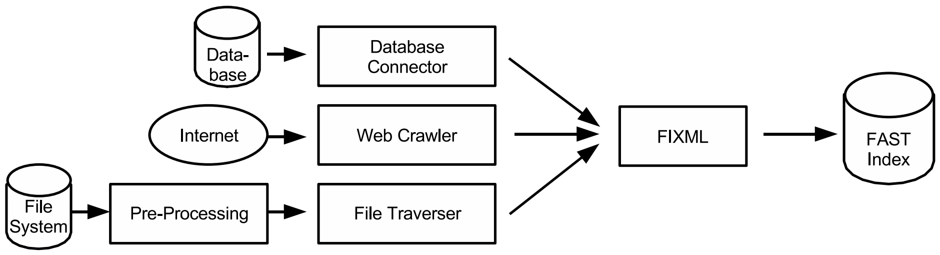


Figure 4.1: FAST Dataflow

The indexing process is highly configurable and covers a list of internal (provided by FAST) and external steps which can be defined and ordered per collection. The indexing stages in FAST are summarized in Table 4.2.

4.2.2 FIXML

The FAST search engine processes incoming data in a flexible, multi-step processing stage. This covers both internal FAST filter stages and additional, individual stages. At the end of this pipeline, a file in the internal FIXML format has been produced which will then be used by the internal FAST index process.

Since some of the steps are really internal to the system and not documented, the following description can only provide an overview.

The Fast IndeXer Markup Language (FIXML) format defines the internal indexing structure according to the index configuration. In particular, the configuration defines the list of fields which may appear in the FIXML files (for context, summary and ranking purposes) and the fields which will be lemmatized.

The FIXML files consist of a context, a summary and a ranking division. The structure is presented in Table 4.4

Internal parts are separated by the token “FASTpbFAST”, for example:

```
<context name="bcondctype"><![CDATA[
FASTpbFAST Text FASTpbFAST article FASTpbFAST
]]></context>
```

Indexing Stage	Description
1. Language and encoding detection	Detects the languages of the document and used encoding and writes this information into the corresponding internal fields (language, languages for multi-language objects, charset). This is later used for lemmatization.
2. Format detection	Detects the document format.
3. Delete MIME Type	In specific cases, the specified format type found in the document is wrong. Therefore this stage deletes the information.
4. Set MIME Type	This is an added stage to reset the mime type externally, in case of correcting wrong information.
5. Handle zipped files (uncompress) and set new format.	Uncompresses zipped object files, detects the occurring document format and resets the internal format field of the unpacked file and its included document.
6. Set content type	This stage sets a BASE–specific field which defines whether the document contains just metadata or metadata with full-text.
7. PostScript conversion	Reads a PostScript document, transforms it into a raw text format and writes the results into specific fields
8. PDF conversion	Reads a PDF document, extracts the raw text and writes the result into specifc fields.
9. SearchML-Converter	This stage processes all input formats (ML for multi-language) and can handle more than 220 different file formats. It extracts the raw text from the original file.
10. Generate teaser based on defined fields	This stage builds the *teaser* document summary.
11. Tokenization of selected fields	Takes field content of specified fields and extracts a normalized form.
12. Lemmatization	This stage handles the fields for lemmatization (defined in the index configuration file) based on the detected language and packs the lemmatized word forms into the internal structures.
13. Vectorization	This step determines a document's vector in vector space. This can be used later for a vector space search. At this point, a file in FIXML format has been produced which contains the result of the processing steps. This file will be processed by the internal FAST index process as final step.
14. Indexing	The index process itself can be influenced by various settings shown in Table 4.3.

Table 4.2: FAST Indexing Stages

Setting Field	Description
name	-
type	used for sorting
string	a free-text string field
int32	a 32 bit signed integer
float	decimal number (represented by 32 bits)
double	decimal number with extended range (represented by 32 bits)
datetime	date and time (ISO-8601 format)
lemmatize	enables lemmatization and synonym queries
full-sort	enables sorting feature for the field
index	enables indexing of this field
tokenize	enables language dependent tokenizing (removing/normalizing punctuation, capital letters, word characters) at the field level
vectorize	enables creation of similarity vectors for the specified field. A changeable stopword list depending on the detected language of the document is used
substring	if specified, the field will be configured to support substring queries
boundarymatch	determines if searches within a field can be anchored to the beginning and/or the end of the field
phrases	enables phrase query support in the index for the defined composite-field

Table 4.3: FAST Indexing Settings

Context	at least once
Ranking	optional
Attribute vectors	optional
Summary	optional

Table 4.4: FIXML Structures

The context part contains those fields which are used for indexing the document, e.g. for querying. The words are added in the catalog's index dictionary in a case-insensitive way (lowercase). It contains both the normalized words (the corresponding field names start with a "bcon") and the lemmatised field (the field name starts with a "bcol") divided into the normalized tags followed by the lemmatised versions. Thus, fields which are defined as "to be lemmatised" appear twice, with a header of "bcon" (normalized) and "bcol" (lemmatised).

Example for a lemmatised field's content:

```
24 april 2003 FASTpbFAST beitrags beitrage beitrages
```

The ranking section defines static ranking values, which may influence (boost) the final ranking. In addition to that, the "attribute vectors" section holds information used for drill-down search; the according field names start with "bavn", for example:

```
<attrVec name="bavncharset">
<avField><![CDATA[utf-8]]></avField>
</attrVec>
```

The summary section contains those fields which are used for preparation of the result display of the specific document. Obviously, this content is stored in the index as well to construct the original format of the document without storing it in its original format. The summary chapter is optional.

The final section describes the document vector, which can be used for “find-like-this”-typed searches. Example:

```
<sField name="docvector"> <![CDATA[[
adressdatenbank, 1][pflege, 0.816497]
[erfassten daten, 0.645497]
]]> </sField>
```

4.2.3 Prospective Federated Search using FAST

For future versions, a new feature called “federated search” is being developed by FAST. While not yet announced officially, this feature seems to support FAST search systems only. If so, it would rather be a distributed search tool for homogeneous systems than a Federated Search environment in our terms, and would not suit the needs of combining search engine systems from different vendors. Still, for a FAST only environment, this is certainly an interesting feature to look into it deeper.

