A Domain-Specific Language for Internal Site Search

Master's Thesis

Elmer van Chastelet

A Domain-Specific Language for Internal Site Search

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Elmer van Chastelet
born in Dordrecht, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

A Domain-Specific Language for Internal Site Search

Author: Elmer van Chastelet
Student id: 1213539
Email: `evanchastelet@gmail.com`

Abstract

The importance of search facilities on a website grows with the size of the content being served. User expectations for internal site search are greatly influenced by global web search engines, requiring developers of web applications to go beyond basic search functionality. In this thesis, a domain-specific language (DSL) for internal site search is designed and integrated as a sublanguage of WebDSL (the base language). WebDSL is an existing DSL for web development. Through an exploration of the problem and solution space, the facets related to internal site search are explained. Furthermore, an iterative approach applied at the development of the DSL is presented. This approach is based on the use of existing base language constructs as core language. The core languages provide access to implemented search features. Linguistic abstractions are added on top of the core languages, constituting the eventual interface of the language. Evaluation by means of enriching two web applications with search features show that the DSL has substantial coverage of the internal site search domain.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, dept. SERG, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. E. Visser, dept. SERG, Faculty EEMCS, TU Delft
Daily supervisor:	Ir. D.M. Groenewegen, dept. SERG, Faculty EEMCS, TU Delft
External Committee Member:	Dr. C. Hauff, Dept. WIS at Faculty EEMCS, TU Delft
Committee Member:	Dr. Ir. F.F.J. Hermans, dept. SERG, Faculty EEMCS, TU Delft

Preface

This thesis concludes my Master's degree in Computer Science at Delft University of Technology. I would like to take this opportunity to express my gratitude to my supervisor Eelco Visser for giving me the chance to work on this project, and for his supportive feedback during the project. I enjoyed working on a deliverable (the domain-specific language) which is now actually used in production. I gained a lot of experience and knowledge in the field of language design, information retrieval and web development, which undoubtedly will help me during my future career. Special thanks go to my daily supervisor Danny Groenewegen for his continuous support and ideas during language development and for his valuable guidance and feedback during thesis writing. I also want to thank fellow graduating students who worked on WebDSL (André Vieira, Chris Melman, and Christoffer Gersen) and all fellow students from "the lab" for their help and for providing me a pleasant and fun place to work. Last but not least, big thanks go out to my family for their continuous trust and for motivating me throughout my study program.

Elmer van Chastelet
Delft, the Netherlands
August 17, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Domain-specific languages (DSLs) are programming languages tailored toward a specific domain by adopting domain concepts into the language. Compared to general purpose programming languages, they trade generality for expressiveness in a limited domain [21]. By offering notations and constructs suitable in the targeted domain, a DSL is easier to use when compared to a general purpose language. Applied properly, a DSL results in increased productivity and lower maintenance costs of the systems created with the DSL. Well known examples of DSLs include HTML for Hypertext web pages, SQL for database queries, Make for managing software build processes and LaTeXas typesetting language.

In the field of web application development, developers need to be familiar with a range of programming languages. Each of these languages is chosen or designed to deal with a specific aspect of web applications. E.g., the backend which constructs (dynamic) web pages may be implemented using PHP, Ruby, Scala, Python or Java. Data to be used for construction of the web pages may be represented in various formats (JSON, XML, POJO) and be retrieved from a database using a database query language (like SQL) or from other data sources. The constructed pages in turn consist of HTML for markup, JavaScript for interaction and Cascade Style Sheets (CSS) to define the style of the web page's elements. Also, a web-developer needs to take in consideration the web application's access control for controlling who may access which parts of the website's content. Data validation should assure that data delivered to the system is processible and checked to comply with formatting rules and value boundaries. A website's navigation structure should allow any content to be browsed and explored efficiently. Then, the design of the website determines how content is actually presented on the range of canvas sizes in use today. A web page may also be optimized for searchability by web search engines. And finally, website's search facilities should make browsable content easy to find by its visitors.

This results in an implementation platform being an amalgam of concern-centric languages and frameworks, adopting the *separation of concerns* design principle at a language and framework level. However, these do not integrate well, leading to verbose definitions across the languages/frameworks and a lack of static checks between cross-language relations. For example, a web application's data model may primarily be specified using Java classes. The Java Persistence API is used to specify how the data is to be persisted by object-relational mapping software (using Java annotations which are not checked to be consistent with the Java class definitions). The retrieval of data objects is done using SQL statements embedded as Strings in Java, which are only checked for consistency at runtime. Then, the search engine may require an XML-schema representation of the data to specify how to construct searchable documents from the data. This schema may contain SQL statements for the retrieval of data in order to synchronize the search index with the actual data. Again, these statements are not checked statically to comply with the SQL syntax and to be consistent with the data model specified in Java.

WebDSL, the language to be extended, is a DSL that tries to integrate the mixture of aspects into a language that is able to statically verify cross-concern specifications, while preserving the separation of concerns paradigm. This is achieved by having sublanguages for different concerns that share a single type system and expression language. As will be explained in Chapter 2, WebDSL currently has sublanguages for an application's data model, access control, data validation, user interface and application logic. In this master's thesis, we explain how WebDSL is extended with a range of information retrieval features to enable more feature rich web applications.

We will denote the range of features to be covered by the DSL extension as the domain of *internal site search*. It is an overarching term for techniques enabling search functionality on a single website and is sometimes referred as *local (web)site search* [6, 19]. It is different from *web*(*site*) *search* [3, 1, 15] (sometimes referred as *internet search*), where web pages are crawled and the content presented on accessible web pages become searchable. It is different from *enterprise search* [22, 13], where heterogeneous collections of (often proprietary) digital information within an organization, stored anywhere in whatever format, are crawled and arranged in a way that it can be searched effectively by a targeted audience (both within and outside an organization). Internal site search is similar to enterprise search in a sense that the information retrieval system is designed to use (meta-)data, which is not exclusively accessible though an unified resource link (URL), as is the case with web search. Also, the information retrieval system may be designed with knowledge about the structure of data and data interrelationships (in case of structured data) such that the system is able to find more relevant results and to present results and/or browsable collections more suitably to the context. However, the practice of obtaining data from a diversity of data sources represented in various formats (considered part of enterprise search [13]) is out of the scope of the internal site search domain. We assume data to be modeled conform a single structured representation. In our case, the data model of WebDSL.

1.1 Contributions

Aside from being a case study in domain-specific language engineering, the main contributions of this master's thesis are (1) an analysis of the domain of internal site search, (2) the design of an external domain-specific language for internal site search, and (3) an approach in integrating this DSL into an existing DSL. The developed DSL covers semantics for indexing configuration, query formulation and refinement, and the retrieval and presentation of searched data. The set of language features is extended iteratively in such a way that each intermediate state of the extended base language supports the use of new language features directly. By means of a WebDSL application developed in parallel with the language extension, the state of the DSL extension is evaluated to drive the future iterations in the development of the DSL extension. In later phases, the applicability of the developed DSL is evaluated by means of integrating search features in production web applications: a web-based source code search engine is developed and an existing digital library application is extended to adopt new search features.

1.2 Thesis Structure

Following this introduction, we give a brief introduction of the base language to be extended with a sublanguage for internal site search: WebDSL (Chapter 2). We then discuss the typical features related to internal site search and difficulties encountered setting up these facilities (Chapter 3). The solution domain is explored in Chapter 4 by means of a comparison on the relevant search capabilities of a selected set of popular open source search engine libraries. In Chapter 5, we explain DSL development in general and previous WebDSL development, followed by a detailed overview of the development process applied for extending WebDSL with a DSL for search. The syntax of the actual language is presented in Chapter 6. Then, in Chapter 7, we put the language into practice by means of 2 case studies and a brief comparison with an ordinary, real world, Java application. The first case study comprises the creation of a web application for searching source code repositories. In the second one, we take an existing digital library created with WebDSL and update it with newly added search features. A brief comparison then shows the boilerplate code normally encountered when implementing search in a Java web application. Other examples of search DSL's and a discussion of future work is given in Chapter 8. Finally, conclusions are drawn in Chapter 9.

Chapter 2

An Introduction to WebDSL

WebDSL is a domain-specific language (DSL) allowing its users to develop dynamic web applications with rich data models. Typical for DSLs, WebDSL abstracts from low level boilerplate code which one usually encounters when using general purpose languages like Java. Web developers would normally use a combination of languages and frameworks to deal with different aspects of web programming, e.g. (X)HTML and CSS for presentation, SQL for data persistence, Javascript for user interaction and a language to generate web pages dynamically like PHP or Java. The WebDSL language handles this complexity and consists of a collection of sublanguages for different aspects that are used interchangeably within one source file. This has the additional advantage that code can be statically checked for errors at edit- and compile time where, in ordinary web development, checking at run-time is often the only way to find errors.

2.1 Data Model Language

The core of a data intensive web application is its data model. WebDSL uses the Java Persistence API (JPA) to enable storage of so-called *entities* into a database. JPA is a standard for *object-relational mapping* (ORM). Entities are instances of a Java class that is annotated with the @Entity annotation. ORM frameworks, like JBoss Hibernate, use these annotations to store Java objects into a database, making Java objects persistent. At the moment of writing, Hibernate version 3.6.2 is used in WebDSL. JPA prescribes that each entity should have a unique *identifier*. The @Id annotation can be added to a field indicating that this value should act as identifier. Additional annotations are needed on properties that refer to other entities, or collections of them, the cardinality between the entity owning the property, and the type of property should be specified using cardinality annotations. While the use of JPA is conceptually simple, it leads to redundant code for getters/setters and annotations might become quite complex.

The data modeling language of WebDSL relieves the developer from this complexity by having an expressive entity language that entails the essential aspects of JPA. Entities can be declared by defining an entity name and a list of properties. Each property has a name and type, and optionally one or more annotations. Current WebDSL built-in types include numeric types like Integer, Long and Float and String-based types like: String, WikiText, Secret, URL and Email. Properties might also be a reference to another Entity type, or even a collection of these. Listing 2.1 and 2.2 show examples of entity definitions.

```
entity Publication {
  title        :: String (name)
  authors      -> Set<Author>
  description  :: WikiText
  creationDate :: Date
}
```

Listing 2.1: WebDSL entity declaration

The entity keyword is followed by the name of the entity and a body between curly braces. Within this body, the properties, functions and validation rules can be declared. In the example, we define the Publication entity with 4 properties. Properties may be of simple type indicated by `::`, reference type for collections or other entities indicated by `->`, or composition type indicated by `<>`. The latter one triggers a cascade deletion when the entity owning this property is deleted. The property `title` has the type String and has a *property annotation* 'name' . This indicates that the value of the `title` property serves as textual representation of a `Publication` entity. This name is shown when an entity is used in an output element. Another special property is a derived property, like `nofPubs` in the entity definition for `Author` (Listing 2.2). The value of this property is not explicitly persisted in the database, but evaluated at runtime.

```
entity Author {
  name          :: String (id, name)
  publications  -> Set<Publication> (inverse = Publication.authors)
  nofPubs       :: Int := publications.length

  function showInfo() : String {
    return name + " has contributions in " + nofPubs + " publications" ;
  }
}
```

Listing 2.2: WebDSL entity declaration with function

Functions in WebDSL can be declared globally or bound to an entity type. In the latter form, entity properties of the instance on which the function is invoked are accessible like class fields are accessible in Java methods. The `showInfo()` function in Listing 2.2 is such an example.

