POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Enriching search results with semantic metadata

Relatore: Prof. Marco Colombetti
Correlatore: Ing. David Laniado

Tesi di Laurea di:
Giuseppe Alberto Mangano, matricola 665701

Anno Accademico 2008-2009

To Mom and Dad

Abstract

Traditional syntactic-only search, albeit reliable, efficient, and generally acceptable when a very large set of documents is available, is greatly limiting in other cases, especially when dealing with small collections of indexed data.

The project described in this thesis enhances Solr, an open source enterprise search server based on the Lucene Java search library, with Semantic Search techniques. To increase the amount of relevant information delivered to the user, our search engine analyzes and semantically expands terms present in documents and query strings with information obtained from sources such as ontologies and geographical taxonomies. Employing metadata in the form of payloads associated to terms added via semantic expansion, we furthermore ensure control over the ranking process to directly reflect the possible decrease in relevancy of documents retrieved using semantics.

Contents

Chapter 1

Introduction

Most modern search engines base their information retrieval process on a syntactic approach. This method, that has been extensively employed and perfected until today, is based mainly on the simple matching of terms in the search query with the ones present in the documents we intend to search. In most cases, and specifically in scenarios where there is a considerable amount of documents available to search, *Syntactic Search* delivers acceptable results in both quantity and relevancy. This approach, though, fails to retrieve documents that cannot be matched syntactically with the search query. Especially in cases where the amount of documents available are limited, the inclusion of *Semantic Search* methods can greatly improve the number of relevant documents returned. Analyzing the contents of documents and query strings and enriching them semantically it is possible to obtain results that traditional search engines fail to deliver.

This thesis describes a working search engine prototype that extends Apache Solr[1], enhancing traditional yet effective *Syntactic Search* methods with the semantic expansion of terms present in documents and query strings. Employing metadata in the form of payloads associated to terms added in the expansion, we furthermore ensure control over the ranking process to directly reflect the possible decrease in relevancy of documents retrieved using semantics.

In the following chapter we outline briefly the State of the Art of Information Retrieval, with an overview of the Vector Space Model and the Syntactic and Semantic approaches to search. In Chapter 3 we describe the goals of this project and the choices we made regarding the base platforms

[1] an open source enterprise search server based on the Lucene Java search library - http://lucene.apache.org/solr/

and sources of semantic data employed. With Chapter 4 we go more into detail on the implementation of our search engine, and in Chapter 5 we present some examples that show how our search engine can greatly improve the amount of relevant information returned to the user by semantically expanding terms at index and query time. Finally, in Chapter 5 we draw our conclusions and outline possible future developments and expansions to our project.

Chapter 2

State of the Art

2.1 Information Retrieval

The meaning of "information retrieval" can be very broad. Looking for a number in a telephone directory, using a Web Search Engine, reading the desserts listed in the menu of a restaurant are all forms of information retrieval. From an academic perspective, "information retrieval" can be defined as follows:

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers) [Manning et al., 2009].

Based on this definition, information retrieval used to be an activity performed by a small group of people (eg. librarians, city hall employees, etc.). In recent years, though, the world has changed rapidly, and thanks to the World Wide Web, information retrieval is performed every day by hundreds of millions of people worldwide. Furthermore, as information retrieval quickly becomes the preferred way of obtaining information, the traditional database-style search (that accesses structured data such as patient records and product inventories) is becoming obsolete.

The simplest form of document retrieval performed by a computer is a linear scan through all documents to match a specific term that is being searched for. This approach can be greatly limiting if large document collections (eg. on-line web pages) must be processed quickly, or if a ranked output of matches is required (eg. the best answers must appear as the first results). To avoid scanning all the documents for each query, documents are indexed in advance. Once this is done, a binary term-document incidence

matrix is obtained. Terms are the indexed units; they are usually words, but can also be formed by more than one word, for example *New York*.

This incidence matrix can be used to perform queries based on the Boolean Retrieval Model (eg. "Cat AND Dog AND NOT Horse"); depending on whether we look at the matrix rows or columns, we have a vector for every term, which shows the documents it occurs in, or a vector for each document, showing the terms that appear in it.

2.1.1 Vector Space Model

Differently from the Boolean Retrieval Model, in the Vector Space Model a user will mainly use *free text queries* (i.e. just typing one or more words instead of using a specific language and operators to build query expressions), and the system will decide which documents are more relevant to the user's information needs.

The Vector Space Model (VSM), or term vector model, is an algebraic model that represents natural language documents in a formal manner using vectors in a multi-dimensional space which has only positive axis intercepts. It was used for the first time by the SMART Information Retrieval system, that was developed at Cornell University in the 1960s. The VSM formal operational procedure can be divided into three main stages. The first stage is document indexing. Here content containing terms are extracted. The second stage deals with the weighting of indexed terms. The third and final stage is responsible for computing similarities between the input query and the indexed documents.

Document indexing incorporates document preprocessing, which might include techniques such as stopword removal and/or stemming. Non-linguistic methods for indexing have also been implemented, such as probabilistic indexing techniques.

The process of term weighting for the vector space model is usually handled by statistics. There are three main factors involved in term weighting: term frequency (TF) factor, term collection frequency factor and document vector length normalization factor. The final term weight might be constructed from all or a subset of the mentioned factors. The inverse document frequency (IDF), for example, assumes that the importance of a term is proportional to the number of documents the term appears in.

The document similarity is obtained by using associative coefficients based on the inner product of a document vector and a query vector (queries are treated as regular documents), where a word overlap indicates similarity. This inner product is usually normalized. In most cases the cosine

coefficient, which measures the angle between document vectors, is used to measure similarity.

The simplest term weighting schema assumes that term weights are equal to the document term occurrences. The classic vector space model as proposed by Salton, Wong and Yang [Salton et al., 1975], known as the TF-IDF, has both local and global parameters incorporated in the term weight equation, and allows seldom terms (considered good discriminators) to obtain a higher weight.

2.1.2 Precision and Recall

One of the most challenging issues in an always more information-driven society is increasing the *effectiveness* of an information retrieval system that a user typically interacts with by submitting a *query*, in an attempt to communicate an information need. A document returned by the system is *relevant* if the user perceives it as containing information of value with respect to his personal information need.

Two key statistics about the system's returned results for a query, that allow us to assess its effectiveness, are:

- **Precision**: "what fraction of the results returned are relevant to the information need?"
- **Recall**: "what fraction of the relevant documents in the collection were returned?"

In the field of information retrieval, Precision is the fraction of retrieved documents that are relevant to the search query.

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \quad (2.1)$$

Precision takes into account all retrieved documents, but it can also be assessed at a given cut-off rank, considering only the topmost results returned by the system when performing a query. For example, for a text search on a set of documents Precision is the number of correct results divided by the number of all the results returned. It is important to note that the meaning and usage of the term *Precision* in the field of Information Retrieval differs from definitions of accuracy and precision within other branches of science and technology.

Recall in information retrieval is the fraction of the documents that are relevant to the search query that are successfully retrieved by the system.

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \quad (2.2)$$

For example, for text search on a set of documents, Recall is the number of correct results divided by the number of results that should have been returned. It is trivial to obtain recall of 100% by returning all the documents available in response to any search query. For this reason, Recall alone is insufficient to assess the quality of a search result; it is also necessary to measure the number of non-relevant documents (eg. by computing the Precision).

2.1.3 Syntactic Search

One of the two major approaches to information retrieval is *Syntactic Search*. A syntactic search engine will perform computations aimed at spotting string similarity between terms, the atomic elements, that can be either single words or multi-word phrases. In general, the results of a syntactic search have a good Recall but a low Precision. Syntactic Search, being the first and currently the most used search method, has been thoroughly studied and perfected. Traditional syntactic search engines are mainly based on three phases (see, among others, *The Anatomy of a Large-Scale Hypertextual Web Search Engine* [Brin and Page, 1998]):

- crawling time: it is the phase in which the resources (HTML pages, multimedia contents, etc.) are collected in order to build a coherent (and more homogeneous) set;
- indexing time: it is the phase during which the crawled resources are parsed and indexed in some particular data structures; those structures are built based on the relevant information contained in indexed resources and are optimized to quickly answer to queries (granting search response-time in the order of milliseconds);
- searching time: it is the run-time phase in which final users submit their queries in order to retrieve meaningful results; in addition to the optimization of the indexes, this phase requires also a good method to rank and/or cluster search results.

2.1.4 Semantic Search

Semantic Search, on the other hand, is based on the computation of semantic relations between concepts. Differently from the Syntactic Search method,

Semantic Search exploits the meaning of words. Semantic Search can solve several known problems that Syntactic Search faces, like Polysemy (when a term has multiple meanings) and Synonymy (when two terms have the same meaning).