4.3 Combining Lucene and FAST Data Search

4.3.1 Plugin Architecture

While being very similar at some points, the two search engine products considered (FAST Data Search and Lucene) clearly expose incompatiblities on several levels: index structures, API and ranking. Since FAST Data Search is a commercial, closed-source product, we were not able to have a deeper look into the technical internals. For the near future, we also do neither expect that these structures can and will be adapted to conform to the Lucene API, nor vice versa. As a long-term perspective, we hope that both systems will ship with a SDARTS-conformant interface.

In the meantime, we propose the following *plugin*-based approach. Given the fact that FAST Data Search provides many valuable features prior to indexing (especially crawling and text-preprocessing), whereas Lucene concentrates on the indexing part, we will combine the advantages of both systems into one application, so most existing applications and workflows can be preserved.

This application will take input data (pre-processed documents) from both FAST and Lucene installations and index them in a homogeneous manner. Think of this indexing component as a plugin to both search engine systems. Since Lucene is open-source and already one of the two engines to be supported, the plugin will be based upon its libraries.

Each search engine which participates in the search federation will have such a (Lucene-based) plugin. Search in the federation is then performed over these plugins instead of the original vendor's system. However, per-site search (that is, outside the federation) can still be accomplished without the plugin.

The plugin-enriched Federated Search infrastructure is shown in figure 4.2. The participating instutions may provide several user interfaces, with different layouts and functionalities, inside the federation, from which a user can perform queries over the federation. This can be accomplished by querying the plugins from the UI in parallel, or let another specialized plugin do this for the user interface.

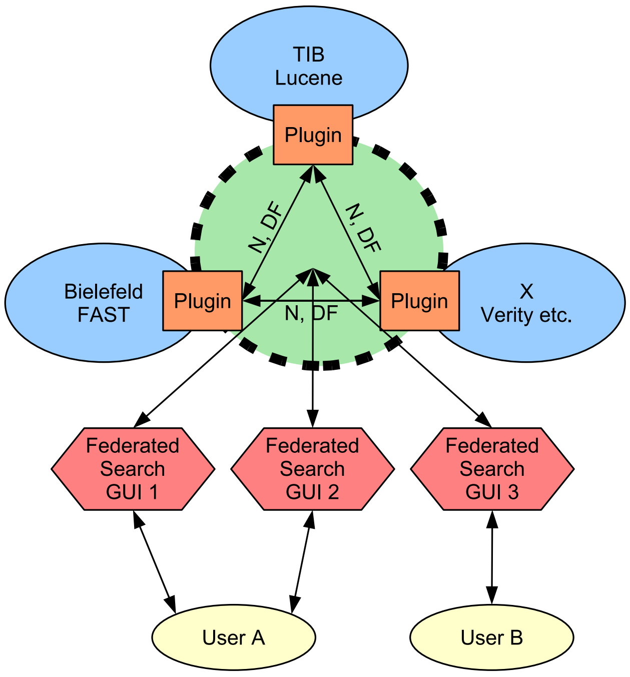


Figure 4.2: Federated Search infrastructure using the Plugin Architecture

This approach provides the advantages of centralized search (homogeneous index structure, ranking and API), while offering distributed search using standard Lucene features. For this, the original document data has to be automatically re-indexed by the Lucene-Plugin, whenever the document collection changes. As a consequence, additional system resources are required (harddisk space, CPU, RAM). However, these resources would probably be necessary for participating in the federation anyway. On the other side, the plugin architecture will impose no major administrative overhead, since the search engine administrators can still specify all crawling- and processing parameters in the original search engine product.

Of course, the concept of a “plugin” is not limited to a Lucene-based implementation. The plugin simply serves as a common (open) platform for homogeneous, cooperative distributed search and Lucene suits very well here.

4.3.2 Importing FAST Data Search Collections into Lucene using FIXML

The aforementioned procedure of re-indexing the document data into the plugin’s index structure clearly depends on the underlying search engine. We will now examine how the plugins can be implemented for each search engine.

FAST Data Search uses an intermediate file format for describing indexable document collections, FIXML. It contains all documentation information which should go directly to the indexer (that means, the input data has already been transformed to indexable terms: analyzed, tokenized, stemmed etc.). FIXML uses data-centric XML for document representation and index description.

A document in FIXML consists of the following parts:

- Contexts (= fields / zones). Each (named) context may hold a sequence of searchable tokens. FIXML stores them as white space-separated literals, enclosed in one CDATA section.

- Catalogs. Contains zero or more Contexts. This can efficiently be used to partition the index.

- Summary Information. Each (named) summary field may hold human-readable, non-indexed data, for result presentation.

- Attribute Vectors. Each (named) attribute vector may hold one or more terms (each enclosed in a CDATA section) which can be used for refining the search (dynamic drill-down), e.g. restrict the results to pages by a specific author or to pages from a specific category etc.

- Additional Rank Information. A numerical document weight/boost.

This structure is comparable to Lucene's document API. Both structures describe documents as a set of fields containing token information. However, the way to define such a structure in Lucene is different. Lucene has less levels of separation. For example, in Lucene, a field can directly be marked "to be stored", "to be indexed" or both. It also does not provide the "catalog" abstraction layer. Moreover, FAST's dynamic drill-down feature using Attribute Vectors is not directly provided in the Lucene distribution, but can be added by a custom implementation.

Since the FIXML data is already pre-processed, we can take this document data and index it using Lucene and some custom glue-code. Also important, the FIXML data only changes partially when documents are added or deleted from the document collection, so a full Lucene-index rebuild is not necessary in most cases, it can be easily accomplished by adding/masking the new/deleted documents. In general, some FAST's data structure concepts can be translated to Lucene as presented in the Table 4.5.

FAST Index Feature	**Lucene Index Counterpart**
Catalog	Modeled as a set of Lucene indexes in the same base directory. Each index contains the same number of Lucene documents. Then, in Lucene, a document in FAST terminology is the union of all indexes' documents which have the same document ID. Such index sets can be accessed via Lucene's ParallelReader class, which makes this set look like a single Lucene index.
Summary information	Could be stored along with the index information. However, summary field names seem to be different from context names in FIXML, so we will simply have a "special" catalog which only contains stored, yet unindexed data.
Attribute Vectors	We also treat them as a special catalog, which only contains indexed data. In contrast to regular catalogs, we do not tokenize the attribute values.