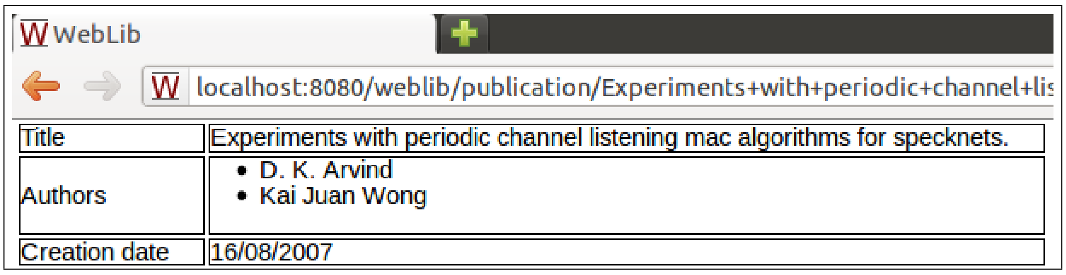
Additionally, WebDSL supports entity inheritance and entity definitions can be extended at different places in the source code, allowing partial entity definitions. Last but not least, validation rules can be added to an entity definition. These rules set constraints on the property values, for example to guarantee value well-formedness. A validation rule consists of an (e, s)-pair, where e is a boolean expression that checks a condition and s is a String that describes the error message. In case of an input form, the error message will be displayed when data entered by the user does not meet the constraints (i.e. the boolean expression e evaluates to *false*).

2.2 Pages, Templates and Actions

Pages and user interaction are expressed using page, template, and action definitions. *Page* definitions are used to describe the web pages to be viewed by the users. A page may display data (entities, entity properties) and input elements for adding, removing or updating data. A page definition consists of a name, a list of parameters to be passed and a page body. Within the page body, additional variables can be declared. Data is decorated and presented on pages using built-in or user-defined templates. Some examples commonly used built-in templates include forms, tables, lists, sections, headings, links and buttons. WebDSL *templates* can best be described as reusable page elements that can be invoked from within a page or other template definition. Similar to pages, a template definition consists of a name and list of formal parameters. By preceding the template signature with the `ajax` keyword, it can be used to replace *parts* of a currently viewed web page using asynchronous JavaScript and XML. The explicit distinction between normal and ajax templates is currently required to handle access control to these templates properly. Listing 2.3 shows a template for viewing information of a `Publication` entity in a table. The corresponding browser view is shown in Figure 2.4.

```
define showInfo(Publication pub) {
  table{
    row{
      column{ "Title" }
      column{ output(pub.title) }
    }
    row{
      column{ "Authors" }
      column{ output(pub.authors) }
    }
    row{
      column{ "Creation date" }
      column{ output(pub.creationDate) }
    }
  }
}
```

Listing 2.3: WebDSL Template definition 'showInfo'


WebLib
localhost:8080/weblib/publication/Experiments+with+periodic+channel+lis

Title	Experiments with periodic channel listening mac algorithms for specknets.
Authors	• D. K. Arvind • Kai Juan Wong
Creation date	16/08/2007

Figure 2.4: Browser displaying 'showInfo'-template (Listing 2.3)

While pages and templates use a declarative language to describe how content is shown, *action code* uses an imperative sub language to describe modifications to the content. Within an *action*, entities can be added, deleted or updated with new information. Also, it may redirect to another page or replace one or more parts of the current page using ajax. Actions belong to a form, in which the user can enter data or click elements. Typically the user enters the data and clicks on a link or button. The action code is attached to this link or button, and gets executed on the server after clicking. Changes are propagated when the entered values are correct (validation succeeds), or an error message is displayed otherwise.

2.3 Sub-languages for Various Concerns

The WebDSL language is designed following an inductive approach, where a base language is extended with other sub-languages for different concerns over time. The core of WebDSL consists of the data modeling language (entity definitions), user interface language (pages/templates) and the action language to model the web application's logic. Later, WebDSL was extended with a sub-language for modeling access control to web pages and data elements; a data validation language for guaranteeing data consistency; and a workflow language to model business workflows in a WebDSL application. In this thesis we study a new sub-language for internal site search.

2.4 Limited Search Support

At the start of this master's project, WebDSL basically offered two ways for the retrieval of data (entity instances). (1) The data store can be queried using a database query language. Additionally, some shortcut functions are generated for each entity that wrap around commonly used queries such as `findEntityByProperty` and `findEntityByPropertyLike`. (2) A search engine was later integrated to perform basic search on textual entity properties.

2.4.1 Using a Structured Query Language

WebDSL incorporates a database query language which is a subset of the Hibernate Query Language (HQL) featured by the Hibernate framework. It facilitates querying the underlying database to find the desired entity instances. This will work for simple cases where a property is known to exactly match some value. However, if one or multiple terms may match anywhere in the text of a property (i.e. perform *full-text search* on a single property), queries become more complex. The example in Listing 2.5 shows a statement where variable `result` is assigned a list of `Publications` that satisfy the constraint to contain 'retrieval' or 'Web' somewhere in its description property value.

```
result := from Publication as p
          where (p.description like '%retrieval%'
              or p.description like '%Web%');
```

Listing 2.5: Using HQL to perform textual search

Depending on the needs, this approach might work as expected. However, what if case sensitivity is important? Will the lowercased term 'web' match the capitalized form? And what if that's not the desired behavior? The problem is that most default SQL collations are set to be case insensitive, so 'web' will be matched. To deviate from this, SQL features the `COLLATE` clause in which one can deviate from the column collation. It is unfortunately not possible to deviate from the collation settings using HQL.

Another issue using this approach is the order in which the results are retrieved. Will entities with more query matches appear higher in this list, or are other heuristics in effect for relevance sorting? Using HQL, relevance sorting can be achieved using the `ORDER BY` clause in a way that it counts the number of matching terms. Listing 2.6 shows the extended version of the previous example. Since `CASE` clauses are not (yet) included in WebDSL's subset of HQL, the example would not parse. The example is solely used to give you an impression of the kind of queries that arise when using SQL/HQL for information retrieval.

```
result := from Publication as p
      where description like '%retrieval%'
        and description like '%Web%'
      order by
        (case when description like '%retrieval%' then 1 else 0 end
       + case when description like '%Web%' then 1 else 0 end)
      desc ;
```

Listing 2.6: Using HQL for textual search, ranking results with more hits higher

Using a database query language to perform searches will quickly result in large and complex queries. Furthermore, it supports limited types of queries and built-in functionality to order the results by relevance are missing. It is therefore not the desired approach when integrating full text search in a web application.

2.4.2 Using Basic Search Functions

For this reason, WebDSL was later extended with basic full text search capabilities. A developer using WebDSL specifies the entities that must become *searchable* by annotating the properties of an entity that should be added to the search index. Listing 2.7 shows an example where the `Publication` entity is made searchable with the values of `title` and `description` to be added to the search index.

```
entity Publication {
  title        :: String (name, searchable)
  authors      -> Set<Author>
  description  :: WikiText (searchable)
  creationDate :: Date
}
```

Listing 2.7: WebDSL declaration of a searchable Entity

If at least one property of an entity is marked searchable, a set of search functions are generated. These functions return a list of entities matching a query. An offset and number of entities to be returned can be passed as argument to enable pagination of results. The returned list is ordered by relevance using the search backend's default ranking, where the backend is Apache Lucene in this case.

```
function searchPublication(query : String) : List<Publication>
function searchPublication(query : String, limit : Int) : List<Publication>
function searchPublication(query : String, limit : Int, offset : Int) : List<Publication>
```

Search, at that moment, only supported String type properties. Properties' content was indexed using one and the same *analyzer*, that splits the content into tokens based on whitespaces and interpunction. Tokens were lowercased and English *stop words* (common words like: the, to, be, is , ...) were ignored. At query time, all searchable properties of an entity were used to find matches. Integration of basic search functionality was powered by Hibernate Search from JBoss, offering seamless integration with Hibernate Core which WebDSL uses for ORM mapping. Hibernate Search uses Apache Lucene for indexing and searching.

Chapter 3

Problem Domain: Internal Site Search

Internal site search is the practice of enabling the content served by a web application to be indexed, searched and displayed. Different from *web search*, where users are seeking over the world wide web or subset of it, internal site search is concerned with search facilities within a single website. Although these 2 concepts are different, user expectations of internal site search are greatly influenced by the standards set by web search engines like Google. One example is the ability of web search engines to supply spell corrections and auto complete suggestions. Search terms are suggested when someone makes a type or spelling mistake, and while entering the query. Web search engines also own a great amount of usage data which they can use to improve their search facilities. In this chapter, we take a look at typical facilities related to internal site search and the issues one encounters setting up such facilities.

3.1 Requirements for Basic Search Capabilities

To meet users' expectations in a web application serving content, having a basic search facility is a minimum requirement. A simple search user interface encompasses at least an input form where visitors may enter a search query and hit search, and an area where search results are presented. Depending on the type and amount of content, a basic search facility might suffice in satisfying the targeted audience. However, apart from questions like how to present search results, more advanced search features are often preferred in order to make a website its content accessible easily. In this section, we will discuss basic tasks and concepts that relate to integrating site search to a web application. Then, more advanced search features are discussed in Section 3.2.

3.1.1 External Web Search Service

First of all, a web developer needs to decide which parts of the website's content to become searchable. Let's imagine a developer that created his own blogging website, build from scratch, i.e. no pre-built solution or online blog service is used. Regarding search functionality, he wants blog posts to become searchable. In this case, each blog post might be considered a searchable unit. A different option is to treat the *pages* (accessible through an URL) that serve the blog posts *and* comments to become the searchable units. In that case a simple solution is to go for an *external search service*. External search services will index the content by regularly crawling the website (i.e. start at some URL and follow any link from the same domain until no unvisited link is left, indexing each page). Most external search services can be embedded on a website using a relatively small code snippet and may also offer features beyond simple keyword search. For instance, Google Site Search [1] has the ability to view results that were updated or added within a specific time frame. However, the downside of using external site search services is their limited access to the data and their lack of knowledge and therefore utilization of the website's data model and business logic. If, for example, some blog posts are exclusively accessible by logged-in members, these won't get crawled and indexed. (Except if the crawler can be authenticated as special user, but then the hidden content may appear in results for searches performed by non-members, which is not desirable). Other problems arise when searches involve data-model relations. An external site search service cannot deal with constraints like: 'find blog posts posted by user 'Tim', where user 'Jessica' left a comment. An external search service would only be able to search for web pages in which both the terms 'Tim' and 'Jessica' appear, no matter in which context these terms appear. In general, if required search functionality involves dealing with data model relations, search has to be integrated into the web application.

3.1.2 Building a Search Index

Luckily there are a variety of libraries/engines that can be used to drive site search. A selection of these will be discussed in chapter 4. These all follow the same principle to use an inverted index where *documents* represent searchable units (like a blog post), and terms are linked to a list of documents that match that particular term. The index may contain additional information, like term frequencies (both within a single document and within all documents), positional information of terms, and other data which might for example be used by the ranking engine of a search framework. Terms in a search index are, in general, textual tokens extracted from the original data that is served by a system. Searchable data must therefore become available in textual form, if this is not yet the case. This means that a system serving articles in binary format (like pdf files or text files packed in a container format) must first preprocess this data by transforming it into a textual representation. After this optional process, the data is ready to be processed for indexing. Figure 3.1 shows the typical workflow of processing data for index addition.

Not all browsable data will be candidate for index addition. After deciding which data entities to represent by documents (i.e. to become searchable), the next step is to make a selection of the data and metadata that will become part of a document's content. This depends on the type of constraints a search facility should support. Looking at the blogging website example, a blog post's [title] and [content] should be searchable. Additionally, there

[1] `http://google.com/sitesearch`

might be [tags] assigned to blogs that can be searched. One might also like to search blogs by the [name of the poster], or visitors might click a username to view blogs where this user left a comment (read: search blogs by a [commenter's name]). And finally, the presentation of search results can also be adapted to be viewed in order of [post-date] and even be filtered for a particular time span (i.e. range of post-dates). The mentioned [data properties] are a possible selection to be used as document's content for the blog search index. The collection of selected data properties should be mapped to *search fields* in each document. The notion of search fields within documents makes it possible to perform searches with constraints related to a subset of these fields. Also, *tokenization* and *normalization* (often referred as *analysis*) can be different for each search field. Text analysis is of great importance when it comes to relevance and recall of results of a search engine.