Semantic Search can improve search by using data from semantic networks to disambiguate queries and document content in order to generate more relevant results. Semantic networks, that represent semantic relations between various concepts, are direct or indirect graphs of various kinds, consisting of vertices, which represent concepts, and edges.

The paper *Concept Search* [Giunchiglia et al., 2008] presents an interesting model aimed at improving the quality of search results using the semantic relationships between concepts.

A similar project is presented in *Hybrid Search: Effectively Combining Keywords and Semantic Searches* [Bhagdev et al., 2008], a search method supporting both document and knowledge retrieval via the combination of ontology-based search and keyword-based matching.

Another interesting approach in introducing semantics in indexing and searching is *ALVIS* [Buntine, 2005], an open-source semantic-based peer-to-peer search engine which, before indexing the documents, enriches them with some "semantic awareness" of the specific subject by using information extraction technology.

2.1.5 Index and query expansion

Data collected from sources such as ontologies, thesauri, taxonomies, etc. can be used to expand indexed terms (index expansion) as well as terms used in queries (query expansion) in order to increase the number of relevant documents returned. When performing and index expansion, we associate to certain terms of a document other terms obtained via semantic expansion. This way, if the user searches for terms that we happen to have added when expanding, the document can be retrieved by matching the searched terms with the semantic ones.

A query expansion consists in expanding the search query of the user to match additional documents already indexed. A prototype that uses semantic query expansion to improve search results is described in the paper *Squiggle: a Semantic Search Engine for indexing and retrieval of multimedia content* [Celino et al., 2006].

The result of index and query expansion is an increase of Recall, while the Precision depends on how well our choice of terms used for the expansion reflects the information needs of the user. As mathematically equated, Pre-

cision should decrease, but can potentially increase if the documents added to the result set are more relevant (i.e. of higher quality) or at least equally relevant.

Chapter 3

Goals and Platform Choices

3.1 Goals

Traditional search engines are based on the syntactic matching of search query terms with terms contained in the indexed documents. This approach can be acceptable when a very large set of documents is available, but is greatly limiting in other cases. For example, a simple phrase like:

```
bed and breakfast in Milan
```

can be easily retrieved using as query terms words contained in the document, but even though this document could still be relevant to the interests of a user searching for *accommodation* and *Italy*, it cannot be retrieved with syntactic matching if the query terms aren't contained in the indexed document.

One way to enable the retrieval of the above document when searching for *Italy* is to reformulate the search query to contain the term *Milan*. This process is known as *query expansion*. Therefore, if the user searches for *Italy*, adding Italy's major cities as query terms would ensure that the document is retrieved. In most cases, though, this approach is increasingly costly in terms of system resources as we try to increment the number of documents retrieved.

On the other hand, a much more efficient system can be obtained by expanding terms semantically at index time. Considering the document used in the above example, if we add to the indexed words the geographical hierarchy locations above Milan:

```
bed and breakfast in Milan Lombardy Italy Europe
```

a search for *Italy* would retrieve the expanded document.

An obvious requisite of such a system is that the document is delivered to the user in its original form (i.e. without the terms we added to render it retrievable). When the output of a search query is a ranked list of documents, it is also important to have control over the individual scores. This way, if documents like:

```
Italy travel guide
```

are present in the index, when searching for *Italy* it is possible to associate a higher score to matches with terms of the original document than matches with terms added semantically.

The goal of this project is to provide a platform based on the Vector Space Model that allows exploiting, when available, semantic information of various kinds to further increase the amount of relevant information delivered to the user.

To perform semantic expansion without losing the advantages of the Vector Space Model, when we have a match with an indexed term that has been added semantically the score of the document won't be the same as if the term were present in the original document. The matched document's score, that is used when displaying ranked results, is multiplied by a factor that will directly influence the rank of the document among the other returned results.

This factor will be associated to each term added semantically as metadata, in the form of a payload that can be added at index and query time. This way, semantic terms that are less relevant (taken, for example, from higher levels in a taxonomy) can be added with decreasing multiplying factors smaller than 1 to ensure that less interesting matches appear in the results with a lower ranking.

The project will initially focus on designing and implementing the core platform components, as well as providing index expansion for specific domains such as geographical hierarchies and simple ontologies.

It is possible to further increase the number of relevant documents retrieved by enabling the semantic expansion of the query string used to search the index. In this case, the number of hierarchy levels used for the expansion must be limited to prevent the retrieval of irrelevant documents.

3.2 Platform choices

3.2.1 Open source software

Open source software is defined as computer software of which the source code and other rights usually reserved for the copyright holders are provided under a software license that meets the Open Source Definition[1] or that is in the public domain. This permits users to use, change, and improve the software, and to redistribute it in its modified or original form. The development process of open source software very often proceeds in a public, collaborative manner.

With his 1997 essay *The Cathedral and the Bazaar*[Raymond, 1997], open source software advocate Eric S. Raymond proposes a model for the development of Open source software known as the Bazaar model. Raymond compares the traditional method of software development to building a cathedral, "carefully crafted by individual wizards or small bands of mages working in splendid isolation". He then suggests that all software should be developed using the bazaar style, which he described as "a great babbling bazaar of differing agendas and approaches".

According to the Cathedral model, development is organized in a centralized way, and the roles are clearly defined: some people are dedicated to designing, others are responsible for managing the project, and another group of people is responsible for the implementation. The traditional software engineering process follows the Cathedral model.

The Bazaar model is quite different. In this model, the roles assumed by people involved in the development process are not clearly defined. Gregorio Robles [Gehring and Lutterbeck, 2004] suggests that software developed with the Bazaar model should exhibit the following patterns:

- **Users should be treated as co-developers** - The users are treated like co-developers and so they should have access to the source code of the software. Furthermore users are encouraged to submit additions to the software, code fixes for the software, bug reports, documentation etc. Having more co-developers increases the rate at which the software evolves. Linus's law states that, "Given enough eyeballs all bugs are shallow". This means that if many users view the source code they will eventually find all bugs and suggest how to fix them. Note that some users have advanced programming skills, and furthermore, each user's machine provides an additional testing environment. This new testing environment offers that ability to find and fix a new bug.

[1] http://www.opensource.org/docs/osd

- **Early releases** - The first version of the software should be released as early as possible so as to increase one's chances of finding co-developers early.
- **Frequent integration** - New code should be integrated as often as possible so as to avoid the overhead of fixing a large number of bugs at the end of the project life cycle. Some open source projects have nightly builds where integration is done automatically on a daily basis.
- **Several versions** - There should be at least two versions of the software. There should be a buggier version with more features and a more stable version with fewer features. The buggy version (also called the development version) is for users who want the immediate use of the latest features, and are willing to accept the risk of using code that is not yet thoroughly tested. The users can then act as co-developers, reporting bugs and providing bug fixes.
- **High modularization** - The general structure of the software should be modular allowing for parallel development.
- **Dynamic decision making structure** - There is a need for a decision making structure, whether formal or informal, that makes strategic decisions depending on changing user requirements and other factors.

The project described in this thesis adds additional functionality to already existing Open source software. The software developed is released under the Apache License, Version 2.0[2].

3.2.2 Apache Lucene

Apache Lucene is a free/open source information retrieval library originally created in Java (and ported to several other programming languages, such as Delphi, Perl, C#, C++, Python, Ruby and PHP). It is supported by the Apache Software Foundation and released under the Apache Software License. Lucene is suitable for any software application that requires full text indexing and searching, and specifically for implementing internet search engines and local single-site searching.

Lucene's API is file format independent because its core logical architecture is based on the concept of documents containing fields of text. Therefore, text from any document (eg. of Microsoft Word, OpenDocument, HTML, PDFs) can be indexed as long as the contained text can be extracted.

[2]http://www.apache.org/licenses/LICENSE-2.0.html

Developers using Lucene can count on powerful features[3] such as ranked searching (best results returned first), several query types (eg. phrase queries, wildcard queries, proximity queries, range queries), fielded searching (eg. title, author, contents), date-range searching, sorting by any field and many others.

3.2.3 Apache Solr - Tomcat

Solr is an open source standalone enterprise search server based on Lucene. It was initially developed by CNET Networks as an in-house project aimed at adding search capability to the company website. In early 2006, CNET Networks donated the source code to the Apache Software Foundation. At the beginning of 2007, Solr graduated from the initial incubation period (that all new projects at the Apache Software Foundation are put through to help solve organizational, legal and financial issues) and started to steadily grow, accumulating features thanks to the contributions of the robust community of users that it attracted. Although quite new as a public project, it is already being used successfully by many high-traffic websites[4] such as Digg, Netflix, News.com, Gamespot, reddit.

Documents are added to Solr (a process called "indexing") via XML over HTTP. The server is then queried via HTTP GET and the results are received in XML. Solr's scalability, flexibility, advanced full-text search capabilities and extensible plugin architecture are only some of the numerous features[5] that made this server the ideal platform to work on.