Table 4.5: Comparison of FAST's and Lucene's data structure concepts

As a proof of concept, we have developed a prototype plugin for FAST Data Search, which translates FIXML document collections into Lucene structures. The plugin consists of three components, consisting of several classes:

1. The Lucene index glue code (*Indexer*)

  - *LuceneCatalog*. Provides read/write/search access to one Lucene index (using *IndexReader*, *IndexWriter*, *IndexSearcher*)
  - *LuceneCollection*. Provides access to a set of *LuceneCatalog*s. This re-models FAST's concept of indexes (collections) with sub-indexes (catalogs) in Lucene. The catalogs are grouped on harddisk in a common directory.

2. The collection converter (*Parser*)
   - *FIXMLDocument*. Abstract definition of a document originally described by FIXML.
   - *FIXMLParser*. Parses FIXML files into *FIXMLDocument* instances.
   - *FIXMLIndexer*. Takes *FIXMLDocument*s and indexes them into a *LuceneCollection*.

3. The query processor (*Searchable*)
   - *CollectionSearchable*. Provides a Lucene Searchable interface for a *LuceneCollection*. The set of catalogs to be used for search can be specified.
   - *QueryServlet*. The Web interface to the *CollectionSearchable*.

As Lucene is a programming library, not a deployment-ready software, it is hardly possible to estimate the requirements of adaptation of an existing search engine for use with the Federated Search infrastructure. However, if the Lucene index files can be accessed directly, the major task is to map the document data to the federation's common schema. This involves the conversion between different Lucene *Analyzer*s and different field names. Usually, this conversion is static, that is, the documents have to be re-indexed, just as with FAST Data Search. However, if the same *Analyzer* has been used, it is likely that only the field names have to be mapped, which can be done dynamically, without re-indexing. We have tested the latter case with a test document set from TIB, the results are discussed in Section 5.

While not being in the focus of this paper, we are confident that the costs for integrating collections from other search engines or databases (e.g. MySQL) are similar to our effort for FAST Data Search. Indeed, the major advantage of our approach is that we do not require the export of collection statistics from the original product.

4.3.3 Plugin Implementation Schedule

Regarding a production version of our implementation, we estimate the additional time needed for implementation of a SDARTS-compliant plugin infrastructure between TIB and Bielefeld Libraries with 8 person months (PM), detailed as follows:

- Additional studies (0.5 PM);
- Implementation of interfaces to support SDARTS functionality (2.5 PM): interfaces for source content summary, query modifiers, filter expressions/ranking expressions, information imbedded in query, information imbedded in result, source metadata attributes;
- Implementation based on these interfaces for TIB Lucene installation (1 PM): document schema investigation, mapping between generic interface and TIB backend instance;
- Implementation based on these interfaces for Bielefeld installation (1 PM): document schema investigation, mapping between generic interface and Bielefeld backend instance;
- Prototype testing with all collections from TIB (1 PM);
- Prototype testing between Bielefeld and TIB (1 PM);
- Writing developer and user guides (1 PM);

Chapter 5

Evaluation

This chapter contains results of the evaluation of our first prototype. We describe test collections, connection setup and some preliminary numbers for indexing and search performance.

5.1 Collection Description

For the representativeness of our evaluation, we used several well-known document collections, a collection based on the HU-Berlin EDOC document set (metadata and full-text), the ArXiv, and Citeseer OAI metadata collections.

EDOC[1] is the institutional repository of Humboldt University. It holds theses, dissertations, scientific publications in general and public readings. The document full-text is stored in a variety of formats (SGML, XML, PDF, PS, HTML); non-text data (video, simulations, ...) is availabe as well. Only publications approved by the university libraries of Humboldt University are accepted. Currently, about 2,500 documents are provided. CiteSeer[2] is a system at Penn State University, USA, which provides a scientific literature DL and search engine with the focus on literature in computer and information science. There is a special emphasis on citation analysis. At the moment, the system contains more than 700,000 documents. The ArXiv preprint server[3] is located at Cornell University Library and provides access to more than 350,000 electronic documents in Physics, Mathematics, Computer Science and Quantitative Biology.

[1]`http://edoc.hu-berlin.de`
[2]`http://citeseer.ist.psu.edu/`
[3]`http://arxiv.org/`

In order to avoid difficulties at pre-indexing stages (crawling, tokenizing, stemming etc.), it was essential to have these collections already in a format suitable for search.

We have collected the annotated metadata via their freely available OAI interfaces. In the case of EDOC, we enriched it with the associated full-text, using Bielefeld's installation of FAST Data Search, the Bielefeld Academic Search Engine (BASE).

Since we had to define a common document schema, we simply re-used BASE's custom structure, defining several fields for metadata properties and a field for the document's full-text (if available). In order to import the collection into our federation setup, we passed the created FIXML files (see Subsection 4.2.2) to our plugin for re-indexing.

5.2 Federation Setup

For our experiments, we have set up a small federation of two search engines between Bielefeld University Library and L3S Research Center, using one server per institution. The servers communicate via a regular internet connection.

On each server, we have installed our plugin prototype and the corresponding Web front-end. The front-end only communicates to the local plugin, which then may access local, disk-based Lucene document collections or external, network-connected plugins, depending on the exact evaluation task.

The server in Hannover was a Dual Intel Xeon 2.8 GHz machine, the server in Bielefeld was powered by a Dual AMD Opteron 250, both showing a BogoMIPS speed index of around 5000 (A: 5554, B: 4782). On the Bielefeld server, a Lucene-plugin based version of the BASE (Bielefeld Academic Search Engine) Web user interface has been deployed, on the Hannover server a fictitious "HASE" (Hannover Academic Search Engine) front-end, also running on top of a a Lucene-plugin search engine. Both GUIs, HASE and BASE are only connected to their local search plugins. The plugins then connect to the local collections (ArXiv and CiteSeer in Bielefeld and EDOC in Hannover) as well as to the plugin at the other side. This setup is depicted in figure 5.1.

For some tests, the setup has also been modified to have the connection between Hannover and Bielefeld uni-directional (that is, a search on HASE would only provide local results, whereas a search on BASE would combine local with remote results, and vice versa).