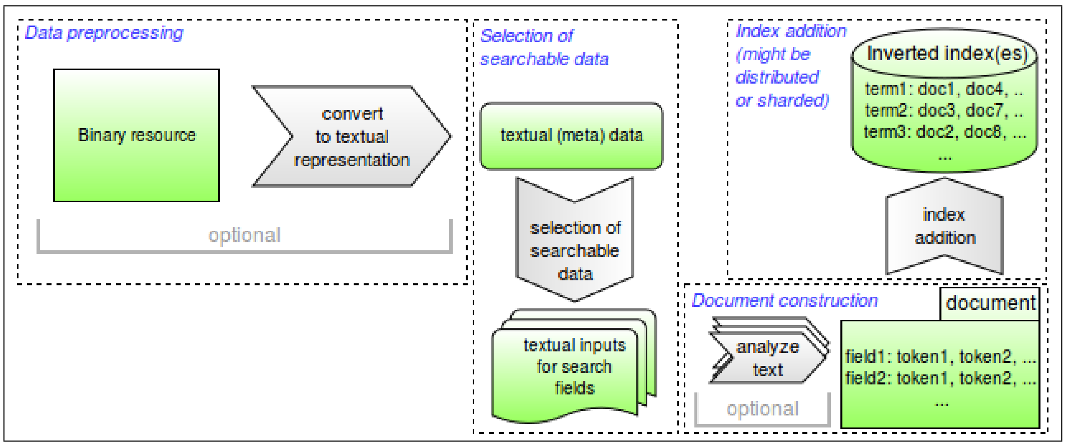

Data preprocessing
Binary resource
convert to textual representation
optional
Selection of searchable data
textual (meta) data
selection of searchable data
textual inputs for search fields
Index addition (might be distributed or sharded)
Inverted index(es)
term1: doc1, doc4, ..
term2: doc3, doc7, ..
term3: doc2, doc8, ...
...
index addition
Document construction
document
analyze text
optional
field1: token1, token2, ...
field2: token1, token2, ...
...

Figure 3.1: Search index creation: the process of transforming browsable data into searchable documents to be added to a search index

3.1.3 Tokenization and normalization

As can be observed in Figure 3.1, textual data coming out of the searchable data selection can be *analyzed* before document addition. Text analysis encompasses tokenization, in which textual data is split into *tokens* (words), often followed by some sort of normalization of the tokens. These tokens will become members of a document's search field and are eventually represented by terms in the search index.

In general, token normalization is applied to associate multiple tokens to the same term. When applied appropriately, this may improve recall (i.e. returning more potentially relevant documents) and precision (i.e. a greater portion of the returned documents are relevant) of the search system. For example, you don't want a capitalized token 'Retrieval' to be missed when a user searches for the non-capitalized form 'retrieval'. Similarly, more (potentially) relevant document will be matched if different forms of the words are matched against each other, e.g. searching for 'repository' should match documents with 'repository' or 'repositories' in it. Common transformations during analysis include: lowercasing, removal of punctuation, splitting compound words, and stemming. *Stemming* removes commoner morphological and inflectional endings from words such that all inflections of a word will be transformed to the same root term. For example, the words ‘carry’, ‘carried’, ‘carrying’ and ‘carries’ all reduce to the same root form ‘carri’.

Another transformation that is often applied during analysis prior to index addition is stop word removal. Stop words are words that are used quite often, but not add any meaning in the context for which they are used. Stop word removal will increase search performance, because a search engine can skip processing the large document-sets linked to these words. For English text, a stop word set is likely to include the words *the, is, to, be, at, a, and (,) an*. However, stop word removal is not always desirable: in a music library, the title ‘let it be’ should be searchable. Another form of token removal which is not uncommon is stripping off metadata tags such as HTML-tags. This is useful in case the data itself contains such markup tags and these are not removed by any preprocessing prior to the analysis phase. Removal of tokens that don’t contribute to the content will hinder such tokens to get matched at query time, which would distort the relevance of documents during retrieval.

To get a better picture of what tokenization and token filtering/normalization does, take a look at the following example:

<emph>On a good website, search functionality is reachable from within any web page.</emph>

In this example, the data to become searchable contains HTML-tags which should not get indexed. These meta tags should therefore be removed first. The next step in analysis might then be to apply a simple tokenizer that splits a stream of text on punctuation and whitespaces, followed by lowercasing the tokens. The tokens may then be filtered to remove stopwords and apply stemming in the end. Applying this analysis, the example will be transformed into the following tokens:

```
[good, websit, search, function, reachabl, from, within, ani, web, page]
```

As a general rule, the same analysis as applied at indexing time is used at query time. This way, tokens that appear in a searchable resource will never mismatch against the same token at query time. However, there are some exceptions to this rule, for example when applying *query expansion*. Query expansion is a useful trick to increase recall by expanding tokens from user queries (or from searchable data at indexing time) with multiple synonyms and/or derivations if available. For example, a search engine set up to include term expansions for ’computer’ may try to match this term also against ’notebook’, ’pc’ and ’laptop’ in the index. Expanding terms is mostly done on queries, because a change in the expansion algorithm won’t require the search index to be rebuilt and index size won’t increase by adding synonyms or word derivations. For this reason, it is generally known as query expansion, and implies a different analysis at query time compared to index-time analysis. Again, the applicability of query expansion depends on the context. Words semantics may differ for different contexts, and therefore expanding terms with synonyms may not always be a good idea. Query expansion may also be limited to only expand specific terms, such as expanding names of chemicals (with their acronyms and other forms used in literature) in a digital library for chemists. While this is a simple example, query expansion can quickly become complex when terms are to be expanded dynamically. That is, by taking into account the context of terms (surrounding words and/or other metadata) in order to construct a list of term expansions that are meaningful in current context.

3.1.4 Searching the Data

Shneiderman et al. [28] formalized the search for information into 4 phases: formulation, action, review of results and refinement. *Formulation* is regarded as complex task, done mostly by the user in search for some piece of information. It includes the selection of information source (where to search), selection of which parts to search for (which fields of documents), the text to use for searching, and what variants of texts to accept. *Action* relates to initiation of actual searching, often done by clicking a search button, but also sometimes implicitely while typing. *Review of results* is tightly bound to the presentation of results. This includes the number of results displayed at once, the order in which the results appear and the way how these results are shown. This should all help to give the information seeking user a clear overview in a way that he can quickly decide whether a result is relevant or not. This may lead to a *refinement* of his search, by changing constraints specified during the formulation, or by narrowing down the result set by adding additional constraints. We will now discuss the technical aspects that are related to the user interface of an application's search facility.

A simple search page constitutes at least an input form where the user can enter his query and a search button that will load the search results. When the user hits the search button, a request including the query will be sent to the web server. The server will then use the search engine to search for results. Most search engines/libraries are able to handle parsing and analysis of a user query. This will transform a user provided query String into a data structure processible by the index searcher component.

When the query is processed, most search engines (servers)/libraries will return a *ranked* list of documents. Often, these documents first need to be translated back to the actual data objects from the data store (the *primary information objects*) which the document represents in order to present the results on a web page. Therefore, indexed documents will own a field with a unique identifier that is associated with the primary information object. So after retrieval of the document list from the search engine, the actual data can be retrieved using the identifier. The actual data can then be used to present them as search results. Depending on the search engine, documents may also hold the original (unanalyzed) textual data for a selection of search fields. In that case, retrieving the primary data objects may not be necessary. A basic view of the typical process of serving a search result page after the user clicks a search button is shown in 3.2.

3.1.5 Presenting Search Results

The presentation of search results is an important concern regarding the effectiveness and usability of an application's information retrieval functionality. In order to help end users of an application's search functionality, search results are often represented differently than the primary information object itself, by limiting the amount of information that is displayed.

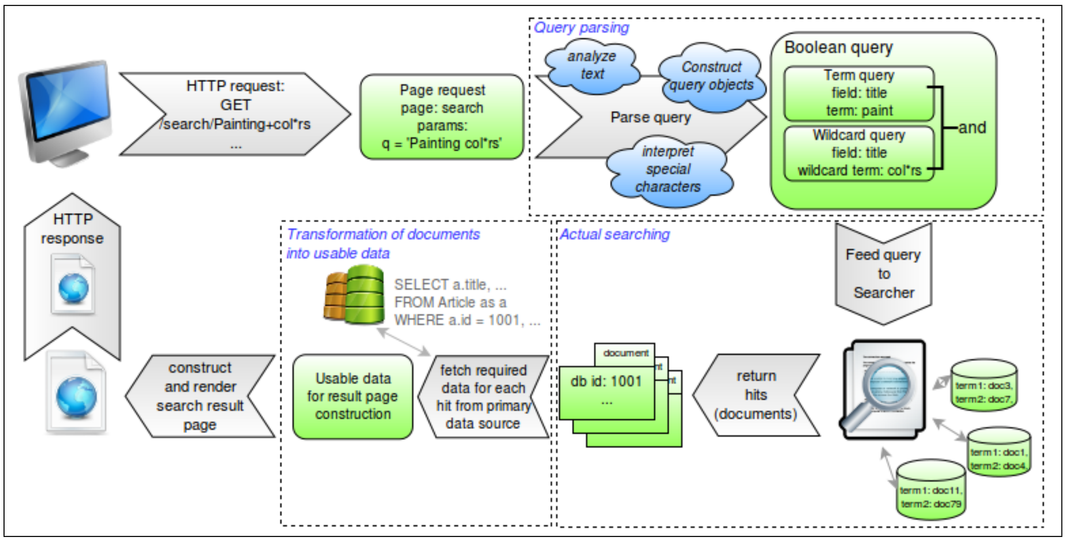


Figure 3.2: Typical flow of processing a search page request

Simply viewing only the top N results will already cut the information shown on a single page. This requires a page index to allow navigation between result pages. To create such an index, the size of the results need to be known (without retrieving the actual documents), and the constraints set in the current *search session* must be memorized somewhere (probably as request parameters in links or buttons) in order to view another result page for the same constraints.

It's also a good idea to limit the information shown for each result. This is often done by presenting previews and/or overviews. These previews/overviews serve as *surrogates* of the primary information objects. When used accordingly, it helps the end user to quickly discriminate interesting results from non-interesting ones [11]. A simple start is to only show the title and maybe some descriptive text, being an example of a *static* fragment (or summary) independent of the query context. Better would be to present dynamically constructed fragments that relate to the user's query terms. These are called *query-biased* fragments (or summaries). Tombros and Sanderson [31] showed that using query-biased summaries significantly decreases the number of times users need to browse to the full text in order to judge a result to be relevant or not. Moreover, their subjects were able to better judge the relevance of results.

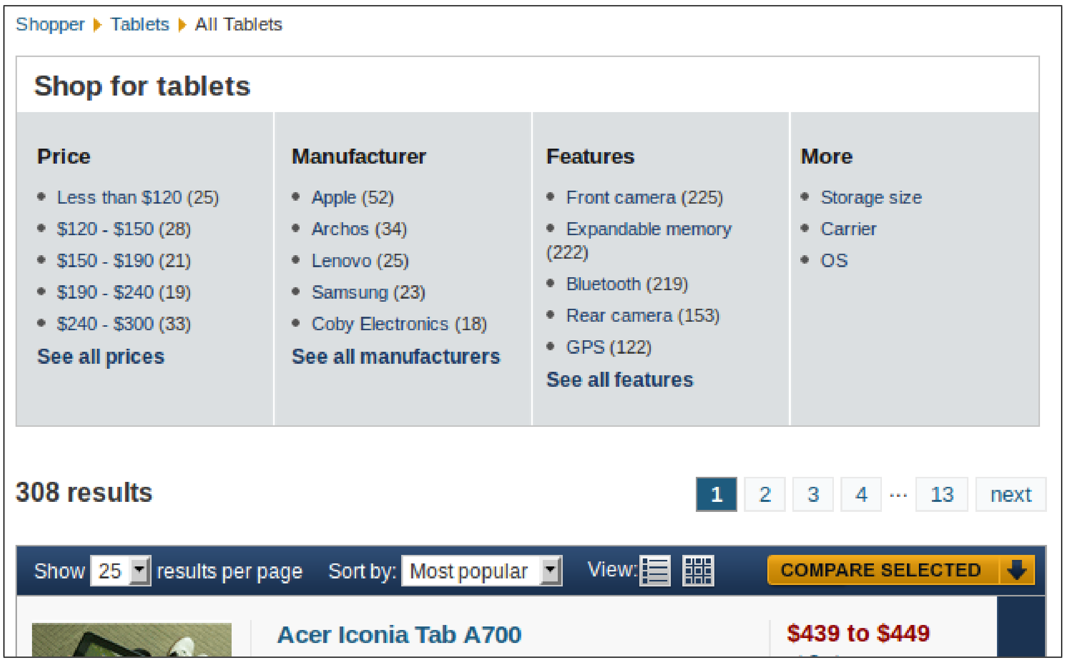
Another commonly applied technique for improving user satisfaction during the review of results is the highlighting of query terms [14]. By emphasizing or coloring query terms in the presented surrogate documents, users can quickly observe why results are considered relevant by the application.