To develop the software described in this thesis we used what is currently the latest stable version (1.3.0, that includes Lucene 2.4-dev). This version requires Java 1.5 and an Application server that supports the Servlet 2.4 standard; for the latter we chose Apache Tomcat (version 5.5), developed by the Apache Software Foundation.

3.2.4 Geonames - DOM

The first set of semantic expansions were performed using geographical hierarchy information. This was done using real data obtained from GeoNames, a geographical database available and accessible free of charge through various Web Services, under a Creative Commons attribution license. GeoNames covers all countries and contains over eight million placenames and

[3] http://lucene.apache.org/java/docs/features.html
[4] http://wiki.apache.org/solr/PublicServers
[5] http://lucene.apache.org/solr/features.html

other data such as latitude, longitude, elevation, population, administrative subdivision, and postal codes.

The data needed for the semantic expansion was obtained in XML format, and parsed using the interfaces for the W3C Document Object Model (DOM), which is a component API of the Java API for XML Processing[6].

3.2.5 Protégé - JENA

Another set of expansions were performed by parsing a custom ontology created with Protégé (version 3.4), a free open-source Ontology Editor and Knowledge Acquisition System.

The Ontology was then accessed using Jena (version 2.6.0), an open source Semantic Web framework for Java that provides an API to extract data from and write to RDF graphs.

[6]http://www.j2ee.me/javase/6/docs/api/org/w3c/dom/package-summary.html

Chapter 4

Implementation

4.1 Lucene's architecture

Lucene is a high-performance, full-featured, open-source, text search engine API written in Java. The heart of Lucene is the Index, that is populated with the data we intend to search to obtain results. The first step is therefore adding data to the Index. When Lucene indexes text, part of this phase is cleaning up (or modifying at will) the text with an analyzer. Lucene provides a few default analyzers, and custom ones can be used too. In most cases, it is best that the same analyzer is used for both indexing and searching. For an in-depth analysis of this issue, see Chapter 4.1 of *Lucene in Action* [Gospodnetic and Hatcher, 2005].

Lucene is an API, not an application, therefore even though it handles the indexing, searching and retrieving of documents it doesn't handle the selection of data files, the retrieval of the search string, and displaying of the search results. Because of this, servers such as Solr are usually employed to create a functioning search engine. Fig. 4.1 shows the architecture of a typical Lucene-based search platform.

Textual data extracted from sources such as databases, websites and emails are first processed by an analyzer that tokenizes the text strings, filters the tokens and adds them to the index. Even the user's query is processed by an analyzer, that is usually similar to the one employed during the indexing phase. Matches between the processed query and the contents of the index are then displayed to the user.

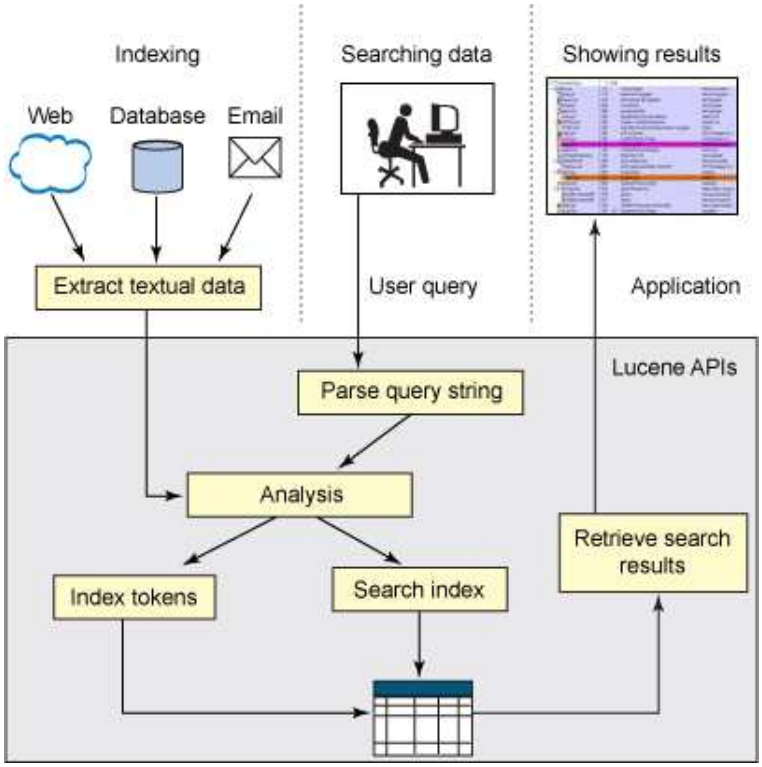

Figure 4.1: a Lucene-based search platform (image by Amol Sonawane)

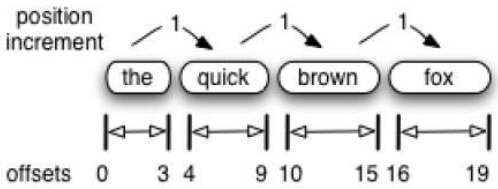

Figure 4.2: a token stream example (from Erik Hatcher's Solr Boot Camp)

4.1.1 Lucene's Tokens

A stream of tokens (see Fig. 4.2) is the fundamental output generated by the analysis process. At index time, fields set for tokenization are processed with the configured analyzer, and every token is written to the index as a term.

Each token represents an individual word of that text. A token carries with it a text value (the word itself) as well as some metadata: the start and end offsets in the original text, a token type, a position increment and an optional payload.

The start offset is the character's position in the original text where the token text begins, and the end offset is the position just after the last character of the token text. The token type is a String, with "word" as default value, that can be controlled and used in the token-filtering process as desired. As text is tokenized, the position relative to the previous token is registered as the position increment value. All of Lucene's built-in tokenizers leave the position increment at the default value of 1, indicating that all tokens are placed in successive positions, one after the other.

The token position increment value relates the current token to the previous one. Usually, position increments are 1, indicating that every word is in a unique and successive position in the field. Position increments greater than 1 allow for gaps and can be used to indicate where words have been removed (as a result, for example, of stop-word removal). A token with a zero position increment places the token in the same position as the previous token. Analyzers that inject word aliases (like the one we implemented for this project) can use a position increment of zero for the aliases.

4.2 Solr's architecture

Solr allows the loading of custom code to perform a variety of tasks. To implement our project we create a custom analyzer that supports two parsers

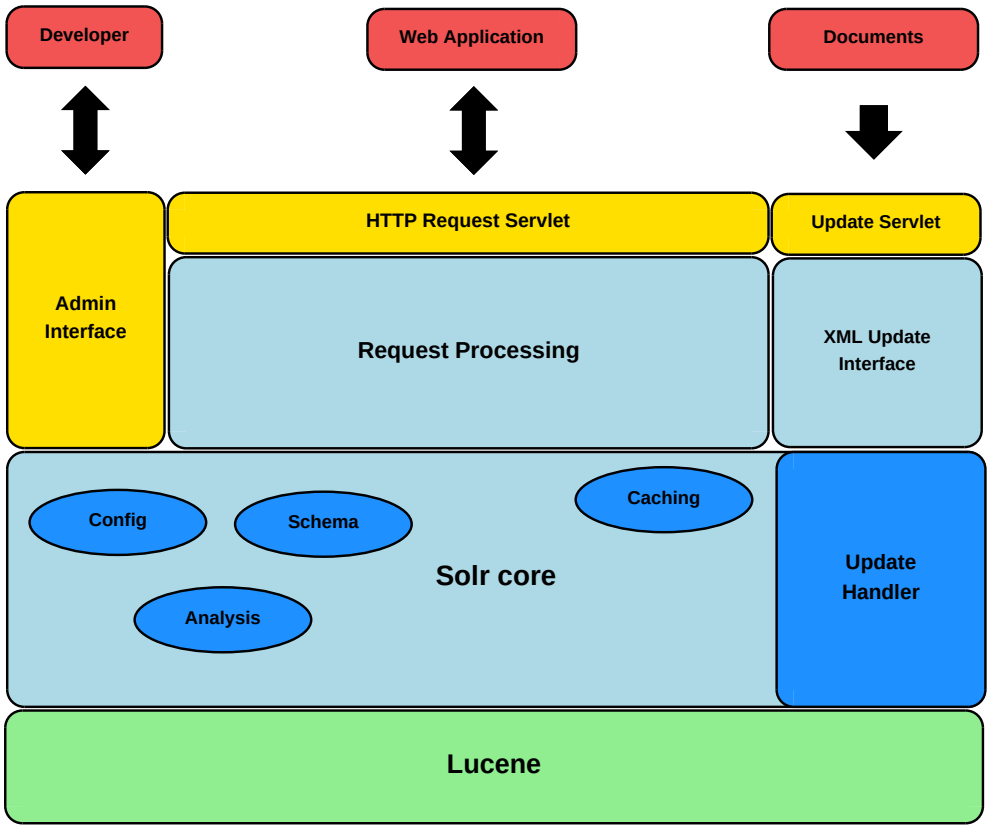

Figure 4.3: Solr's architecture

(GeoNames parser and Ontology parser), as well as two custom request processing components: a Similarity and a Query Parser (all described in the following sections). A general overview of Solr's architecture is described in Fig. 4.3.