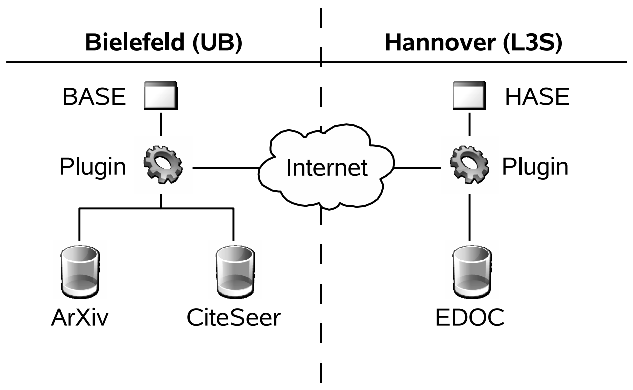


Figure 5.1: Test scenario for Federated Search

5.3 Hypotheses

Due to the underlying algorithms (see 3), we might expect that there is no difference in the result set (simple presence and position in the result list) between a local, combined index covering all collections and the distributed scenario proposed using Lucene-based Federated Search.

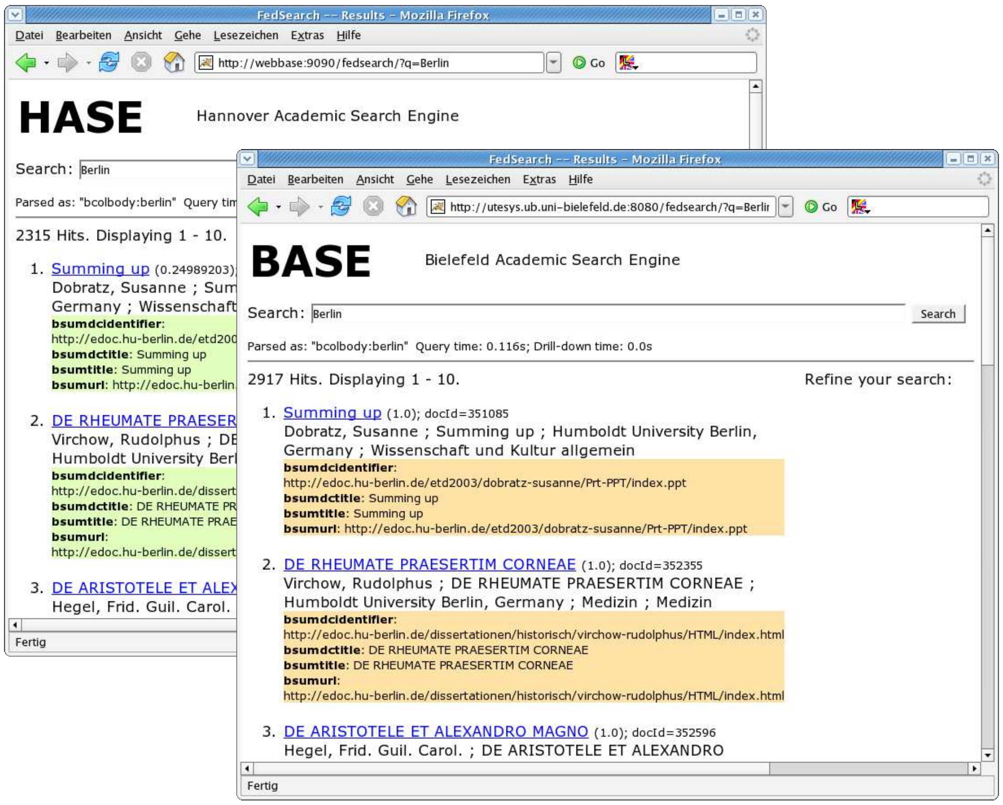
Also, we expect almost no difference in search performance, since the collection statistics information is not exchanged at query time, but only at startup time and whenever updates in any of the included collections have occurred. Since the result display will only list a fixed number of results per page (usually 10 short descriptions, all of about the same length), the time necessary for retrieving this results list from the contributing plugins is constant, i.e. not dependent on the number of found entries, depending only on the servers' overall performance (CPU + network I/O).

Figure 5.2 shows how the outputs of HASE and BASE should look like. Please note the different number of results (HASE: 2315 vs. BASE: 2917) in the picture, due to the uni-directional connection setup mentioned above.

5.4 Results

5.4.1 Re-indexing

Our current prototype re-indexed the metadata-only ArXiv collection (350,000 entries) in almost 2 hours; the EDOC full-text document set (2500 documents)


FedSearch -- Results - Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://webbase:9090/fedsearch/?q=Berlin
Go
Fertig
FedSearch -- Results - Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://utesys.ub.uni-bielefeld.de:8080/fedsearch/?q=Berlin
Go
Search
Fertig

HASE Hannover Academic Search Engine

Search: Berlin

Parsed as: "bcolbody:berlin" Query tim

2315 Hits. Displaying 1 - 10.

1. Summing up (0.24989203);
Dobratz, Susanne ; Sum
Germany ; Wissenschaft
bsumdcidentifier:
http://edoc.hu-berlin.de/etd200
bsumdctitle: Summing up
bsumtitle: Summing up
bsumurl: http://edoc.hu-berlin.

2. DE RHEUMATE PRAESER
Virchow, Rudolphus ; DE
Humboldt University Berl
bsumdcidentifier:
http://edoc.hu-berlin.de/dissert
bsumdctitle: DE RHEUMATE PR
bsumtitle: DE RHEUMATE PRAE
bsumurl:
http://edoc.hu-berlin.de/dissert

3. DE ARISTOTELE ET ALEX
Hegel, Frid. Guil. Carol.

BASE Bielefeld Academic Search Engine

Search: Berlin

Parsed as: "bcolbody:berlin" Query time: 0.116s; Drill-down time: 0.0s

2917 Hits. Displaying 1 - 10.

Refine your search:

1. Summing up (1.0); docId=351085
Dobratz, Susanne ; Summing up ; Humboldt University Berlin, Germany ; Wissenschaft und Kultur allgemein
bsumdcidentifier:
http://edoc.hu-berlin.de/etd2003/dobratz-susanne/Prt-PPT/index.ppt
bsumdctitle: Summing up
bsumtitle: Summing up
bsumurl: http://edoc.hu-berlin.de/etd2003/dobratz-susanne/Prt-PPT/index.ppt

2. DE RHEUMATE PRAESERTIM CORNEAE (1.0); docId=352355
Virchow, Rudolphus ; DE RHEUMATE PRAESERTIM CORNEAE ; Humboldt University Berlin, Germany ; Medizin ; Medizin
bsumdcidentifier:
http://edoc.hu-berlin.de/dissertationen/historisch/virchow-rudolphus/HTML/index.html
bsumdctitle: DE RHEUMATE PRAESERTIM CORNEAE
bsumtitle: DE RHEUMATE PRAESERTIM CORNEAE
bsumurl:
http://edoc.hu-berlin.de/dissertationen/historisch/virchow-rudolphus/HTML/index.html

3. DE ARISTOTELE ET ALEXANDRO MAGNO (1.0); docId=352596
Hegel, Frid. Guil. Carol. ; DE ARISTOTELE ET ALEXANDRO

Figure 5.2: Bielefeld and Hannover prototype search engines

was imported in 19 minutes. We expect that re-indexing will be even faster with a production–level implementation.