The time spent by a user on reviewing the results can be diminished further if the order of results reflects the relevance in the currently browsed context. Default relevance ranking applied by search engines is not always desirable. Sometimes it is better to base the order of results on (meta) data properties of a document. When browsing for products in a web shop for example, one might prefer ordering the results by ascending price, or best user rating.

Or when searching for new releases in literature, one would like to view results ordered by date. Allowing control by the user over the order of results, or just deviating from the default ranking applied by a search engine can greatly reduce the time span of reviewing results.

3.2 More Advanced Search User Interfaces

There are many ways to improve user satisfaction with respect to a website's navigation and its search user interface. We already discussed some techniques such as hit highlighting and selection of interesting fragments (biased to a search query) on a search result page. The implementation of such features mostly comes with the search framework in use, because it requires interaction with parts of the search engine in order to determine which fragments are interesting and which terms need to be highlighted. Just using regular expressions won't work in most cases, because query terms are first analyzed and will probably match more than the exact term. More useful features can be implemented by reusing the information available in search indexes. An example of this that is increasingly used is faceted search.


Shopper ▸ Tablets ▸ All Tablets

Shop for tablets

Price	Manufacturer	Features	More
Less than $120 (25)	Apple (52)	Front camera (225)	Storage size
$120 - $150 (28)	Archos (34)	Expandable memory (222)	Carrier
$150 - $190 (21)	Lenovo (25)	Bluetooth (219)	OS
$190 - $240 (19)	Samsung (23)	Rear camera (153)	
$240 - $300 (33)	Coby Electronics (18)	GPS (122)	
See all prices	See all manufacturers	See all features	

308 results

1 2 3 4 ··· 13 next

Show 25 results per page Sort by: Most popular View: COMPARE SELECTED

Acer Iconia Tab A700 $439 to $449

Figure 3.3: Example of faceted search on CNET's shopper.com

3.2.1 Faceted Search

Faceting is a dynamic way of presenting navigation and search filter options when browsing a collection of items. Figure 3.3 shows the e-commerce web page CNet shopper when browsing for the category tablets. The lists of navigational subsets offer the user an overview of the available sets of products by some criteria. In this example, we see faceting on price, manufacturer and features and links to other criteria (like storage size). Compacting the set of information presented to the user is also in effect for faceting. For the price facet, ranges are limited up to 300 USD, and for the manufacturer and feature facets, the top 5 values within that constraint are shown. If a user wants to see more values within a facet criteria, the link at the bottom of the list can be clicked.

The context where facets are presented is not limited to the search page of a (web) application. Actually, the CNet shopper example demonstrates the use of faceted search as an organized way of exploring a website's content. Still, faceting and search go hand in hand. Facet values are mostly retrieved from a search index, and when a user selects (i.e. clicks) a facet, the search engine is probably the resource used to filter the content that matches the selected facet. This way, faceting constraints can easily be applied in the context of other search constraints.

Prerequisites regarding the search index

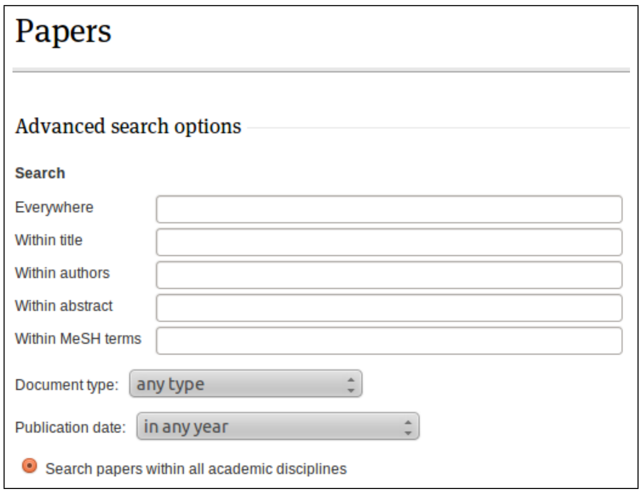
The use of search indexes for faceting requires some preparation regarding the data that is stored in the indexes. Most search solutions use inverted indexes where terms are linked to a list of documents that contain such terms. When used for faceting, these terms will act as constraint values which get presented to the users as facet values. Facets should basically be seen as pair of constraint value and hitcount, where the hitcount is the number of documents that will match when selecting this facet, i.e. filtering on the constraint value. In order to retrieve facets, each facet criteria (like 'manufacturer') needs to be represented by a field (say 'manufacturerfld') in targeted documents (i.e. documents that represents products and match the criteria to be part of the category 'tablet' in the above example). While, or after, performing a search request, documents that are added to the result set are scanned on the specified facet fields to constitute a list of unique terms with their hit counts which is the list of facets. However, fields used for searching are mostly configured to hold normalized tokens. The manufacturer name 'Coby Electronics' may be normalized and split into the tokens [coby, electronic], making it unusable for faceting. In general: if the terms in existing search fields are not the desired terms for faceting, it will be required to introduce additional search fields for faceting. In case of manufacturer names, indexing them without performing any tokenization or normalization during analysis could provide the terms useful for faceting. For this to work, one must assure that there is only a single typographic form for each entity or value to be used for faceting. When this is not the case (e.g. 'Samsung Electronics', 'Samsung' and 'samsung' represent the same entity), some preprocessing is required to assure that these values share the same unique term in the search index. If not done properly, navigation through faceting will become unusable because of unpredictable and incomplete results on facet selections.

The same negative effect can be experienced when data is missing for the search fields used for faceting. In the CNet shopper example, filtering on a the feature facet will only give complete results when all features are set for all tablet (or other products) appropriately.

Upon facet selection

The purpose of faceting is to present an organized overview of a collection and to provide navigation options for narrowing down a browsable collection. A collection that is currently browsed can be seen as a context in which faceting is applied. This context can be a simple constraint like the collection of all `Products` (e.g. class=Product), or a collection of results for a more advanced search query. Whenever a facet is selected (or removed from selection), the context needs to be updated with additional constraints (or removed constraints) for which a facet selection stands. In most cases, the constraints that constitute the context are translated to a search query. Whenever there is an update in the selection of facets, the constraints encoded as search query need to be updated, and the web application elements that relate to the context need to be updated. These include: 1) any element that may control the context, for example the panes that show selectable facets. These must be updated to only show available facets in the updated context and depends on how facets are combined (OR, AND, NOT). The presented hit counts for each facet should reflect the number of results when selecting this facet for the updated context; 2) the elements that represent the currently selected facets (might be within the list of presented facets itself); 3) the result pane must be updated with the new result collection.

3.2.2 Advanced full text search



Papers

Advanced search options

Search

Everywhere	
Within title	
Within authors	
Within abstract	
Within MeSH terms	
Document type:	any type
Publication date:	in any year

● Search papers within all academic disciplines

Figure 3.4: Example of advanced search form on Mendeley.com

Search functionality available on a website should by default be easy to use by any visitor. Often, websites offer a simple search input form with a search button, but allow more advanced users to use additional search features through an advanced search form (Figure 3.4). Advanced search forms often allow users to define which keywords must all match, which keywords are optional and which keywords should be excluded. Additionally, other types of queries may be allowed and different queries may be combined. The following enumeration discusses a variety of popular query types used on the web.

Text and Term queries are the most basic type of queries. Term queries are basically a tuple of 1 or more search fields and a term. The term is matched against the search index as-is, thus without any normalization. For simple keyword searches, the query words entered by the user must first be tokenized and normalized, as is done with data at indexing time. Then, each token is transformed into a term query. In case of multiple term queries, the queries are wrapped into a boolean query. It depends on the application context how the queries are combined (*or/and/not*). A query taking as input the text as typed by a user, followed by tokenization, normalization and construction of a single (boolean) query is often denoted as *text query*.

Boolean queries allow to combine multiple, possibly mixed types of queries. They consist of one or more *clauses*. Each clause is accompanied with an occurrence specification, describing the behavior of the clause with respect to matching: for a document to match, it *should*, *must* or *must not* match the specified clause. These occurrence operators are analogue to the logical *or*, *and* and *not* operators. Often, (advanced) search user interfaces allow users to specify which terms must all appear, which terms are optional and which terms may not appear. This will be translated into a boolean query when processed. In general boolean queries enable one to specify queries with multiple constraints with a variety of conditions.

Range queries enable one to find documents matching a specific range of values. These queries are often used in the context of dates, prices and ratings.

Wildcard queries add flexibility to search. Some search systems allow users to specify a query with *wildcard* terms. Wildcard characters (like '*' and '?') can be used when there are different written forms of a term (like 'color' and 'colour') or just to workaround possibly misspelled words ('advi?e' will match both 'advice' and 'advise'). Similarly it can be used to match multiple inflections of a word ('limit*' will match 'limiting', 'limits', ...).

Fuzzy queries are close to wild card queries. A term in the index matches a fuzzy query if the query and index term are *similar* to some extent. String metrics like the *Levenshtein Distance* [20], in which the number of single character edits needed to transform one word into another are counted, are used to determine to which extent terms are similar. A minimum value of similarity will control whether terms do match or not. Fuzzy and wildcard queries differ from each other in their kind of similarity restrictions. Wildcard queries state that the non-wildcard characters must be equal and in order and the wildcard characters are bound to the location of adjacent non-wildcard characters, where fuzzy queries only require terms to be similar to some extent based on a textual similarity algorithm.

Proximity queries are used to put a positional restriction on multi-term queries. Multiple terms should appear within a specified distance (number of terms) from each other. Additionally a constraint can be set on the order of terms. A prerequisite for proximity queries is that the search index maintains positional information about the tokens.

Geospatial queries adopt the notion of location. For users of web applications, location information can be of essential value, and the level of relevance/interest of served content is possibly bound to spatial constraints (e.g. on auction websites or a web application sharing points of interests like restaurants). Geospatial queries make it possible to restrict the result set to some area (mostly a geometrical figure like circle or square) around a centric location.

Filtering Recurring Constraints

Sometimes constraints are to be applied repeatedly, such as matching documents in a particular language. This can be modeled by adding an additional must-clause next to the user query, e.g. '`content:userQuery AND language:EN`'. This will indeed remove non-English results. However, when the sophisticated relevance scoring performed by search engines treat the additional constraint as ordinary user query, it may contribute to the scores differently for each document of the result set. This is not the desired behavior, because the additional constraint should only filter the result set, not contributing to the relevance score of the individual documents. For this reason and for increased performance, most search engines offer the possibility to specify recurring constraints using *filters*. These do not disrupt the result ranking and their result sets can often be cached, relieving the search engine from recalculating the result sets every time the same filter is applied.

3.2.3 Presenting (Navigation) Context

An important aspect of a good user interface is to minimize the (short term) memorization effort needed by its users [28]. It is therefore important to clearly present the context in which a user is currently browsing a collection of data. The panes that present such navigation history are often referred to as *breadcrumbs*. A good breadcrumb shows the constraints on a browsable collection that are set by the user and allows easy undoing of previously added constraints. In figure 3.5 an example is shown where a collection of hard drives is browsed with an additional search term as constraint. The `X` in the upright corner of each constraint allowing to undo previously set refinements.