4.3 Solr custom analyzer - SemanticFilter

In Solr, to define the way a field type is analyzed, you can edit the schema.xml configuration file, specifying for each fieldType two analyzers: an index-time and a query-time analyzer. For each analyzer, you can specify a TokenizerFactory followed by a list of optional TokenFilterFactories, that will be applied in the listed order (for an example configuration see Fig. 5.6).

From the default TokenizerFactories included in Solr we chose WhitespaceTokenizerFactory, that creates tokens of characters separated by split-

ting on white space occurrences. A Token[1] is the occurrence of a term in the text of a field. It consists of a term's text, the start and end offset of the term in the text of the field, and a type string.

A TokenFilterFactory[2] creates a TokenFilter to transform one TokenStream into another.

Solr includes several useful TokenFilterFactories (see TokenFilterFactory's documentation for a complete list) such as:

- StandardFilterFactory: removes dots from acronyms and 's from the end of tokens
- LowerCaseFilterFactory: lowercases the letters in each token;
- StopFilterFactory: discards common words. Besides the default English stop words (such as "a", "an", "and", "are", etc), an optional list can be specified;
- KeepWordFilterFactory: the opposite of StopFilterFactory;
- SynonymFilterFactory: matches strings of tokens and replaces them with other strings of tokens.

The first component needed to perform the semantic expansion is a custom TokenFilter, that we named SemanticFilter, and its SemanticFilterFactory (that extends Solr's BaseTokenFilterFactory).

When SemanticFilterFactory is listed after WhitespaceTokenizerFactory as one of the TokenFilterFactories, one of the two developed parsers (described in the next paragraphs) is specified via the "parser" argument. This way, multiple instances of SemanticFilter can be invoked, each one with a different parser. Our SemanticFilter extends Lucene's CachingTokenFilter, that caches all Tokens locally in a List. Each instance will only expand Terms that have not been processed by previous instances. This is done by checking the type of each Token: all consecutive Tokens with the default type value ("word") are gathered in an ordered group and parsed with the selected parser. Each parser will group the Tokens into shingles (groups of contiguous words that form a term, like *New York* and *bed and breakfast*), and if the *onlyTokenize* option (one of the arguments of SemanticFilterFactory) is set to "false" the semantic expansion will be performed by creating semantic Tokens and adding them to the same position as the one being expanded.

[1] http://lucene.apache.org/java/2_3_0/api/org/apache/lucene/analysis/Token.html
[2] http://lucene.apache.org/solr/api/org/apache/solr/analysis/TokenFilterFactory.html

The type of the original term (be it single-word or multi-word) will then be changed to "processed". The types of the added semantic Tokens are parser-dependant; listing the types in the *spaceSeparatedTypes* argument and the correspondent float values in the *spaceSeparatedBoosts* argument makes it possible to set a custom boost for each kind of semantic term.

4.3.1 GeoNames parser

Using the GeoNames Search Webservice, each group of tokens received by the parser is analyzed to recognize all single and multi-word terms (shingles) that identify a geographical location, be it a Country, a State, a City, etc... The identified terms are then expanded semantically with the names of locations that constitute the geographical hierarchy above them. The sample document *bed and breakfast in Milan* would therefore be expanded as follows:

```
[bed] [and] [breakfast] [in] [Milan]
                             [Lombardy]
                             [Italy]
                             [Europe]
```

The current implementation limits searches to "featureClass=A" (country, state, region,...) and accepts as valid match the first returned result (Polysemy managing is currently not supported, but quite easy to implement). To identify multi-word terms, a combination of the "name" and "name_equals" arguments is used to check for partial and exact matches. The shingle matching algorithm is described further on, in paragraph 4.3.3.

For example, checking for an exact match of the term *New York* translates into the following request:

```
http://ws.geonames.org/search?name_equals=New%20York&featureClass=A
```

This request returns an XML file with the output of the query, that we parse using the Java API for XML Processing[3]. Once the parsing process has been completed, for each identified location we have obtained (from the XML output) its unique GeoNames identifier (geonameId). These identifiers are used to retrieve the geographical hierarchy from GeoNames via the Hierarchy Web Service; for example, New York's hierarchy is obtained via the following request:

```
http://ws.geonames.org/hierarchy?geonameId=5128638
```

[3]http://www.j2ee.me/javase/6/docs/api/org/w3c/dom/package-summary.html

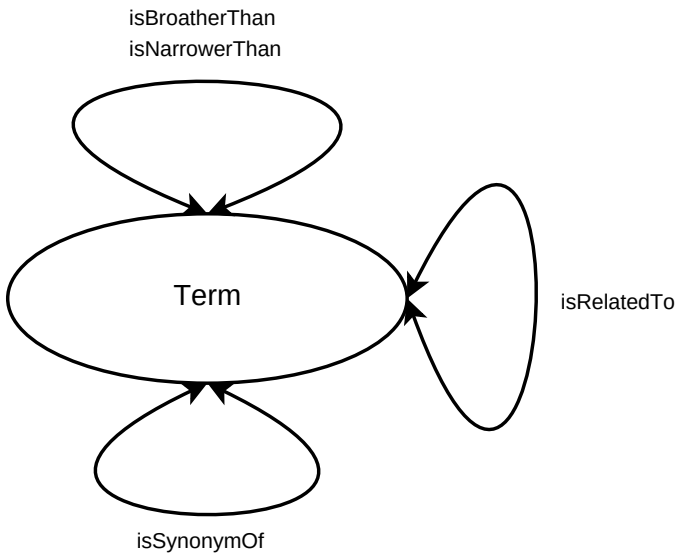

Figure 4.4: our test ontology

This other XML file is parsed too to obtain the data needed to create the appropriate semantic Tokens.

Unmatched Tokens are left untouched, with the token type set to "word", and matched Tokens are marked as type: "processed". Tokens added with the semantic expansion are created as type: "geonames-hierarchy-x", where x stands for the level of expansion; the number of levels with specific boosts can be specified in schema.xml, for both index and query phases. For debugging purposes, the geonames unique identifier is added as a Token too, and marked as type: "geonames-id".

4.3.2 Ontology parser

Using the Protégé Ontology Editor we created a simple ontology with only one class, *Term*, and the properties: *isBroaderThan, isNarrowerThan, isRelatedTo, isSynonymOf*. All four properties have "Term" as domain and target. The first two are one the inverse of the other, and the last two are symmetric (see Fig. 4.4).

We then populated the class with some individuals and created relationships between them, like:

```
dog isBroaderThan poodle
dog isNarrowerThan pet
pet isBroaderThan cat
```

```
pet isNarrowerThan animal
hotel isNarrowerThan accommodation
bed_and_breakfast isNarrowerThan accommodation
bed_and_breakfast isRelatedTo sleep
hotel isBroaderThan hilton
```

Unmatched Tokens are left untouched, with the token type set to "word"; matched Tokens are marked as type: "processed", and the Tokens added with the semantic expansion are created with a Token type that identifies one of the four properties listed above. If the token was added via the *isNarrowerThan* property, the token type will have the format "isNarrowerThan-x", where x stands for the level of expansion; each level can have a different boost, that will be set in schema.xml with the other parameters.

4.3.3 Shingle matching - the algorithm

To recognize multi-word units that can be expanded, a shingle matching algorithm was designed, and a custom implementation of it was created for each parser. A flowchart representation of the matching process, created with Dia[4], can be viewed in Fig. 4.5.

The algorithm starts building a temporary shingle by adding the first token of the stream. It then attempts to find an exact match among the terms of our semantic data. If the match is successful, it checks to see if other tokens can be appended to the shingle. If there aren't any more tokens to append, the shingle is saved for expansion and the algorithm terminates; if there are more tokens, the current shingle is recorded as a multi-word candidate, the next token of the stream is appended, and the algorithm attempts another exact match.