The resulting index structure was about 20-40% of the size of the uncompressed FIXML data (depending on collection input) and also, interestingly, about a third of the size of the original FAST index data. This might be due to redundancy in the FAST data structures, or, more likely, to extra data structures in the FAST index not used in Lucene. However, we have no further documentation regarding the FAST Data Search index structure.

The values show that there is only a slight overhead induced by the re-indexation compared to a centralized index based on FAST Data Search, or a possible implementation of Federated Search provided natively by FAST.

5.4.2 Search

We performed several searches (one-word-, multi-word- and phrase queries) on our federation as well as on a local index, both using Lucene. We measured average query time and also compared the rankings for one word, multi-word and phrase queries. Query times of a distributed setup were almost equal to a local setup (always about 0.05-0.15 seconds per query, with an overhead of ca. 0.4 seconds for the distributed setup).

While in most cases the rankings were equal, as expected, in some cases we noticed that the ranking is only *almost* identical. The difference between local and distributed search results comes from the fact that Lucene only ranks by numerical score, then by document ID, which in turn is influenced by Lucene's `MultiSearcher` — all documents from a collection A have lower document IDs that those from collection B. Hence, whenever identical documents are spread over several collections, it is not guaranteed that they are directly listed after each other. A solution to this problem is to simply sort all documents with the same score literally by URI. This adds no major performance penalty, as we can do this in the GUI (i.e., only for the top-ranked results).

Chapter 6

Conclusion and Further Work

6.1 Project Results and Recommendations

This report analyses Federated Search in the VASCODA context, focusing on the existing TIB Hannover and UB Bielefeld search infrastructures. Specifically, it describe general requirements for a seamless integration of the two full-text search systems FAST (Bielefeld) and Lucene (Hannover), and evaluate possible scenarios, types of queries, and different ranking procedures, and then describes how a Federated Search infrastructure can be implemented on top of these existing systems.

An important feature of the proposed federation infrastructure is that participants do not have to change their existing search and cataloging systems. Communication within the federation is performed via additional plugins, which can be implemented by the participants, provided by search engine vendors or by a third party. When participating in the federation, all documents (both full-text and metadata) stay at the provider side, no library document / metadata exchange is necessary.

The integration of collections is based on a common protocol, SDARTS, to be supported by every member of search federation. SDARTS is a hybrid of the two protocols SDLIP and STARTS. SDLIP was developed by Stanford University, the University of California at Berkeley, the California Digital Library, and others. STARTS protocol was designed in the Digital Library project at Stanford University and based on feedback from several search engines vendors. Additional advantages can be gained by agreeing on a common document schema, as proposed by the Vascoda initiative, though this is not a precondition for Federated

Search.

The main advantages of this Federated Search architecture are the following:

1. The quality of the search results is better than in usual metasearch approach, because the final document ranking is consistent, i.e. equal to a centralized setup.

2. The infrastructure does not require a centralized control point, every DL keeps its content private and does not need to uncover it to other participants.

3. The connection to the federation requires only a small effort, one SDARTS-compatible plugin has to be developed per one search engine product (not per installation). This can be accomplished by the original product vendor or by third-party.

4. The local search system management infrastructure and workflow does not change, so every DL can connect Federated Search indexes to its current interface, not being forced to switch to another software. This makes the participation in DL federation easy, and increases coverage and quality of search.

A preliminary prototype of the proposed plugin mechanism integrates FAST and Lucene-based document collections and produces a combined homogeneous document ranking. The prototype currently does not employ SDARTS itself, it merely shows the principal possibility of Federated Search using well-known algorithms present in current search engine systems. However, we recommend to use the existing SDARTS protocol for the production version.

6.2 Joining Costs and Setup Considerations

The important question is what a DL has to do for joining the federation, how much it costs and what actions are required? One might guess, that additional Federated Search capabilities might require a lot of efforts comparable to the installation of a complex search system. Fortunately, this is not the case, every new participant has just to make a reasonably small effort. A DL has to provide a fast Internet connection and additional computational power, it can be a new server or parts of resources of current servers (depending on the current setup). A new participant also has to provide storage space for the Federated Search index files, which in our experiments lead to an increase of about 30% over the size of original FAST index. If the local search engine product is already used by another library in the federation, the cost of developing a new plugin for a prospective participant can be shared with all other members which use the same type of search engine. For example, all FAST users can share the price for developing and maintaining a Federated Search plugin for FAST Data Search.

The proposed plugin architecture can be deployed in *several* ways (including combinations of the following):

- Plugin provision
  1. The library provides its own plugin
  2. A third-party provides the plugin on behalf of the library
- Search Interface provision
  1. The library provides its own search interface
  2. A third-party provides a common search interface for the federation

Providing the plugin through a third-party makes sense whenever a digital library does not have sufficient resources to directly participate in the federation. In this case, the DL may supply the service provider with documents/ abstract information or just with the Lucene index files generated by a local plugin (which does not have to be connected to the federation). The latter case might be considered safer with respect to copyright issues because only information strictly required for the search functionalities is exchanged.

While there is no limit on the number of search interfaces in principle, we suggest that at least one common portal provides access to all collections made available by the participants (most probably the Vascoda portal).

Compared to a homogeneous search engine architecture, the additional costs are managable since they do not increase with the number of participants but with the number of different systems. Any additional expenses, like network, computational and human resources apply to both solutions, homogeneous (e.g. FAST only) and heterogeneous architectures (SDARTS).