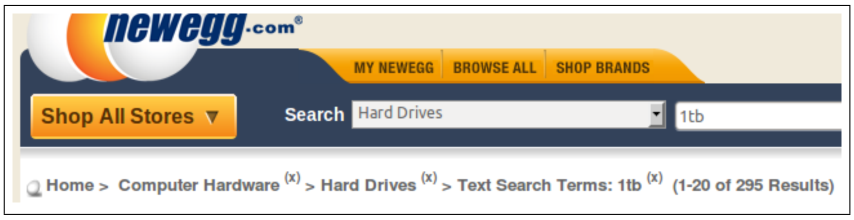
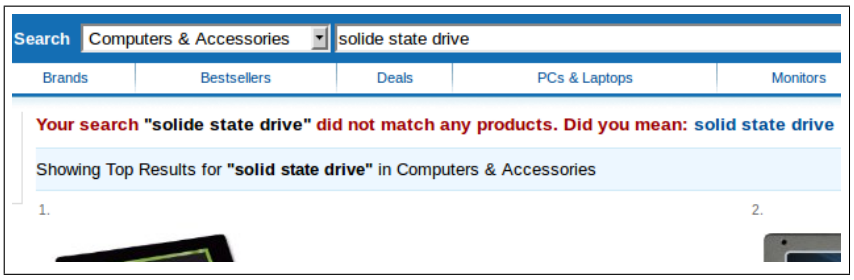

newegg.com®
MY NEWEGG
BROWSE ALL
SHOP BRANDS
Shop All Stores ▼
Search
Hard Drives
1tb
Home > Computer Hardware (x) > Hard Drives (x) > Text Search Terms: 1tb (x) (1-20 of 295 Results)

Figure 3.5: Example of navigation breadcrumb on NewEgg.com, showing browsed category and search term

In order to present the browse history, the web application must keep track of previously added constraints during a browse session. Removal or addition of constraints often require the complete query, which might be a search, database, xpath or any other retrieval query, to be rebuild. So in order to have adjustable browse history elements, all constraints (search query, filters, facet selections) need to be shared among browse requests such that a new context can be constructed when the user decides to add/remove a constraint.

3.2.4 Assist Your Users: Spell Checking



Computers & Accessories | solide state drive

Your search "solide state drive" did not match any products. Did you mean: solid state drive

Showing Top Results for **"solid state drive"** in Computers & Accessories

Figure 3.6: Example of spell checking in action on Amazon.co.uk

Offered by most web search engines, and nowadays by an increasing number of web applications, is a spell checking facility which suggests terms in case a user makes a typographic or spelling mistake in his search query. This is often referred as *did you mean?*-functionality. Sometimes queries are automatically corrected and results are displayed for the updated query in case some condition evaluates to true (for example when there is no or very few results compared to a corrected term).

In order to give meaningful suggestions terms in a search query need to be compared to terms from some dictionary. If a term is 'similar enough' (e.g. by measuring the Levenshtein distance between terms), but not equal to the original term, it will be candidate for suggestions. In order to serve spell check suggestions, one needs to decide which collection to use as a dictionary. It can be a list of words (e.g. from WordNet[2] or another dictionary). Another option is to reuse the terms from the search index. The advantage of the latter approach is that the suggested terms are part of the website's content and is therefore useful as search query term, where this is less likely when using an external dictionary. This will work well for single term suggestions, but suggesting *meaningful* multi-term suggestions will add some difficulties. The problem is that terms used in a search query are used in context of each other.

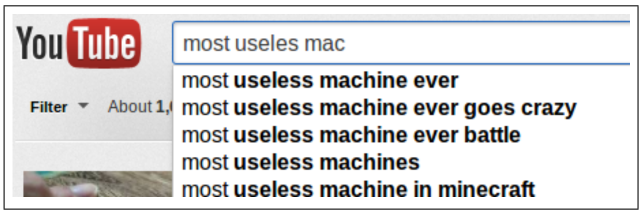
3.2.5 Assist Your Users: Meaningful Type-ahead Suggestions

Part of most modern browsers is support for search suggestions returned by web search engines. Web search engines and websites with a large user base may choose to reuse queries previously entered by users which are approved to be meaningful (e.g. when entered my multiple users, or when a minimum number of search hits are reached). For smaller websites without such usage data, *type-ahead suggestions* (or *autocompletion*) can be established similar to 'did you mean?'-functionality. An index can be constructed from a dictionary resource or an existing search index can be reused. The latter would provide more mean-

[2]`http://wordnet.princeton.edu/`

ingful auto complete suggestions in the context of the website for the same reason as for spell checking: it is able to serve terms or phrases that are part of website's content.

Instead of comparing similarity during spell checking, a prefix query would retrieve type-ahead suggestions from a search index. For multi-term (or phrase) suggestions, the index should contain phrases or positional information about terms such that useful suggestion phrases can be constructed. This can be taken one step further, namely by also recovering misspellings when suggesting completions. Figure 3.7 shows a example of Youtube in which the suggestion engine also performs some sort of spell checking making it able to serve suggestions even when a misspelling is made.



most useles mac

most **useless machine ever**
most **useless machine ever goes crazy**
most **useless machine ever battle**
most **useless machines**
most **useless machine in minecraft**

Figure 3.7: Example of autocompletion on Youtube which also corrects misspelled words while typing

3.2.6 Search Namespaces

Some websites can be served in different languages. Those websites often also offer search targeted to the language that is currently viewed. In that case, only part of the documents that is available in the viewed language need to be searched. In general, content might be divided into namespaces based on some (metadata) property. Other examples of namespace distinctions are: departments within a company, different product categories on an e-commerce website, different access levels to content served on the website. Also, multi-tenant applications where a single instance of software serves multiple customers, will have similar distinctions.

With respect to search, namespace division can be achieved by keeping different indexes for each namespace or by keeping all documents in a single (or distributed) index and by filtering documents in the targeted namespace. When deciding to do so during the development of a web application, other facilities that elaborate on the search index must support this way of namespace division, otherwise these facilities become unusable when used in a namespace-aware context or even in totality. In general, division into namespaces will add complexity to the implementation when the search engine in use has no notion of namespaces or when using various software artifacts that share the same index.

3.3 Scalability

When the size of search indexes grow and the number of searches increases, a single machine might become too slow to handle all this. In the scope of search, scaling means *replicating* search indexes to several machines, or by *sharding* (splitting) indexes and distributing them over multiple machines.

Index replication is used when a single machine cannot handle the amount of queries. A *master* machine keeps track of the index and *slave* machines create and maintain a copy of the master index. Instead of a single machine executing all search queries, the queries can be assigned to multiple machines, lowering the system load.

Another problem is when a search index becomes too large such that a single machine is unable to execute a single query within an acceptable amount of time. This is where *index sharding* comes into play. By distributing parts of the index over multiple machines, these machines can execute searches faster for a part of the index, and thus return partial results. The partial result must be merged together into a single list of results.

3.4 Maintainance of Indexes

When browsable data is changed, added or deleted, this changed data must be propagated to the search indexes in order to keep these usable for search. It would therefore be ideal to have a central place where these changes are processed in the implementation of the web application. This implementation can then be extended to further process these changes for indexing. A different approach is to fully or partially reindex data periodically. If an application has no fast changing data, and changed data is not required to become searchable quickly, this approach will work. However, when *(near) realtime search* is required, where data must become searchable (almost) immediately after change/addition, the first approach will be more appropriate. Other search features such as suggestional services and faceting might need similar treatment regarding changes to the browsable data.

Besides keeping a search index synchronized, continuous development of a web application may include changes that require data to be fully reindexed. For example, when the analysis for textual data (applied at both indexing and query time) is improved, or when additional data properties become searchable. The way how full reindexing is setup will control the duration of and the limitations on available functionality in a running web application. A special maintenance mode might be required during reindexing, but should be avoided if possible. One solution might be to reindex data using a separate process running in parallel with the online web application, creating a new index but leaving the current one as is. When the reindexation process is finished, the index in use by the online web application can then be replaced by the newly created one. The newly (improved) web application can now be deployed.

3.5 Towards a Solution

Many aspects can be considered when integrating internal site search. There is no single solution that fits all contexts of web applications. The retrieval and selection of searchable data, and the way how this data should become searchable (i.e. the way this data is analyzed before index addition) is specific to the context and domain of the web application. This also applies to the design and implementation of the search user interface. Here, the variability is in the design of the document surrogates presented on a search results page and the ability to navigate to browsable sub-collections (and the presentation of these). Furthermore, a developer of the search user interface must decide to which extent a user is able to control the search and if the interface should feature suggestion services like autocompletion and spell checking.

The DSL extension that is to be integrated in WebDSL should allow a WebDSL user to express the variable elements of search, without the need to dig into underlying implementation details. The DSL should do this by translating expressive constructs into executable code. Existing open source search solutions (engines and libraries) can be used to power search functionality in the backend of WebDSL. In the next chapter, we will look at a range of software solutions to power search. A comparison will expose which solution to be the most suitable for use in WebDSL's backend.

Chapter 4

Solution Domain

Search engine technology can be considered mature in the sense that a range of libraries and frameworks are available for many programming languages. Studying a *program family* is an excellent source for inspiration during language design. A program family is a set of programs that share common properties, making it advantageous to study these common properties before analyzing individual programs of that family [23]. It helps understanding the concepts and best practices adopted by these programs. Analyzing members of a program family for repetitive program patterns may yield to candidate domain abstractions to be supported syntactically by the DSL [4]. In our case, we will study a program family for enabling search functionality.

4.1 Answers to Internal Site Search

In this chapter we take a look at the software solutions that try to offer the functionality and techniques discussed in Chapter 3. Additional software artifacts may be needed to fully cover search functionality commonly used on the web. We will focus on search libraries and engines that can be used in the scope of extending the WebDSL language.
This requires the software to:

- allow interaction from within Java code (WebDSL's target language)
- be freely available and open source

Software solutions respecting these requirements include Apache Lucene (being an open source search library) and a range of search libraries and engines based on this library. We made a selection of frameworks that have an active community, and compared them on a number of aspects related to site search. Additionally, we take into account the integration into the WebDSL framework with respect to its implementation as it was at the start of this thesis project.

4.1.1 Apache Lucene

Apache Lucene[1] is a well established search library. Its first open source released version (0.1) was in 2000 on source forge and it became part of the Jakarta family of Apache Software Foundation in 2001.In 2005 it became a top-level project of Apache. Apache Lucene is a simple search library and a range of information retrieval engines are build around Lucene (like search engine SOLR, web search engine/crawler Nutch and recommender system Mahout also from the Apache Software Foundation). Lucene offers basic indexing and search functionality and additional modules (like spell checking and hit highlighting) are contributed and packaged with the binary package available for download. During the exploration in the solution domain, Lucene was at version 3.1.0.

4.1.2 Apache SOLR

SOLR[2] (pronounced 'solar') is a sister project from Apache Lucene and adds a search server with a range of search components making it a complete search server for most applications. SOLR adds integrated support for search functionality like faceting, hit highlighting and geospatial search. It comes with a framework of analyzer components covering most text analysis requirements during document construction and querying. During the exploration in the solution domain, SOLR was at version 1.4.1.

4.1.3 Elastic Search

Similar to Apache SOLR, Elastic search[3] offers a search engine which is based on Apache Lucene. Elastic search' predecessor is Compass, being an open source search engine with its focus on seamlessly integrating search into Java applications. Later, support was added for easy distributed setup and better integration with object-relational mapping (ORM) frameworks, like Hibernate ORM.

A newer version of Compass would support easy use of advanced features like faceting and flexible, seamlessly scaling from a single to multiple machines. It would also have support for other languages than Java. An update to Lucene (2.9) also required code to be rewritten, and because of these changes and feature additions, main developer Shay Banon decided to write a new search engine from scratch, named Elasticsearch. It has a RESTful interface using JSON over HTTP and has a Java library to be used from within Java. The most recent version at the time of analysis of the solution domain was 0.15.0.

[1]http://lucene.apache.org

[2]http://lucene.apache.org/solr

[3]http://www.elasticsearch.org

4.1.4 Hibernate Search

Jboss offers a collection of tools under the Hibernate umbrella that enable persistence of data using POJO domain models. Hibernate Search[4] falls under this umbrella and has the great advantage of seamless integration with Hibernate ORM, which is used in the back end of WebDSL for mapping POJOs (plain old java objects) to a relational database. Hibernate Search uses Lucene as search library and provides an interface for accessing its Lucene indexes. The most recent version at the time of analysis of the solution domain was 3.3.0.