If an exact match fails, a partial match is attempted. If it is successful, the algorithm verifies if there are tokens left. If there are, the next token is appended and the algorithm performs another exact match attempt; if there aren't any more tokens to append, the algorithm checks if there is a multi-word candidate available. If there is, the candidate is recognized as a valid multi-word unit, and the temporary shingle is cleared. At this point, if there are tokens left to process, the algorithm adds the next one to a new temporary shingle (that happens to be the first available token after the ones of the candidate saved for expansion) and goes back to perform another exact match attempt. If there isn't a candidate available, the shingle is cleared and the first token (that was at the head of our temporary shingle)

[4]http://dia-installer.de/index_en.html

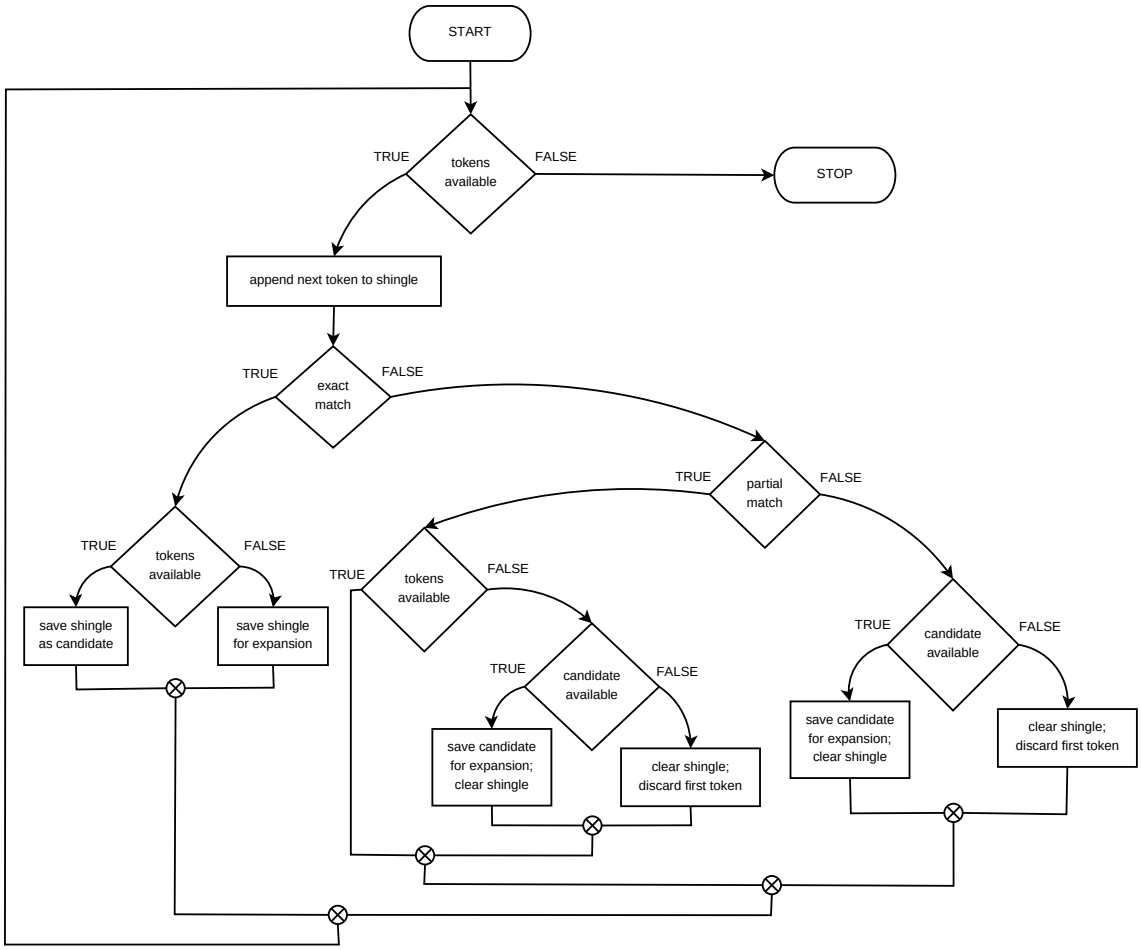

Figure 4.5: shingle matching algorithm

is discarded (because it wasn't matched by the algorithm); at this point, if there are tokens left, the algorithm resumes (adds the next token, attempts an exact match, etc.); if not, it ends.

When a partial match attempt fails, the algorithm checks the availability of a multi-word unit candidate. If there is, the candidate is recognized as a valid multi-word unit, and the shingle is cleared (the algorithm then verifies if there are tokens left to start building a new temporary shingle). If there isn't a candidate available, the shingle is cleared and the first token is discarded. At this point, if there are tokens left the algorithm starts building a new temporary shingle; elsewise, it stops.

4.3.4 Payloads

Payloads let Lucene users optionally store a byte array of information on a term by term basis. A Payload is metadata that can be stored together with each term occurrence. To store payloads in the index a TokenStream that produces Tokens containing payload data has to be used. At the time of this writing, the status of Lucene's Payloads feature is experimental, and the current APIs are subject to possible changes.

Using the encodeFloat method of Lucene's PayloadHelper class and the setPayload method of the Token class, our SemanticFilter adds as payloads of the semantic Tokens the float values of the *spaceSeparatedBoosts* option specified in schema.xml. These float values will be used to fine-tune the scoring of our semantic Term matches: values greater than 1 will increase their rank, while values smaller than 1 will decrease it.

A the time of this writing, Solr doesn't have query side support for payloads. An issue report regarding this missing functionality is present in JIRA[5].

- Key: SOLR-1337
- Type: New Feature
- Status: Open
- Priority: Major
- Created: August 5th, 2009

4.4 Subclassing Similarity - PayloadsBoostingSimilarity

In Solr's schema.xml configuration file it is possible to use a <similarity> declaration to specify the subclass of Similarity that we want to use when dealing with the index. If no Similarity class is specified, the Lucene DefaultSimilarity is used. The Similarity class[6] is a native Lucene concept that determines how much of the score calculations for the various types of queries is executed.

The score of query **q** for document **d** correlates to the cosine-distance or dot-product between document and query vectors in a Vector Space Model

[5] https://issues.apache.org/jira/browse/SOLR-1337

[6] http://lucene.apache.org/java/2_4_0/api/org/apache/lucene/search/Similarity.html

(VSM) of Information Retrieval. A document whose vector is closer to the query vector in that model is scored higher. The score is computed as follows:

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t\ in\ q} (tf(t\ in\ d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d)) \tag{4.1}$$

where:

1. tf(t in d) correlates to the term's frequency, defined as the number of times term t appears in the currently scored document d. Documents that have more occurrences of a given term receive a higher score. The default computation for tf(t in d) in DefaultSimilarity is:

$$tf(t\ in\ d) = frequency^{1/2} \tag{4.2}$$

2. idf(t) stands for Inverse Document Frequency. This value correlates to the inverse of docFreq (the number of documents in which the term t appears). This means rarer terms give higher contribution to the total score. The default computation for idf(t) in DefaultSimilarity is:

$$idf(t) = 1 + log(\frac{numDocs}{docFreq+1}) \tag{4.3}$$

3. coord(q,d) is a score factor based on how many of the query terms are found in the specified document. Typically, a document that contains more of the query's terms will receive a higher score than another document with fewer query terms. This is a search time factor computed in coord(q,d) by the Similarity in effect at search time.

4. queryNorm(q) is a normalizing factor used to make scores between queries comparable. This factor does not affect document ranking (since all ranked documents are multiplied by the same factor), but rather just attempts to make scores from different queries (or even different indexes) comparable. This is a search time factor computed by the Similarity in effect at search time. The default computation in DefaultSimilarity is:

$$queryNorm(q) = queryNorm(sumOfSquaredWeights) = \frac{1}{sumOfSquaredWeights^{1/2}} \tag{4.4}$$

The sum of squared weights (of the query terms) is computed by the query Weight object. For example, a boolean query computes this value as:

$$sumOfSquaredWeights = q.getBoost()^2 \cdot \sum_{t\ in\ q} (idf(t) \cdot t.getBoost())^2 \tag{4.5}$$

5. t.getBoost() is a search time boost of term t in the query q as specified in the query text (see query syntax), or as set by application calls to setBoost(). Notice that there is really no direct API for accessing a boost of one term in a multi term query, but rather multi terms are represented in a query as multi TermQuery objects, and so the boost of a term in the query is accessible by calling the sub-query getBoost().

6. norm(t,d) encapsulates a few (indexing time) boost and length factors:

- Document boost - set by calling doc.setBoost() before adding the document to the index.
- Field boost - set by calling field.setBoost() before adding the field to a document.
- lengthNorm(field) - computed when the document is added to the index in accordance with the number of tokens of this field in the document, so that shorter fields contribute more to the score. LengthNorm is computed by the Similarity class in effect at indexing.

When a document is added to the index, all the above factors are multiplied. If the document has multiple fields with the same name, all their boosts are multiplied together:

$$norm(t,d) = doc.getBoost() \cdot lengthNorm(field) \cdot \prod_{field\ f\ in\ d\ named\ as\ t} f.getBoost() \quad (4.6)$$

However the resulted norm value is encoded as a single byte before being stored. At search time, the norm byte value is read from the index directory and decoded back to a float norm value. This encoding/decoding, while reducing index size, comes with the price of precision loss - it is not guaranteed that decode(encode(x)) = x. For instance, decode(encode(0.89)) = 0.75. Also notice that search time is too late to modify this norm part of scoring, e.g. by using a different Similarity for search.