6.3 Next Steps

The next major step is to implement a full production-version plugin for all collections of the current participants, Bielefeld Library and Hannover Technical Library. This will require additional programming and testing, the final search interface should then be able to search over any number of available collections. We expect that the implementation takes about 8 person months (see Section 4.3.3). It is possible to speed up this process by distributing the tasks among several people.

The second step is to set up a larger federation between several institutions. After that, additional services built upon the Federated Search infrastructure are possible, such as application-domain specific user interfaces, search result post-processors services and others, which can help to improve search experience.

Once a federation is established, we recommend to start a dialog with the search engine vendors who could directly supply the search software together with a module suitable for Federated Search, thus enabling libraries to get the necessary components through regular software updates.

Chapter 6

Zusammenfassung & Ausblick

6.1 Ergebnisse und Empfehlungen

Dieser Bericht analysiert verteilte Suche (Federated Search) im VASCODA-Kontext, ausgehend von den Suchinfrastrukturen der TIB Hannover und der UB Bielefeld. Die Arbeit beginnt mit der Spezifikation grundsätzlicher Anforderungen für eine nahtlose Integration bestehender Volltext-Suchsysteme, im speziellen FAST Data Search (Bielefeld) und Lucene (Hannover), vergleicht deren Funktionalitäten und evaluiert mögliche Szenarien für den Einsatz der Systeme im Rahmen einer verteilten Suchinfrastruktur.

Der Bericht beschreibt eine verteilte Suchinfrastruktur, die aufbauend auf diesen bestehenden Systemen implementiert werden kann. Wichtig hierbei ist, dass alle Teilnehmer an dieser Föderation ihre bestehenden Such- und Katalogsysteme weitestgehend weiterverwenden können. Die Kommunikation innerhalb der Föderation erfolgt mittels zusätzlicher Komponenten, sogenannter Plugins, die durch den Suchmaschinen-Anbieter, den Teilnehmer selbst oder einem Drittanbieter implementiert werden können. Ein Austausch von Dokumenten / Metadaten zwischen den Teilnehmern ist hierbei nicht notwendig.

Die Integration der Dokumentsammlungen erfolgt über ein gemeinsames Protokoll, SDARTS, das von jedem Teilnehmer unterstützt wird. SDARTS setzt sich aus den zwei Protokollen SDLIP und SDARTS zusammen. SDLIP wurde von der Stanford University, der University of California at Berkeley, der California Digital Library und anderen entwickelt. Das STARTS Protokoll wurde im Digital Library Projekt in der Stanford University zusammen mit verschiedenen Suchmaschinenanbietern entwickelt. Die Nutzung eines gemeinsames Dokument

/ Metadatenschemas ist von Vorteil, aber keine Voraussetzung für verteilte Suche.

Die wichtigsten Vorteile der in diesem Bericht beschriebenen verteilten Sucharchitektur lassen sich wie folgt zusammenfassen:

1. Die Qualität der Suchergebnisse ist besser als mit herkömmlicher Meta-Suche, da das erzeugte Ranking konsistent ist, d.h. identisch zu dem einer einzigen Suchmaschine.

2. Die Infrastruktur erfordert keine zentrale Umschlagstelle, jede digitale Bibliothek behält weiterhin die volle Kontrolle über die eigenen Inhalte und muss diese anderen Teilnehmern nicht offenlegen.

3. Die Anbindung an die Föderation erfordert nur geringen Zusatzaufwand. Lediglich ein SDARTS-kompatibles Plugin muss pro Suchmaschinen-Produkt entwickelt werden (nicht pro Installation). Dies kann z.B. durch den Hersteller oder durch Dritte erfolgen.

4. Die bibliotheksinterne Infrastruktur und ihre Verwaltungsabläufe bleiben erhalten. Jede digitale Bibliothek kann so ohne vollständige Systemumstellung an der verteilten Suchinfrastruktur teilnehmen. Durch die einfache Integrierbarkeit erhöhen sich die Abdeckung und die Qualität der Föderation.

Ein vorläufiger Prototyp des vorgeschlagenen Plugin-Mechanismus integriert FAST- und Lucene-basierte Dokument-Kollektionen und erlaubt homogenes Dokument-Ranking über verteilte Kollektionen. Der Proof-of-Concept-Prototyp selbst setzt SDARTS derzeit nicht ein, sondern zeigt vielmehr die prinzipielle Einsatzbarkeit von verteilter Suche auf Basis etablierter Algorithmen. Für den Produktiveinsatz empfehlen wir aber jedenfalls den Einsatz von SDARTS.

6.2 Beitrittskosten und mögliche Konstellationen

Für jede an einer solchen verteilten Sucharchitektur interessierte Bibliothek stellt sich natürlich die Frage, was zu tun ist, um der Föderation beizutreten und was dies kostet. Man mag annehmen, dass eine zusätzliche Funktionalität wie Federated Search einen ähnlich großen Installations- und Wartungsaufwand mit sich bringt wie die Installation eines kompletten Suchmaschinen-Systems. Glücklicherweise ist dies nicht der Fall, der Aufwand für die Teilnahme ist angemessen: Eine digitale Bibliothek muss eine schnelle Internetverbindung und zusätzliche Rechenleistung bereitstellen (z.B. ein neuer Server oder dedizierte Ressourcen eines bestehenden, abhängig von der jeweiligen lokalen Situation). Ferner muss der Teilnehmer gegebenenfalls zusätzlichen Speicherplatz für die Reindexierung der Datenbestände zur Verfügung stellen. In unseren Experimenten mit FAST Data Search war dies etwa 30% Zusatzaufwand (Festplattenspeicher) zum ursprünglichen FAST-Index.

Für den Fall, dass das eingesetzte Suchmaschinen-System bereits von einer anderen digitalen Bibliothek verwendet wird, können die Entwicklungskosten für das Plugin geteilt werden.

Die in diesem Bericht beschriebene verteilte Suchmaschinenarchitektur kann in folgenden verschiedenen Ausprägungen eingesetzt werden (inklusive Kombinationen):

- Bereitstellung des Plugins
  1. Die Bibliothek stellt das Plugin selbst zur Verfügung.
  2. Ein Drittanbieter stellt das Plugin im Namen der Bibliothek der Föderation zur Verfügung.
- Such-Oberfläche
  1. Jede Bibliothek bietet eine eigene Suchmaschinen-Oberfläche.
  2. Ein Drittanbieter stellt eine einheitlicher Oberfläche für die gesamte Föderation bereit.