4.1.5 Sphinx

Sphinx[5] is a search engine that was built to integrate well with MySQL. Setup is easy by using SQL queries to select the data for index addition, with database rows as documents and columns as the search fields. The engine offers basic search functionality.

The last stable released version of Sphinx is dated December 2009 (version 0.99). Because of its low release frequency of stable versions, we decided to look at the latest released beta version, which is numbered 1.10 beta (released July 2010),

4.2 A Comparison of Search Engines

In this section, a comparison is made between the search engine solutions just discussed. We compare feature support for most important aspects related to the internal site search domain and their suitability for integration in the backend of WebDSL.

4.2.1 Document Construction Features

One task during index construction is the creation of documents from the content (data) that can be browsed in a (web) application. A developer needs to decide which data entities should become represented by documents, and which data to become fields for each such document. Most search solutions incorporate the notion of *field types*, where a field can be assigned a type that describes how data must be tokenized and normalized. These types can be referenced during the creation of a document, such that the indexer (part of the search engine or library) knows how to put that data into the search index. At query time, the search engine uses this information to perform appropriate tokenization/normalization on a query (probably the same as at indexing time). SOLR and Elasticsearch adopt the notion of types and allow specification of an *analyzer* for each type with an optional distinction between indexing and query time analyzer (e.g. for query-time term expansion). An analyzer

[4]`http://www.hibernate.org/subprojects/search`

[5]`http://www.sphinxsearch.com`

describes how textual data must to be tokenized and normalized. SORL provides an analyzer framework in which three types of building blocks can be used to create an analyzer that performs tokenization and text transformations as specified. An analyzer consists of a tokenizer and optionally one or more filters:

- *character filters*: These filters are applied before the tokenization of text. Character filters might be useful to strip off text (HTML tags for example) before the text gets tokenized.

- *tokenizer*: A tokenizer transforms the input stream of text into seperate tokens. A simple tokenizer might split tokens on white spaces and interpunction symbols. More advanced tokenizers are able to distinguish acronyms, names, phone numbers and (web and email) addresses from other words and treat them as single tokens. Regular expressions can also be used to filter tokens from a stream of text.

- *token filters*: Applied after tokenization, to further transform the tokens delivered by the tokenizer. SOLR includes a range of token filters. Commonly used filters are the lowercase, stop word, synonym and stemming filters. The synonym filter is used to expand a single token into multiple tokens that share the same meaning (in the context of the application) and uses a synonym mapping file to specify this expansion. Regarding stemming (transforming words to a root form), multiple filters are available for various languages using different algorithms. A set of filters to perform phonetic matching is also included. These filters transform tokens into their phonetic encodings and are most useful for matching misspelled, but phonetically similar names or words.

In figure 4.1 a visual representation is shown of an analyzer consisting of a character filter that strips off HTML tags, followed by a tokenizer and normalization by a lower case filter and stop word remover.

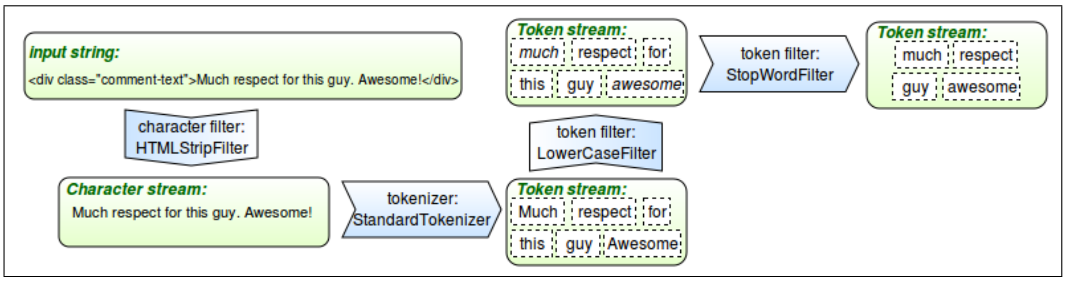

Input string:
<div class="comment-text">Much respect for this guy. Awesome!</div>
character filter: HTMLStripFilter
Character stream:
Much respect for this guy. Awesome!
tokenizer: StandardTokenizer
Token stream:
Much respect for this guy Awesome
token filter: LowerCaseFilter
Token stream:
much respect for this guy awesome
token filter: StopWordFilter
Token stream:
much respect guy awesome

Figure 4.1: Example of Lucene/SOLR analyzer consisting of a character filter, tokenizer and 2 token filters

A set of useful filters and tokenizers is supplied with Lucene and SOLR, which can be accessed from within SOLR. In case these analyzer components do not suffice, one can implement their own which can be used in conjunction with the supplied tokenizers/filters by implementing the corresponding interface. Elastic search and Hibernate search reuse the analyzer components provided by the Lucene analyzers library and SOLR. Elastic search is limited to the analyzer components that come with the Lucene/SOLR packages, i.e. no custom implementations are supported.

The core Lucene library let index addition and mapping of types up to the users of the library: there is no mapping of which analyzer to be applied on which field. Each time a document object is created, the fields with data and reference to a built-in or user constructed analyzer must be passed. Analyzers are supplied with Lucene itself, but the analysis package of SOLR can be used to build analyzers from the filters and tokenizers.

Analysis in Sphinx is different. First, Sphinx is focused on seamless integration with SQL databases. Searching data is similar to querying your database using tables and columns. There is no notion of types, but fields are the columns as they appear in a database table. Each field can be configured with options to enable normalizations and token filtering (like stemming, stop word lists, query expansion). This could be sufficient for needs in a web application but as soon as different normalization is required, like for example the tokenization of *camelcased* words (e.g. a mobile app name like 'SoundCloud' could be tokenized and normalized into [sound, cloud]), Sphinx would require to normalize the data beforehand and store this normalized data in a separate database table for indexing. This makes Sphinx Search less flexible regarding text analysis.

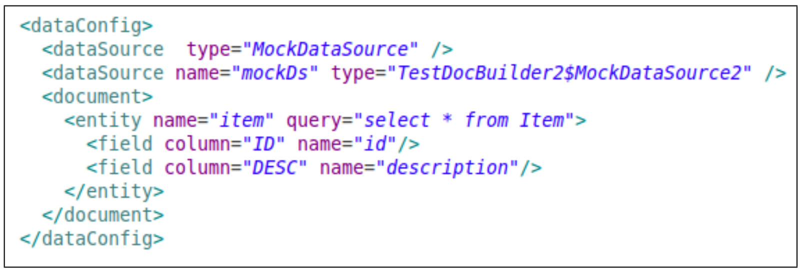
	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
Text analysis	+/-	++	++	+/++	+/-

4.2.2 Database Integration

Content to be served in web applications can be stored in various ways like relational databases or XML documents. Typically, data retrieved from a search engine are documents, each carrying a subset or derivation of the original data. The presentation of search results, or more general browse results, often includes data that is not stored in the search index' documents. The missing data needs to be retrieved from the primary data store where an application's content is stored. For this reason, it is often required to map a document to the original data, by maintaining a document field with a unique identifier that maps to a unique data entity in the primary data store. In the core Lucene library there is no built-in facility to specify such mappings, neither do Elasticsearch, SOLR and Sphinx have such facilities. So, in order to retrieve the original data entities represented by search results, a mapping module must be implemented that takes documents as input and returns the data entities from the primary data store represented by these documents.

SOLR Data Import Handlers

While there is lack of mapping support at retrieval time, SOLR provides so called *data import handler*s to ease the indexation of data. These handlers (part of SOLR contrib module) allow a configuration driven way of importing data into SOLR's search index. Common data sources like relational databases, XML and HTTP based data sources are supported, and one specifies which data (using queries or xpath expressions) to get indexed in which field, optionally first applying a so-called *data transformer* (which should be seen as data preprocessor). Specification of which data to be indexed and how is done in a *data source*, of which Figure 4.2 shows an example. SOLR's data import handlers allow both full and incremental indexations, so it is possible to only add new data to the index without the need rebuild the whole index. However, in order to process *deleted* searchable data, some trace must be left in the primary data store in order to let the data import handlers delete the associated documents from the index. The application might need adaptation in order to ignore deleted documents that are still in the primary data store just for the sake of index deletion.



```
<dataConfig>
  <dataSource  type="MockDataSource" />
  <dataSource name="mockDs" type="TestDocBuilder2$MockDataSource2" />
  <document>
    <entity name="item" query="select * from Item">
      <field column="ID" name="id"/>
      <field column="DESC" name="description"/>
    </entity>
  </document>
</dataConfig>
```

Figure 4.2: Example of a SOLR data config specification

Data import handlers are used periodically to process changed data. This exposes another potential problem, namely the asynchronicity between the search index and the actual data available for browsing in an application. In case changes to data need (near) real-time propagation to the search index, using a data import handler for incremental backup is not the way to go. It would then be better to extend the process of data modification with propagation to the search index by instructing the SOLR search server to add or remove a document. This is similar to the workflow of indexing in Elasticsearch.

Elasticsearch

Elasticsearch has no notion of data import handlers and the best way to initially index existing data is to launch multiple threads that retrieve data from an application's data source, construct JSON documents and post these to the server (Figure 4.3) which will distribute this index change to other nodes if configured this way.

Compass, the discontinued predecessor of Elasticsearch, supported seamless integration with ORM software like Hibernate ORM, for automatic retrieval and index propagation of objects (data entities) in case there was a change in data. Unfortunately, such integration is not supported (yet) in Elasticsearch, and the translation and synchronization between documents and the actual data is carried out to the developer who uses Elasticsearch.

Seamless Integration with Hibernate Search

Hibernate Search, part of the JBoss Hibernate framework, integrates seamlessly with their object relational mapping software Hibernate ORM. The search index is kept in sync with the data source. One only needs to specify the desired data properties to become search fields by using Java annotations or through their programmatic API. A component called