To manage the scoring of semantic Terms we created a custom Similarity class, PayloadsBoostingSimilarity, that extends Lucene's DefaultSimilarity. In it, we override the scorePayload function (which returns 1 by default), and use PayloadHelper's decodeFloat method to extract from the payload and return our boost value (that will be retrieved by Lucene's BoostingTermQuery, described in paragraph 4.5). A PayloadsBoostingSimilarityFactory was implemented (to create our custom Similarity) and added to schema.xml.

4.5 BoostingTermQuery

To invoke the (overridden) scorePayload method of PayloadsBoostingSimilarity we use BoostingTermQuery[7], currently the only payload aware Query available in Lucene. BoostingTermQuery is very similar to the SpanTermQuery[8], except that it factors in the value of the payload located at each of the positions where the Term occurs. In the current implementation of BoostingTermQuery (left untouched), payload scores are averaged across Term occurrences in the document.

4.6 Custom query parser - PayloadsQParserPlugin

Solr's QParserPlugin[9] can be used to create custom query structures for Solr. To handle the payloads of the semantic Tokens we created PayloadsQParserPlugin, that extends QParserPlugin and uses Lucene's BoostingTermQuery to search for individual Terms specified in the user's query.

Specifying the same sequence of SemanticFilter instances in the query-time analyzer (via schema.xml) the query expression will be broken down into the same shingles that are created at index time. To disable query time expansion the *onlyTokenize* option must be set to "true". When setting it to "false", the number of hierarchy levels used in the expansion must be set to prevent unwanted matches. Not limiting the levels could allow unwanted documents to appear in query results; a search for *Sicily*, for example, would retrieve documents containing the term *Santiago De Compostela* because both locations have *Europe* in their full semantic expansions.

The current implementation of PayloadsQParserPlugin performs queries for multiple terms (both single and multi-word) using the boolean OR operator. This has been implemented declaring in our custom QParserPlugin a BooleanQuery[10] and adding to it the individual BoostingTermQueries as "should occur" BooleanClauses[11].

[7] http://lucene.apache.org/java/2_2_0/api/org/apache/lucene/search/payloads/BoostingTermQuery.html
[8] http://lucene.apache.org/java/2_2_0/api/org/apache/lucene/search/spans/SpanTermQuery.html
[9] http://lucene.apache.org/solr/api/org/apache/solr/search/QParserPlugin.html
[10] http://lucene.apache.org/java/2_3_1/api/org/apache/lucene/search/BooleanQuery.html
[11] http://lucene.apache.org/java/2_3_1/api/org/apache/lucene/search/BooleanClause.html

Chapter 5

Results

5.1 Examples

In this section we use Solr's analysis administration capabilities (located by default at *http://localhost:8080/solr/admin/analysis.jsp*) to conveniently create some semantic expansion examples. The Analysis page accepts snippets of text for both queries and documents, as well as the Field name that identifies how the text should be analyzed, and returns stepwise results of the text as it is being modified. After illustrating an index expansion example, we provide a scoring output sample (as XML output) and an example of how expanding the submitted query string can further increase the number of relevant documents retrieved.

5.1.1 Index expansion

To expand semantically our indexed documents, we have to configure properly our index analyzer in the schema.xml configuration file, as shown in Fig. 5.1.

The first option that needs to be set is the tokenizer class we intend to use. For our examples we chose one of the most commonly used tokenizers: the WhitespaceTokenizer. We therefore set the tokenizer class to *solr.WhitespaceTokenizerFactory*. We then listed two instances of our SemanticFilter: the first one will use the GeoNames parser, and the second one the Ontology parser, in this order. The GeoNames parser, besides adding the geonames-id as a token with 0.1F as boost value, will expand identified locations four levels up their geographical hierarchy. The tokens added will have the boost values specified with the *spaceSeparatedBoosts* option:

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="geonames-parser-2"
        spaceSeparatedTypes="geonames-id geonames-hierarchy-1 geonames-hierarchy-2 geonames-hierarchy-3 geonames-hierarchy-4"
        spaceSeparatedBoosts="0.1F 0.4F 0.16F 0.064F 0.0256F"
        onlyTokenize="false"
        />
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="ontology-parser-2"
        spaceSeparatedTypes="isNarrowerThan-1 isNarrowerThan-2 isNarrowerThan-3 isRelatedTo isSynonymOf"
        spaceSeparatedBoosts="0.4F 0.16F 0.064F 0.4F 0.7F"
        onlyTokenize="false"
        />
    </analyzer>
```

Figure 5.1: schema.xml index analyzer settings for index time expansion

- geonames-hierarchy-1: 0.4F
- geonames-hierarchy-2: 0.16F
- geonames-hierarchy-3: 0.064F
- geonames-hierarchy-4: 0.0256F

For our test case we have chosen to multiply by 0.4 the boost value as we go up in hierarchy level to reflect the decrease of relevancy. The values of the *spaceSeparatedTypes* option will be used by the SemanticFilter to set the token types. The Ontology parser will expand identified terms three levels up in their taxonomy hierarchy, and will also add terms that are connected to them in our test ontology via the *isRelatedTo* and *isSynonymOf* relationships. The semantic tokens obtained with the Ontology parser will have the following types and boosts:

- isNarrowerThan-1: 0.4F
- isNarrowerThan-2: 0.16F
- isNarrowerThan-3: 0.064F
- isRelatedTo: 0.4F
- isSynonymOf: 0.7F

We chose to decrease the boosts as we go up in hierarchy level like we did with the GeoNames parser, to influence coherently the scoring of matched documents. The *onlyTokenize* option is set to *false* in both instances of our SemanticFilter, because we don't need to disable the semantic expansion of the indexed documents.

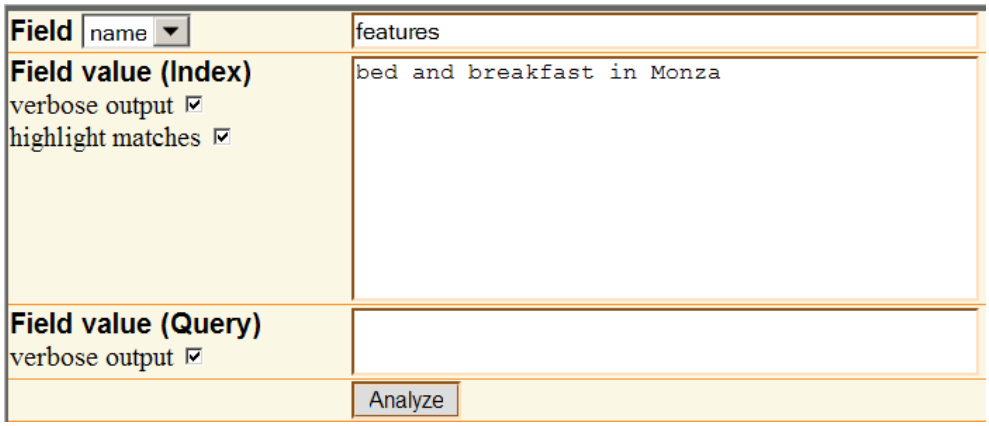

Figure 5.2: sample document input for index time expansion

To test the semantic expansion at index time we use the Field Analysis function of Solr's analysis administration interface. The data input of our sample document is displayed in Fig. 5.2.

For the *field name* option we specify the *features* field, that in schema.xml is described as being of fieldType *text*. We then insert in the *Field value (Index)* field the sample document:

```
bed and breakfast in Monza
```

After enabling all the extra options (*verbose output* and *highlight matches*), we click **Analyze** to see the indexing steps performed by our analyzer.

Our sample document is first broken down into tokens by the WhitespaceTokenizer as seen below:

```
[bed] [and] [breakfast] [in] [Monza]
```

This fist step is displayed in Fig. 5.3.

Each token occupies one *term position*, has its *term text*, its *term type* (currently set to *word*, the default value) and information on starting and ending characters (*source start,end*). Currently no payloads have been created.

At this point, our SemanticFilter uses the GeonamesParser to expand the token *Monza* with the hierarchy levels above and the geonames-id, as displayed below:

term position	1	2	3	4	5
term text	bed	and	breakfast	in	Monza
term type	word	word	word	word	word
source start,end	0,3	4,7	8,17	18,20	21,26
payload					

Figure 5.3: tokens created by WhitespaceTokenizer at index time