Das Bereitstellen von Plugins über Drittanbieter macht dann Sinn, wenn der digitale Bibliothek selbst nur unzureichende Ressourcen zur Verfügung stehen, um direkt an der Föderation teilzunehmen. In diesem Fall kann die Bibliothek dem Drittanbieter die Original-Dokumente (Zusammenfassungen etc.) übermitteln, oder einfach die Lucene-Indexdaten, die von einem lokalen Plugin erzeugt werden (dieses Plugin muss dann nicht an die Föderation angebunden sein). Der zweite Ansatz kann als "sicherer" in Bezug auf urheber- und lizenzrechtliche Belange angesehen werden, da ausschließlich Informationen übermittelt werden müssen, die tatäschlich für die Durchführung der Suche notwendig sind.

Obwohl der vorgeschlagene Ansatz die Anzahl der eingesetzten Such-Oberflächen (Portale) nicht impliziert, schlagen wir vor, dass es ein gemeinsam vermarktetes Portal gibt, das den Zugriff auf alle angeschlossenen Dokument-Kollektionen bietet, z.B. das Vascoda-Portal.

Im Vergleich zu einer homogenen Suchmaschinenarchitektur sind die Anschaffungskosten überschaubar, da sie nicht mit der Anzahl der Teilnehmer skalieren, sondern mit der Anzahl der unterschiedlichen Suchsysteme. Alle weiteren Ausgaben, wie Netzwerk-, Rechen- oder Personalressourcen fallen bei beiden Ansätzen an, sowohl für homogene Systeme (z.B. Federated Search mit FAST-Suchmaschinen) als auch für heterogene Systeme (mittels SDARTS).

6.3 Ausblick

Der nächste Schritt ist die Implementation eines voll funktionsfähigen Plugins auf der Basis von SDARTS, zwischen den beiden Installationen in Bielefeld und Hannover. Dies erfordert zusätzliche Programmierung und Tests; die endgültige Version soll in der Lage sein, beliebig viele Dokument-Kollektionen in der Föderation anzubieten. Wir gehen davon aus, dass die Implementation etwa 8 Personenmonate in Anspruch nehmen wird (siehe Abschnitt 4.3.3 für Details).

Im Anschluß daran ist vorgesehen, die Föderation um andere Teilnehmer zu erweitern. Weiters kann über zusätzliche Dienste nachgedacht werden, die auf der verteilten Sucharchitektur aufbauen, wie z.B. anwendungsspezifische Benutzeroberflächen, augmentierte Suche usw.

Sobald eine solche Föderation aufgebaut ist, sollte der Dialog mit Suchmaschinen-Herstellern gesucht werden, um deren Systeme direkt mit einem SDARTS-kompatiblen Modul auszustatten, das dann bequem per Softwareupdate eingespielt werden kann.

Bibliography

[1] Harald Alvestrand. Tags for the Identification of Languages (RFC 1766). `http://asg.web.cmu.edu/rfc/rfc1766.html`, 1995.

[2] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, pages 22–28, March 2003.

[3] An American National Standard Developed by the National Information Standards Organization. Information Retrieval (Z39.50): Application Service Definition and Protocol Specification, 2003.

[4] J. P. Callan, Z. Lu, and W. Bruce Croft. Searching Distributed Collections with Inference Networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *SIGIR '95: Proceedings of the 18th Annual International Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, 1995. ACM Press.

[5] Nicholas Eric Craswell. *Methods for Distributed Information Retrieval*. PhD thesis, ANU, January 01 2001.

[6] W. Bruce Croft. Combining Approaches to IR. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[7] Doug Cutting et al. Lucene. `http://lucene.apache.org`.

[8] Panagiotis G. Ipeirotis et al. SDARTS Server Specification. `http://sdarts.cs.columbia.edu/javadocs/index.html`, 2004.

[9] Otis Gospodnetic and Erik Hatcher. *Lucene in Action*. Manning, 2005.

[10] Luis Gravano, Kevin Chen-Chuan Chang, Hector Garcia-Molina, and Andreas Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching. In

SIGMOD '97: Proceedings of the 1997 ACM International Conference on Management of Data, pages 207–218, 1997.

[11] Noah Green, Panagiotis G. Ipeirotis, and Luis Gravano. SDLIP + STARTS = SDARTS a Protocol and Toolkit for Metasearching. In *JCDL '01: Proceedings of the The First ACM and IEEE Joint Conference on Digital Libraries*, pages 207–214, 2001.

[12] Open Archives Initiative. The Open Archives Initiative Protocol for Metadata Harvesting Protocol Version 2.0 of 2002-06-14. `http://www.openarchives.org/OAI/openarchivesprotocol.html`.

[13] Andreas Paepcke, R. Brandriff, G. Janee, R. Larson, B. Ludaescher, S. Melnik, and S. Raghavan. Search Middleware and the Simple Digital Library Interoperability Protocol. In *D-Lib Magazine*, volume 6, 2000.

[14] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Research and Development in Information Retrieval*, pages 275–281, 1998.

[15] Luo Si and Jamie Callan. A Semisupervised Learning Method to Merge Search Engine Results. *ACM Transactions on Information Systems*, 21(4):457–491, 2003.

[16] Luo Si, Rong Jin, James P. Callan, and Paul Ogilvie. A language modeling framework for resource selection and results merging. In *CIKM '02: Proceedings of the ACM 11th Conference on Information and Knowledge Management*, pages 391–397, 2002.

[17] Vascoda. Einsatz von Suchmaschinentechnologie für die Zusammenführung und Aufbereitung heterogener wissenschaftlicher Fachdatenbanken aus dem Deep Web. Antrag des HBZ vom 29.08.2005. `http://intranet.vascoda.de/fileadmin/vascoda-storage/Steuerungsgremium/Protokolle/SG2005-09-02_Unterlagen.zip`.

[18] Vascoda. Minutes from the Vascoda regulation board meeting. `http://intranet.vascoda.de/fileadmin/vascoda-storage/Steuerungsgremium/Protokolle/SG_2005-09-02_Unterlagen.zip`.