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}
```

Figure 4.3: Posting a document to Elasticsearch server

mass indexer can be used to initially index or reindex all instances of an entity type. Index configurations can be setup for each entity class and searches are scoped to these entity types. Upon retrieval of search results, a list of the desired entities (as they would be retrieved using their ORM framework, i.e. Java objects) is returned. Entity inheritance is also dealt with transparently: when targeting a particular entity class, it returns all entities that inherit from this class.

Sphinx Data Sources

Regarding index configuration, Sphinx let us specify searchable data through SQL queries. So-called *data source*s represent a source to be used for an index, and its specifications includes an SQL query that selects the rows and columns to be used as documents and search fields respectively. Additionally, *attributes* can be specified, which are values that will be stored in the index enabling sorting and additional filtering using these values. Multiple *index* specifications may refer to one and the same data source. This enables the indexation of the same data in various forms (i.e. applying different analysis). Specification of analysis components to be applied during index construction is bound and part of an index specification. Changes to data in a data source are not automatically synchronized to the index. Reindexing of the data is a common task when using Sphinx in an application with fast changing content. Sphinx can also be configured to use *delta indexes*. Delta indexes are actually normal indexes used in combination with a *main index*, and are configured to only index new documents that do not yet appear in the main index. New data is distinguished from ‘data already in the main index’ by keeping track of a counter (like an incrementing primary key). A delta index can be merged with a main index, or the main index can be rebuilt from scratch to keep the delta index small. A small delta index can be reindexed periodically such that changes to an application’s content are quickly reflected in the index. As of version 1.10beta, there is also support for *real-time indexes*. These indexes are not using a data source anymore, but documents are added to these indexes using SQL queries. It is similar to document addition in Elasticsearch or SOLR, where the data to put in a document’s field are specified, and the document is added. However, real-time indexes are added in version 1.10beta and is labeled work-in-progress. Some features are not yet supported, like wild-card queries and multi-valued attributes (a multi-valued attribute may hold a collection of values instead of one, e.g. for product categories or tags).

Search results retrieved from Sphinx are the indexed documents. So, just like in SOLR, Elasticsearch and Lucene, additional data must be retrieved from the primary data store using an identifier which is part of the indexed documents.

	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
DB handling/synchronization	-	+/-	++	-	+/-

4.2.3 Query Type Support

In our exploration of the solution domain, we looked at the types of queries that are supported by the selected frameworks. All observed frameworks support the commonly used query types: term, phrase, range, wildcard and boolean queries. Sphinx has no support for fuzzy queries, so it won't be able to fuzzy match a query term 'defin*a*tely' against an indexed term 'definitely'.

Query syntax

Besides building query objects programmatically, Lucene offers a query syntax to specify a range of queries, including the above mentioned query types. This syntax is available in all Lucene based search frameworks. SOLR offers additional modes with syntax for queries, with *disjunction max* (DisMax) being the most popular syntax, because it allows to search multiple fields at once and it is designed to never throw an exception during parsing. It recovers from erroneous input by treating problematic parts as string literals. The scoring of documents is also different. A should-clause is generated for each queried field. Where the default parser scores clauses equally, SOLR's DisMax query mode will only use the score of the field clause with maximum score for a term. This makes query-time field boosting more effective. The disjunction max query is also supported by Lucene and Elasticsearch. The DisMax query syntax is only available in SOLR.

Sphinx also comes with various matching modes with different query syntaxes. Its extended query syntax supports most query types.

Filters

For constraints that are to be applied repeatedly, most search frameworks offer a filtering mechanism. In Lucene, SOLR, Elasticsearch and Hibernate Search, filters are bit sets, where each bit represents a document in the index. Only documents with value 1 (or 0 in case of a negated query) in the bit set are considered as possible hit when processing the query. The filters can be constructed from any query result. When a filter is applied the first time, there is some overhead for constructing the bit set, but these bit sets are cached and reused when applied a second time. Thus only for recurring constraints it is advised to use filters for better efficiency.

Sphinx also supports filters. Filtering (and sorting) is limited to the data that is modeled as attributes in the data sources. Sphinx attributes are stored as integer values. This limits the kind of constraints that can be modeled as filters. Also, more work is needed to filter on string values, because these are translated to integer attributes by Sphinx. In order to filter on string values, we must first convert the strings to (for example) their CRC32 value at indexing time, and repeat this for the constraint values at query time.

Geospatial search

Sphinx supports filtering on geographic distance by letting you set a targeted location (the center location). It can then serve results that are within a specific range and/or order the results by distance from the center location. This is also supported in Elasticsearch and it even has faceting support for it, which allows retrieval of distance ranges (like results within 10 km, or between 50 and 100 km from a targeted location). For Lucene, there exists a contrib module for geospatial search, but in version 3.0.3 of lucene, this version was limited and was known to have some bugs. SOLR did not yet support this module. An external SOLR plugin was available that added spatial search features. Like Lucene, Hibernate Search 3.3.0 is limited to the geo-spatial module from Lucene contrib.

	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
Query type support	+	++	+	+	+/-
Geospatial search	+/-	-	+/-	++	+

4.2.4 Serving Spell and Type Ahead Suggestions

A good search user interface supports the user by suggesting corrections on mis-spelled/typed words, or better: suggest completions while a query is being entered. Regarding spell checking, Apache Lucene provides a spell checker component in the contrib package. This spell checker extracts terms from an existing Lucene index or text file, and builds a separate index for spell check purposes. After creation of the spell index, the spell checker module can be used to suggest terms that are similar to some extend. It uses a similarity algorithm (Levenshtein Distance by default) to compare a query term against the terms in the index. The threshold of similarity for (not) returning a term can be configured. The spell checker also offers functionality to return suggestions on query terms that do exist in the spell check index (i.e. it are valid terms), by only suggesting terms that are *more popular*. A term is more popular if the *term frequency* (*tf*) of a correction is higher than the *tf* of the original query term (it obtains the *tf* information from the search index). The downside of using this module is that it needs to create an additional index in order to work, and this index might need to be renewed periodically.

Besides Lucene, this spell check module is available for use in SOLR. It can be used separately to retrieve suggestions for a query through the spell checker request handler, or by applying it as part of an ordinary query request, where SOLR tries to correct a query directly instead of using multiple request (one for spell checking, one for querying). While there is no built-in support in Hibernate Search for spell checking, the spell checker module from Lucene's contrib package can be used on the indexes that are maintained by Hibernate Search. This module is not yet exposed in Elasticsearch, because of the required maintenance of an additional spell index. Elasticsearch therefore lacks support for spell checking. The same holds for Sphinx, there is no notion of a spell checker or suggestion module.

The lack of spell checkers can be overcome by using external libraries like Aspell (LGPL license). Most spell checking software comes with bundled dictionaries for various languages. However, such dictionaries often overlap only partly with the actual terms in the searchable content (i.e. terms for which the search engine can return results). So if a dictionary lacks terms that are common in the application's content, it might mistakenly correct terms that are meaningful within the application or return terms that make no sense in the context of the application. In order to return meaningful suggestions by the spell checking component, it will in most cases be required to build a custom dictionary, preferably based on searchable content. This means that the searchable data from a data store/search index first needs to be converted into the dictionary format the spell check component accepts. If it is also desired to serve suggestions on *correctly* spelled terms, in case there are *more popular* terms, interaction with the search index or another data store that contains information about term frequencies and/or term order is required.

Type Ahead Suggestions

Although completing queries while users are typing is a common way for supporting the user in his search for information, there is no explicit notion of an auto completion module in any of the discussed frameworks. This does not imply that it cannot be achieved using built-in facilities of the search engines/libraries. One way to setup such functionality is to reuse the terms that are in a search index and treat these as documents itself in an additional index. Auto completion suggestions can now be retrieved using *prefix* queries. Additional search fields might be used to order the completions in a way that makes sense to the application, for example the term frequency such that terms that appear more frequently are ranked higher than others with that share a common prefix.

Another way to get auto completion within an application is to search instantly on every keypress, using prefix queries again. This will increase the performance requirements due to the retrieval of results on every keypress, but the user can be served more information than a set of strings.

Multi-term Suggestions

When a user is typing a query 'computer mon...', the system should suggest 'computer monitor', and not 'computer monkey', while 'monkey' might be a good completion when completing a single termed query. The same holds for spell suggestions. In order to provide multi-term suggestions, information must be available about the appearance of terms next to each other. A possible solution to accomplish this is by storing each n adjacent tokens (n >1) in the index, such that a set of n-length *shingles* can be used to construct type-ahead or spell suggestions.

	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
Spellcheck suggestions	+/-	++	+/-	-	-
Autocomplete suggestions	-	-	-	-	-

4.2.5 Hit Highlighting, Fragment Selection

Instead of presenting a list of result identifiers (like book titles or product names), a user is better assisted when each result is accompanied with one or more fragments that emphasize why a result is considered relevant. This eliminates the effort of browsing into each result (probably by clicking on it), followed by scanning the browsed result in order to find relevant fragments and, in case this result does not contain the desired information, browsing back to the result page to review other results.

The compared solutions do all provide functionality to extract relevant fragments from a text (based on a query or query words) and are able to highlight tokens within a fragment that match a query after normalization (Lucene and Hibernate Search through the use of Lucene contrib module). SOLR has a little more highlight options than Elasticsearch (e.g. specify boundaries for a fragment such that the fragments returned are complete sentences).

	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
Hit highlighting	+/-	++	+/-	+	+

4.2.6 Faceted Search

Faceting, the presentation of a taxonomy within a currently browsed collection, is supported by SOLR and Elasticsearch out-of-the-box. The observed versions of Lucene and Hibernate Search had no built-in support for faceting at that time (as of version 3.4 Hibernate Search added partial support for faceting, Lucene added faceting module in its contrib package in version 3.4).

Sphinx supports faceting implicitly. Besides constructing a main query, one may construct additional queries for faceting. Sphinx features *group-by* functionality natively (where Lucene does not), which means that it is capable to return groups of results, where each group of results has a value (or range of values) in common. Each group's value can be used as browsable facet category and the the size of a group will match the number of elements within the targeted facet category. So in order to have faceting using Sphinx, besides having a main query, additional queries must be specified for each facet. Each such query is a possibly extended main query (with additional clauses for already filtered facets) and should be set to retrieve results grouped by the column values for the targeted facet (column values are similar to field values in Lucene, as Sphinx reuses concepts of relational databases). The additional query clauses in the extended main query may vary for each facet to be retrieved. This is because some facets may be expandable (a user may select multiple values within the same facet category) where others only allow a single value to be selected.

SOLR and Elasticsearch both have support for faceting. These make explicit notion of the facet concept in their API. The faceting APIs allow to set parameters like:

- the targeted search field of a facet
- the number of facets to return

- the order of facets (number of hits, alphabetic, ascending/descending)
- whether or not to retrieve the number of documents that have no value set for the facet field
- the minimum number of hits a facet value should have in order to retrieve it

SOLR and Elasticsearch have support for discrete faceting (single value per facet, e.g. for tags) and range facets (range of values per facet, e.g. for price range facets), although range facets in SOLR were limited to date ranges in version 1.4.1. *Hierarchical faceting*, where a facet has multiple levels (e.g. on a retailers website the facet 'electronics' has multiple browsable sublevels like 'electronics/computers' 'electronics/audio/headphones'), is not supported by Elasticsearch. SOLR has no explicit support for this, but it can be managed using its *facet prefix* filter. The idea of this approach is that search fields used for hierarchical faceting hold one or more values for each level a document belongs to, with the level information encoded in the search field's value. If done properly, this encoding, which is performed at indexing time, allows one to retrieve facets for a specific level using a facet prefix filter. Exclusive in Elasticsearch are histogram and statistical facets. They let you retrieve counts over a range of intervals (histrogram) and statistical data including (not limited to) mean, minimum and maximum. Moreover, Elasticsearch provides geo distance faceting which provides information for ranges of distances from a provided center location.

The compared versions of Lucene and Hibernate Search had no built-in option to perform faceting. Luckily, there is a partial solution for this job. An open source software package called Bobo-Browse[6] can retrieve facets from and perform searches on Lucene indexes. It also features hierarchical faceting.

	LUCENE	SOLR	HIBERNATE SEARCH	ELASTICSEARCH	SPHINX
Faceting	-	+	-	+/++	-

4.2.7 Scalability