```
[bed] [and] [breakfast] [in] [Monza]
                             [6537122]
                             [Europe]
                             [Italy]
                             [Lombardy]
                             [Milan]
```

The tokens are added in the term position of the token that is being expanded (*Monza*). As you can see in Fig. 5.4, each token added semantically has its own payload, that is displayed in the hexadecimal system. The values of the payloads (after being converted from hexadecimal to float) and the token types are the ones specified in schema.xml (see Fig. 5.1). The original terms that were expanded semantically have been marked as *processed* (by changing the term type).

Now the tokens are processed by our Ontology parser. The term *bed and breakfast* is identified, isolated as a single token, and expanded semantically:

```
[bed and breakfast] [in] [Monza]
[sleep]                  [6537122]
[accommodation]          [Europe]
                         [Italy]
                         [Lombardy]
                         [Milan]
```

Fig. 5.5 shows the final result of the index time expansion as displayed by Solr's Field Analysis function.

The two terms added semantically (*sleep* and *accommodation*) have the payload values (displayed in hexadecimal) specified in schema.xml and term types that reflect their relationship with the term being expanded. The expanded term's type is set to *processed*. At this point, all the desired expansions have been performed, and the document is ready to be added to the index.

term position	1	2	3	4	5
term text	bed	and	breakfast	in	Monza
					6537122
					Europe
					Italy
					Lombardy
					Milan
term type	word	word	word	word	processed
					geonames-id
					geonames-hierarchy-4
					geonames-hierarchy-3
					geonames-hierarchy-2
					geonames-hierarchy-1
source start,end	0,3	4,7	8,17	18,20	21,26
					21,26
					21,26
					21,26
					21,26
					21,26
payload					3dcccccd00000000
					3cd1b71700000000
					3d83126f00000000
					3e23d70a00000000
					3eccccd00000000

Figure 5.4: GeonamesParser index time expansion

term position	1	2	3
term text	bed and breakfast	in	Monza
	sleep		6537122
	accommodation		Europe
			Italy
			Lombardy
			Milan
term type	processed	word	processed
	isRelatedTo		geonames-id
	isNarrowerThan-1		geonames-hierarchy-4
			geonames-hierarchy-3
			geonames-hierarchy-2
			geonames-hierarchy-1
source start,end	0,17	18,20	21,26
	0,17		21,26
	0,17		21,26
			21,26
			21,26
			21,26
payload	3eccccd00000000		3dccccd00000000
	3eccccd00000000		3cd1b71700000000
			3d83126f00000000
			3e23d70a00000000
			3eccccd00000000

Figure 5.5: OntologyParser index time expansion

5.1.2 Query processing

To prevent unwanted behavior when performing queries on indexed documents, it is important to process the query string in a way that ensures the best results when attempting to match the contents of the index. A good start is to process the query with the same analyzer used on the documents before they are indexed. In our case, though, it is important to control the semantic expansion process to prevent unwanted matches between the query and indexed documents. If we process the query string *visiting Milan* with the same analyzer used on the indexed documents without controlling the semantic expansion, the resulting tokens:

```
[visiting] [Milan]
           [6537122]
           [Europe]
           [Italy]
           [Lombardy]
```

would be matched, for example, with all documents containing names of places in Europe (or documents containing the term *Europe*). Therefore, searching for *visiting Milan* could yield results such as *hotels in Helsinki*, *sightseeing in Paris*, and so on. For this reason, when setting in schema.xml the same analyzer used to create our index, we change the *onlyTokenize* option to true, as shown in Fig. 5.6. This way, even though the terms present in the query string won't be expanded, multi-word terms (such as *New York*) will still be identified and merged into a single term, that is the fundamental unit of search.

The query string must be inserted in the *Field value (Query)* field, as shown in Fig. 5.7.

In Fig. 5.8 we see how, after clicking on *Analyze*, WhitespaceTokenizer breaks down our query string into tokens.

Then, SemanticFilter identifies but doesn't expand *Milan* (because we blocked the expansion with our *onlyTokenize* option), as seen in Fig. 5.9. The output from the OntologyParser is identical, because in our sample ontology there aren't any terms that are present in the inputted query.

Having enabled the *highlight matches* option, Solr's Field Analysis function now highlights the matched terms, as shown in Fig. 5.10

5.1.3 Scoring

To obtain the results and scoring output we must first add documents to our index. To do so we use the *post.jar* utility located in the *example/exampledocs/*

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="geonames-parser-2"
        spaceSeparatedTypes="geonames-id geonames-hierarchy-1 geonames-hierarchy-2 geonames-hierarchy-3 geonames-hierarchy-4"
        spaceSeparatedBoosts="0.1F 0.4F 0.16F 0.064F 0.0256F"
        onlyTokenize="false"
        />
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="ontology-parser-2"
        spaceSeparatedTypes="isNarrowerThan-1 isNarrowerThan-2 isNarrowerThan-3 isRelatedTo isSynonymOf"
        spaceSeparatedBoosts="0.4F 0.16F 0.064F 0.4F 0.7F"
        onlyTokenize="false"
        />
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="geonames-parser-2"
        spaceSeparatedTypes="geonames-id geonames-hierarchy-1"
        spaceSeparatedBoosts="0.1F 0.4F"
        onlyTokenize="true"
        />
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="ontology-parser-2"
        spaceSeparatedTypes="isNarrowerThan-1"
        spaceSeparatedBoosts="0.4F"
        onlyTokenize="true"
        />
    </analyzer>
</fieldType>
```

Figure 5.6: schema.xml index and query analyzer settings for index time expansion

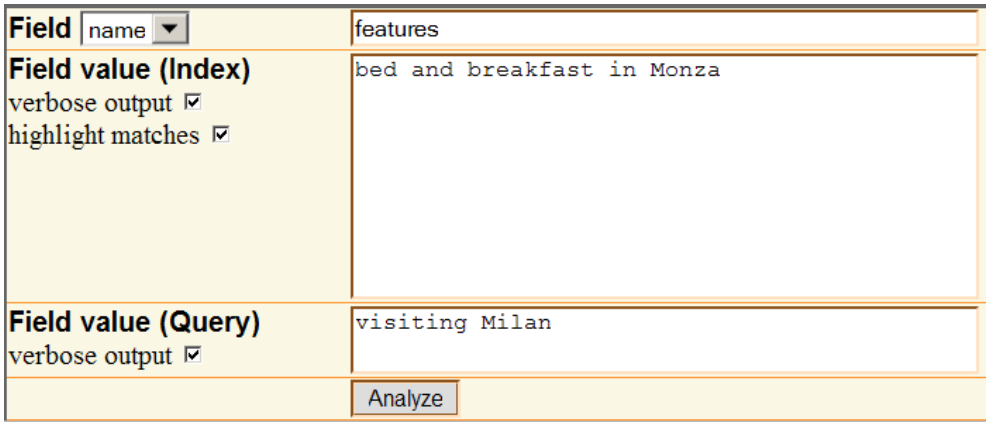

Figure 5.7: sample document input for index time expansion

term position	1	2
term text	visiting	Milan
term type	word	word
source start,end	0,8	9,14
payload		

Figure 5.8: query string broken down into tokens by WhitespaceTokenizer

term position	1	2
term text	visiting	Milan
term type	word	processed
source start,end	0,8	9,14
payload		

Figure 5.9: query string processed with semantic expansion disabled

folder of Solr's nightly builds. The two documents we used for our example were structured in XML (as displayed below) and added to Solr's index following the instructions of Solr's official tutorial[1].

```
<add>
<doc>
<field name="id">0</field>
<field name="features">bed and breakfast in Monza</field>
</doc>
<doc>
<field name="id">1</field>
<field name="features">nightlife in Milan</field>
</doc>
</add>
```

To retrieve the ranked results in XML format we submit the query *visiting Milan* with a web browser:

```
http://127.0.0.1:8080/Solr/select?q={!payloads%20f=features}visiting%20Milan&debugQuery=true
```

[1]located at http://lucene.apache.org/solr/tutorial.html

term position	1	2	3
term text	bed and breakfast	in	Monza
	sleep		6537122
	accommodation		Europe
			Italy
			Lombardy
			Milan
term type	processed	word	processed
	isRelatedTo		geonames-id
	isNarrowerThan-1		geonames-hierarchy-4
			geonames-hierarchy-3
			geonames-hierarchy-2
			geonames-hierarchy-1
source start,end	0,17	18,20	21,26
	0,17		21,26
	0,17		21,26
			21,26
			21,26
			21,26
payload	3eccccd00000000		3dccccd00000000
	3eccccd00000000		3cd1b71700000000
			3d83126f00000000
			3e23d70a00000000
			3eccccd00000000

Figure 5.10: highlighting of matches in Solr's Field Analysis function

Figure 5.11: XML query output - results

In Fig. 5.11 we can see the portion of XML output that displays the returned results. As expected, the document *bed and breakfast in Monza* is listed after *nightlife in Milan*.

In Fig. 5.12 we have the portion of XML output that displays the scoring process.