[19] Vascoda. Strategie vascoda. Verabschiedet auf der Sitzung des Steuerungsgremiums am 28.07.2004 in Hannover. `http://intranet.vascoda.de/fileadmin/vascoda-storage/Steuerungsgremium/Strategie/strategievascoda20040728.pdf`.

[20] Vascoda. Vascoda Application Profile Version 1.0. Zur Standardisierung von Metadatenlieferungen an Vascoda. Stand August 2005. `http://www.dl-forum.de/dateien/vascoda_AP_1.0_vorb.pdf`.

Appendix A

Appendix: Lucene, FAST and STARTS Interoperability

Table A.1: Lucene and FAST compatibility with STARTS

Property	Default Value/Description	FAST	Lucene
Source Content Summary			
Stemming	on/off	no +	yes
Stop-words	included/not included	no +	yes
Case sensitive	on/off	no +	yes *
Total number of documents in source	-	yes	yes
List of words	List of words that appear in the source.	no +	yes
Query Modifiers			
<=, >=, !=, <, =, >	If applicable, e.g., for fields like "Date/time-last-modified", default: =	yes	yes
Phonetic (soundex)	no soundex	no	yes *
Stem	no stemming	no	yes *
Thesaurus	no thesaurus expansion	yes	yes *
Right-truncation	the term "as is", without right-truncating it	yes	yes *
Left-truncation	the term "as is", without left-truncating it	yes	yes *
Case-sensitive	case insensitive	no	yes *
Filter Expressions			
AND	-	yes	yes
OR	-	yes	yes
AND-NOT	-	yes	yes
PROX	Specifies two terms, the required distance between them, and whether the order of the terms matters.	yes	yes
Ranking Expressions			
AND	-	yes	yes
OR	-	yes	yes
AND-NOT	-	yes	yes
PROX	Specifies two terms, the required distance between them, and whether the order of the terms matters.	yes	yes
LIST	Simply groups together a set of terms.	no +	yes

*: Can be accomplished/extended by additional implementation
+: At least to our knowledge, this is not supported

Property	Default Value/Description	FAST	Lucene
Information Propagated with a Query			
Drop stop words	Whether the source should delete the stop words from the query or not. A metasearcher knows if it can turn off the use of stop words at a source from the source's metadata. Default: drop the stop words.	no +	yes *
Default attribute set	Default attribute set used in the query, optional, for notational convenience. Default: Basic-1.	yes	yes *
Default language	Default language used in the query, optional, for notational convenience, and overridden by the specifications at the l-string level. Default: no.	yes	yes *
Additional sources	Sources in the same resource, where to evaluate the query in addition to the source where the query is submitted. Default: no other source.	yes *	yes *
Returned fields	Fields returned in the query answer. Default: Title, Linkage.	yes *	yes
Sorting fields	Fields used to sort the query results, and whether the order is ascending ("a") or descending ("d"). Default: score of the documents for the query, in descending order.	yes	yes
Documents returned Min	Minimum acceptable document score. Default: no.	no	yes
Documents returned Max	Maximum acceptable number of documents. Default: 20 documents.	no	yes
Information Propagated with a Result			
Score	The unnormalized score of the document for the query	no	yes
ID (of source)	The id of the source(s) where the document appears	yes	yes
TF (for every term)	The number of times that the query term appears in the document	no	yes
TW (for every term)	The weight of the query term in the document, as assigned by the search engine associated with the source, e.g., the normalized TFxIDF weight for the query term in the document, or whatever other weighing of terms in documents the search engine might use	no	yes
DF (for every term)	The number of documents in the source that contain the term, this information is also provided as part of the metadata for the source	no	yes
DSize	The size of the document in bytes	yes	yes
DCount	The number of tokens, as determined by the source, in the document	yes	yes

*: Can be accomplished/extended by additional implementation
+: At least to our knowledge, this is not supported

Property	Default Value/Description	Required	FAST	Lucene
Source Metadata Attributes				
FieldsSupported	What optional fields are supported in addition to the required ones. Also, each field is optionally accompanied by a list of the languages that are used in that field in the source. Required fields can also be listed here with their corresponding language list.	yes	yes	yes
ModifiersSupported	What modifiers are supported. Also, each modifier is optionally accompanied by a list of the languages for which it is supported at the source. Modifiers like stem are language dependent.	yes	yes	yes
FieldModifierCombinations	What field-modifier combinations are supported. For example, stem might not be supported for the author field at a source.	no	yes	yes
QueryPartsSupported	Whether the source supports ranking expressions only ("R"), filter expressions only ("F"), or both ("RF"). Default: "RF."	no	yes	yes
ScoreRange	This is the minimum and maximum score that a document can get for a query; we use this information for merging ranks. Valid bounds include -infinity and +infinity, respectively.	yes	no +	yes
RankingAlgorithmID	Even when we do not know the actual algorithm used it is useful to know that two sources use the same algorithm.	yes	no	yes
TokenizerIDList	E.g., (Acme-1 en-US) (Acme-2 es), meaning that tokenizer Acme-1 is used for strings in American English, and tokenizer Acme-2 is used for strings in Spanish. Even when we do not know how the actual tokenizer works, it is useful to know that two sources use the same tokenizer.	no	no	yes
SampleDatabaseResults	The URL to get the query results for a sample document collection.	yes	yes *	yes *
StopWordList	-	yes	no +	yes
TurnOffStopWords	Whether we can turn off the use of stop words at the source or not.	yes	no +	yes *
SourceLanguage	List of languages present at the source.	no	yes *	yes *
SourceName	-	no	yes *	yes
Linkage	URL where the source should be queried.	yes	yes *	yes *
ContentSummaryLinkage	The URL of the source content summary; see below.	yes	yes *	yes *
DateChanged	The date when the source metadata was last modified.	no	yes *	yes *
DateExpires	The date when the source metadata will be reviewed, and therefore, when the source metadata should be extracted again.	no	yes *	yes *
Abstract	The abstract of the source.	no	yes *	yes *
AccessConstraints	A description of the constraints or legal prerequisites for accessing the source.	no	yes *	yes *
Contact	Contact information of the administrator of the source.	no	yes *	yes *

*: Can be accomplished/extended by additional implementation
+: At least to our knowledge, this is not supported