5.1.4 Query expansion

It is possible to further increase the number of relevant documents retrieved by expanding semantically the terms of the query string. As shown in Fig. 5.13, we set the *onlyTokenize* option to leave semantic expansion enabled for both the indexed document and the query string. It is important, though, to limit the hierarchy levels of our query string's semantic expansion to prevent the retrieval of irrelevant documents.

The data input of our query expansion example is displayed in Fig. 5.14.

Figure 5.12: XML query output - scoring

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="geonames-parser-2"
        spaceSeparatedTypes="geonames-id geonames-hierarchy-1 geonames-hierarchy-2 geonames-hierarchy-3 geonames-hierarchy-4"
        spaceSeparatedBoosts="0.1F 0.4F 0.16F 0.064F 0.0256F"
        onlyTokenize="false"
        />
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="ontology-parser-2"
        spaceSeparatedTypes="isNarrowerThan-1 isNarrowerThan-2 isNarrowerThan-3 isRelatedTo isSynonymOf"
        spaceSeparatedBoosts="0.4F 0.16F 0.064F 0.4F 0.7F"
        onlyTokenize="false"
        />
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="geonames-parser-2"
        spaceSeparatedTypes="geonames-id geonames-hierarchy-1"
        spaceSeparatedBoosts="0.1F 0.4F"
        onlyTokenize="false"
        />
        <filter class="org.semanticsolr.analyzer.SemanticFilterFactory"
        parser="ontology-parser-2"
        spaceSeparatedTypes="isNarrowerThan-1"
        spaceSeparatedBoosts="0.4F"
        onlyTokenize="false"
        />
    </analyzer>
</fieldType>
```

Figure 5.13: schema.xml settings for index and query expansion

We use as sample document the string *bed and breakfast in Monza*, and as query string *visiting Legnano*. Legnano is a town in the Province of Milan. Our query string will be expanded as described below:

```
[visiting] [Legnano]
           [6537118]
           [Milan]
```

In Fig. 5.15 we see how WhitespaceTokenizer breaks down our query string into tokens.

Then, SemanticFilter identifies and expands *Legnano*, as seen in Fig. 5.16. The expansion has been limited to one level. The output from the OntologyParser is identical, because in our sample ontology there aren't any terms that are present in the inputted query.

Our sample document (*bed and breakfast in Monza*) has been processed like already described in section 5.1.1, and *Milan*, the matched term, appears highlighted, as already shown in Fig. 5.10, because it has been matched with the expansion of *Legnano*.

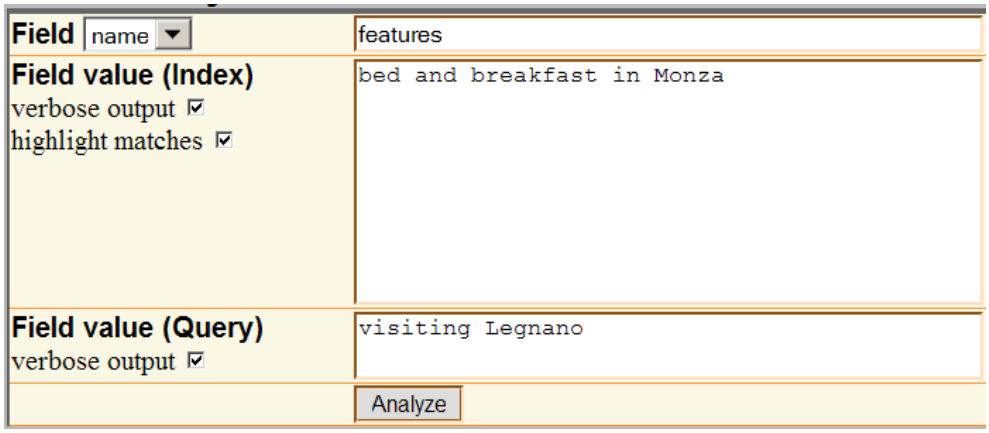

Figure 5.14: inputting the query expansion example data

term position	1	2
term text	visiting	Legnano
term type	word	word
source start,end	0,8	9,16
payload		

Figure 5.15: tokens created by WhitespaceTokenizer at query time

term position	1	2
term text	visiting	Legnano
		6537118
		Milan
term type	word	processed
		geonames-id
		geonames-hierarchy-1
source start,end	0,8	9,16
		9,16
		9,16
payload		3dcccccd00000000
		3ecccccd00000000

Figure 5.16: query expansion performed by GeonamesParser

Chapter 6

Conclusions and Future Developments

6.1 Conclusions

Stored information is near useless if users can't search it efficiently, retrieving the documents relevant to their information needs. Traditional syntactic-only search, albeit reliable and efficient, is greatly limited by the gap between the way machines work and the way we think.

Coupling the Vector Space Model with semantic expansion helps us close this gap by using knowledge representation data we can build for our custom application or obtain from numerous sources. We therefore created a working prototype of a search engine based on this model that uses a payloads-based approach to ensure control over the ranking process. The tests described in the previous chapter show how our platform enriches search results with documents that traditional search engines fail to retrieve.

Our project gives developers the tools to semantically increase the quality of search results by increasing Recall while tailoring returned results to fit the customization needs of different fields, user bases and document sets.

6.2 Future Developments

Being a working prototype, the current implementation of the project lacks some refinements that require software implementation not related to its core functionality, or that won't be necessary once the software reaches its deployment phase.

The two parsers developed for the current implementation access two very different kinds of data.

The GeoNames parser has access to vast amounts of geographical information that have not been filtered to fit the needs of custom implementations of our platform. For example, the term "or" is matched with the State of Oregon (U.S.A.) when performing a *name_ equals* search in GeoNames, because "OR" is listed among Oregon's alternateNames (a field in the XML search output). If "or" isn't excluded from the geographical processing (for example using Solr's StopFilterFactory) it may lead to false positives.

Handling of Polysemy, not implemented in the current release, is another issue that must be addressed when customizing the project for specific needs. Even some major cities share the same name, like "San José", the capital city of Costa Rica, and "San Jose", in California. A possible solution to this problem can be achieved by evaluating the context in which a specific term is placed. Analyzing other words present in the document it is possible to contextualize terms affected by Polysemy.

The Ontology parser accesses data that was created manually to perform simple tests aimed at verifying that the software behaves as designed. In the transition from the prototype to a product ready for deployment, it will be necessary to carefully analyze the case scenario the platform is being customized for, and evaluate the tradeoffs between trimming down and filtering existing data sets (eg. GeoNames, Wordnet, etc) and building them from scratch.

Storing all the data used for the semantic expansions in an SQL database would increase the performance of index and query time parsing. Using a database would enable setting different boosts for different term expansions based on specific customization needs related to the data and user base of the finished product.

The boost values employed in our indexing and query examples were chosen to reflect the decrease in relevancy as we climb up hierarchy levels. As the project nears the final stages that precede its deployment, it is also important to research the best boost values to employ.

Bibliography

[Bhagdev et al., 2008] Bhagdev, R., Chapman, S., Ciravegna, F., Lanfranchi, V., and Petrelli, D. (2008). *Hybrid Search: Effectively Combining Keywords and Semantic Searches*. Springer Berlin / Heidelberg, University of Sheffield, United Kingdom.

[Brin and Page, 1998] Brin, S. and Page, L. (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Computer Science Department, Stanford University, Stanford University, Stanford.

[Buntine, 2005] Buntine, W. (2005). *Open source search: a data mining platform*. ACM New York, University of Helsinki & Helsinki University of Technology, Finland.

[Celino et al., 2006] Celino, I., Valle, E. D., Cerizza, D., and Turati, A. (2006). *Squiggle: a Semantic Search Engine for indexing and retrieval of multimedia content*. CEFRIEL, Politecnico of Milano, Italy.

[Gehring and Lutterbeck, 2004] Gehring, R. A. and Lutterbeck, B. (2004). *A Software Engineering Approach to Libre Software*. Technical University of Berlin, Berlin, Germany.

[Giunchiglia et al., 2008] Giunchiglia, F., Kharkevich, U., and Zaihrayeu, I. (2008). *Concept Search*. University of Trento, Italy, Trento, Italy.

[Gospodnetic and Hatcher, 2005] Gospodnetic, O. and Hatcher, E. (2005). *Lucene in Action*. Manning Publications, Greenwich.

[Manning et al., 2009] Manning, C. D., Raghavan, P., and Schütze, H. (2009). *An Introduction to Information Retrieval*. Cambridge University Press, Cambridge, England.

[Raymond, 1997] Raymond, E. S. (1997). *The Cathedral and the Bazaar*. O'Reilly.

[Salton et al., 1975] Salton, G., Wong, A., and Yang, C. S. (1975). *A vector space model for automatic indexing.* ACM New York, Ithaca, NY.