Elasticsearch is the most powerful when it comes to scalability. You can set up a cluster of multiple Elasticsearch nodes (machines), which are able to discover each other. Each node will also be informed when another node goes down, and a cluster is able to manage distribution of its indexes (or index shards) automatically. Multiple indexes can be configured on each (possibly single node) cluster and each index can be set to have a number of shards and number of replicas. The number of shards is only configurable at index creation, while the number of replicas can be increased at runtime (changing the number of shards requires a rebuild of an index, which is expensive). The idea of Elasticsearch is to use the number of shards as a scaling factor. A shard can easily be replicated or moved to another node without the need of reindexing, because the whole shard is copied/moved. So if an index is set to have 4 shards, which are currently maintained by a single node cluster, adding a second

[6] `https://github.com/javasoze/bobo`

node to that cluster would cause Elasticsearch to automatically distribute half of the shards to the second node. This is similar for the number of replicas of an index. If the number of replicas is changed at runtime, Elasticsearch will replicate indexes (possibly distributed over multiple shards) to the available nodes. A shard will never be replicated within the same node. Figure 4.4 illustrates how shards and replications are distributed on adding a node to a cluster.

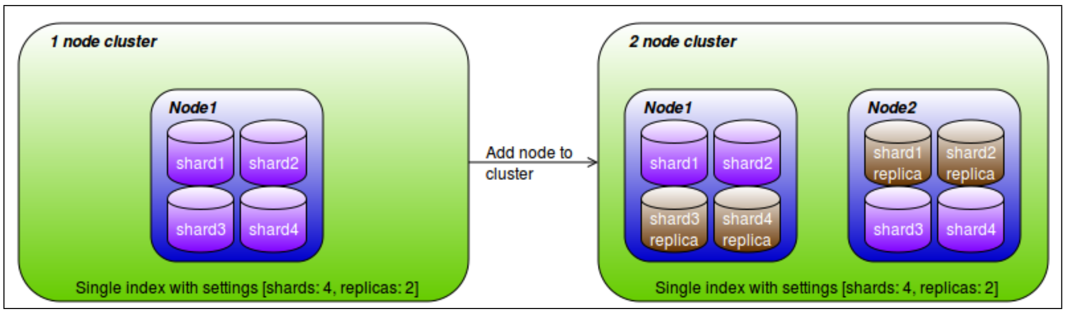


Figure 4.4: Elasticsearch automatically redistributes its shards and replicas on node addition

SOLR also supports sharding and replication, but distribution and management of shards and replicas must be managed outside SOLR. This is similar for Hibernate Search. A sharding strategy can be configured (e.g. to use different shards for content of different language) and servers can be configured as master or slave for replication of search indexes/shards. Replication is absent in Sphinx, although index sharding can be configured. Multiple Sphinx instances may maintain different index shards and one instance will be configured with a distributed index configuration. By querying this instance, it combines the results from all shards. The core Apache Lucene library has no replication or sharding functionality built in. It must completely be managed within the application code itself.

	LUCENE	SOLR	Hibernate Search	Elasticsearch	Sphinx
Scalability	+/-	+	+	++	+/-

4.3 Choosing a Solution Suitable for WebDSL

All results of our comparison are summarized in table 4.5, showing the strengths and weaknesses of each search solution. An important conclusion we may draw is that there is no single solution that covers all features out-of-the-box. Regarding search features, Elasticsearch and SOLR are the most rich search engines. However, compared to Hibernate Search, these search engines require more effort to get data synchronized between the primary data store and search index. For that to work, a mapping must be maintained that keeps track of which field type and/or field properties are bound to which POJO properties. At runtime, this mapper should be used when propagating data changes to the search index. So we need to keep track of which data is changed, added or removed. Moreover, a document in the search index might also need to be updated even when this document is not representing a changed entity. This is the case when the document contains data that resides in a different POJO than the one it represent. For example when a Product object `p` has a searchable property referring to a Category object `c`. At some time the property `c.name` changes from ‘notebooks’ to ‘laptops’. This should not only trigger an update to the document that represents `c`, but also for `p` and any other document that uses data from `c`. Then, a mapper is also needed that can convert documents retrieved as search results to objects as they are used in the application code.

Other issues arise when looking at the integration in WebDSL. SOLR and Elasticsearch normally run as separate servlet and separate java process respectively. In order to use SOLR, there are some servlet container specific settings which need to be set. Additional difficulties come to light when we want to run multiple WebDSL applications on a single machine: Are we going to use a single instance of SOLR/Elasticsearch server and keep track of the running applications, or are we going to use multiple instances of SOLR, one for each deployed WebDSL application? For both cases it means that a person deploying a WebDSL application, or each instance of a deployed WebDSL application itself, needs to keep track of the search engine(s) that are running, and start/stop them whenever this is needed. Luckily both SOLR and Elasticsearch provide Java APIs that allow to run the servers embedded within an application.

The latter is not the case for Sphinx, being implemented in C++. Different binaries are required on different platforms. For *nix platforms, a source code tar archive is distributed, which needs to be compiled. Together with the fact that it misses opportunities to have spellcheck and autocompletion, Sphinx is a less attractive solution for integrating search in WebDSL.

	Lucene 3.0.3	SOLR 1.4.1	Hibernate Search 3.3.0	Elasticsearch 0.15.0	Sphinx 1.10 beta
Text analysis	+/-	++	++	+/++	+/-
Query type support	+	++	+	+	+/-
Geospatial search	+/-	-	+/-	++	+
DB handling/synchronization	-	+/-	++	-	+/-
Spellcheck suggestions	+/-	++	+/-	-	-
Autocomplete suggestions	-	-	-	-	-
Faceting	-	+	-	+/++	-
Hit highlighting	+/-	++	+/-	+	+
Scalability	+/-	+	+	++	+/-
License	Apache License v2	Apache License v2	LGPLv2.1	Apache License v2	GPLv2

Table 4.5: Comparison of feature support in search solutions: Lucene, SOLR, Hibernate Search, Elasticsearch and Sphinx

Out of the box, Hibernate Search is not as complete as Elasticsearch or SOLR are. However, it completely handles data store/search index synchronization when using Hibernate ORM. Being based on Lucene and because of the ability to access its underlying Lucene indexes, Hibernate Search’ functionality is extensible by using contributed modules or external libraries that work with Lucene. This way, absent features like hit highlighting and spell check services can be realized using Lucene’s contrib modules. Faceting can be realized using an external library like Bobo-Browse. Only some auto completion module must be implemented in order to return suggestions based on searchable content.

Because of the little setup effort that is needed for integration into WebDSL and the ability to have most search features accessible without implementing this functionality ourselves, we decided to use Hibernate Search as our search backend. This means that the Java code that is generated from WebDSL application code will interact with Hibernate Search. The process of integrating search features into WebDSL itself is done incrementally, where we started with basic search functionality and minimal document construction options. The approach is explained in the next chapter.

Chapter 5

Extending WebDSL with Search

In this chapter, we will describe the approach applied in integrating search into WebDSL. This process comprises the extension of an existing language (WebDSL), which already covers a large range of concerns in the field of web development. Being a DSL extension, the search language should be compatible with the grammar and concepts of the base language WebDSL. For instance, the index configuration in which one specifies which data object must become searchable relates to the data model of WebDSL. Search expressions may be embedded at various places, such as WebDSL functions, action code or variable declarations. Regarding the development process of the language, some restrictions are inherited from previous WebDSL development. These include the implementation platform and compiler architecture that is being used. We will first discuss the process of DSL Development in general, followed by a description of the process of previous WebDSL development. We then describe development process of the search DSL.

5.1 DSL Development Process

DSL development is mostly a creative and iterative process of various tasks. In [21], Marjan Mernik et al identify 5 key phases during DSL Development: *decision*, *analysis*, *design*, *implementation* and *deployment*. The development of a DSL is not a sequential process of these phases. Different phases can alternatively and repeatedly be dealt with during the process: in order to decide whether or not to develop a DSL, a preliminary analysis may be performed. Further analysis of existing software artifacts, like software families used in the targeted domain, may then give a better understanding of domain concepts and their reusability in the implementation of the DSL. This may in turn be related to the design of the DSL, which can be based on an existing DSL (language exploitation through extension, specialization or selectively picking of language features). Or the DSL is created from scratch (language invention) and does not reuse language features from an existing language.

Then, the *implementation* of a DSL determines how a DSL-program eventually gets translated into executable code. One approach is to rely completely on an existing language (the *host language*) by implementing the DSL conform the syntax of the host language. We call this an *internal DSL*. This often results in a language with noise inherited from the notation of the host language. Static error checking and reporting is also limited to that done by the host language, making it less tailored to the domain in question. Better analysis and reporting can be achieved when implemented as interpreter or compiler. In case of an interpreter, DSL constructs will be processed and executed at runtime (by fetch-decode-execute cycles) without analyzing the complete program beforehand. The latter is the case when a language is implemented as compiler. For this reason, compilers are in general better in performing optimizations and reporting errors, because they can access/analyze the whole program. Interpreters will be limited to analyzing statements (or sentences) which it then translates (decodes) to machine code to be executed. Advantages of interpretation over compilation are simpler implementation, easier extensibility and greater control over the execution environment. However, it requires more processing power at runtime. Another approach is to implement the DSL as preprocessor. In this approach, the DSL-code is translated to a base language by one or more, possibly pipelined, language processors. Static analysis is limited to what is performed by the compiler/interpreter of the base language.

5.1.1 Extending a DSL

In this thesis, the design pattern we deal with is *language extension*, where an existing language (WebDSL) is extended within its syntactic and semantic framework [29]. In our case, a more complete DSL for development of web applications with rich data models. When developing a DSL extension, previous decisions during the development of the base language to be extended will put restrictions on the design and implementation of the DSL extension. WebDSL is implemented as source-to-source compiler (an application generator) by transforming DSL language constructs into Java classes using model-to-model and model-to-source transformations. Generated java classes and configuration files are then compiled and packed into a deployable web application archive (WAR-file).

The implementation platform that has been used is the *language workbench* Spoofax [17], offering features to implement DSL's with plugin support for the integrated development environment (IDE) Eclipse. The grammar of the DSL is specified using the *syntax definition formalism* [33] (*SDF*), integrating lexical and context-free syntax into a single formalism. Spoofax creates a parser from the SDF-definition, which is used to create an abstract syntax tree representation from program code written in the DSL. This tree representation of the program is then used to further process the DSL program. Spoofax uses the transformation language and toolkit Stratego/XT [2] to specify rewrite rules in order to perform model-to-model transformations.

Other dependencies in the development of a language extension are the existing concepts and related language constructs of the base language. For instance, a data model in WebDSL consists of entity definitions with properties, each property having a name and type. WebDSL also allows entity definitions to be partly defined at multiple places. Furthermore, an entity definition may inherit the definition of a parent entity (inheritance). An index configuration for search will have relations to WebDSL's data model concepts in order to make the data searchable. Similarly, expressions for actual searching must be introduced in a way that these can be used like other WebDSL expressions in pages, templates and functions. The creation of a syntax for search is therefore (partly) restricted due to the required interaction with language constructs from the base language. Keywords used within the syntax are still free to choose.

Although we are restricted in the implementation by a predefined compiler architecture and grammar of the base language, we are still free in choosing the approach for development of the DSL extension. We will now look at the development process applied during previous development of WebDSL.

5.2 Previous WebDSL Development

Several methodologies and patterns for domain-specific language development are discussed in literature [30, 21], which often include a thorough domain analysis phase with domain abstractions being described formally [5, 27]. Because most approaches do not consider implementation details during domain analysis and language design, there might be lack of existing software artifacts (like libraries and frameworks) that are suitable for the designed language, adding up the effort required for implementation of the language. WebDSL has been developed differently, more in a bottom-up fashion where domain analysis and implementation go hand-in-hand. Analysis of the domain elaborates on the maturity of a solution domain: when mature software artifacts exist in the solution domain (libraries and frameworks with a great user base and community), these artifacts are assumed to have a well established interface adopting objects and operations from the targeted domain. The advantage of predicating upon existing implementations is a more efficient DSL implementation process. However, one must be careful that the syntax and structure of libraries/tools in the solution domain should not drive the design of the DSL grammar, as this may not necessarily reflect the actual needs in the targeted domain.

5.2.1 Iterative Incremental Design

Studying the development process so far, WebDSL is developed iteratively, where basic language features were first designed and integrated, and domain coverage is then iteratively extended by integrating more features [34]. Instead of building a complete system in the end, the development iterations deliver intermediate states of the system which can be verified on correctness and completeness. This offers quick feedback on newly added abstractions, and insight in the priority of future abstractions. Reasoning at this level, the development of the search extension can be considered such an iteration: we increase the coverage of the web application domain by introducing search functionality. As will be explained in the following sections, the iterative approach is also applied during the development of the search extension itself.

5.2.2 Programming Patterns, Core Language and Syntactic Abstractions

As is explained in the WebDSL case study [34], the approach applied during the past development of WebDSL started by studying existing technology in order to find *programming patterns*. The available technology within the domain is explored to find code fragments that occur frequently, being it with slight variations. Commonalities are then captured and put into *code templates* with 'gaps' for variable elements or, if suitable, into a library of common components to be used by the code generator. The idea is to develop a *core language* which is internally used by the compiler. This core language must have enough expressive power such that it supports the needed variabilities in the targeted domain. Without such a core language, variabilities that are adopted in the eventual syntax of the DSL would lead to many versions of the same code templates used by the generator leading to lots of code duplication within the compiler. The core language must be seen as an intermediate language used by the compiler which has the required expressiveness to adopt the domain abstractions, while not exposing lower level code and additional complexity which usually come with expressive code templates.

Syntactic abstractions (also known as *syntactic sugar*) are then introduced to form the actual DSL. These abstractions will be *desugared* in the compiler to the core language. A DSL program is thus being transformed into its core language equivalent. The core language representation can then be used to efficiently generate code in the target language (which is Java in case of WebDSL).

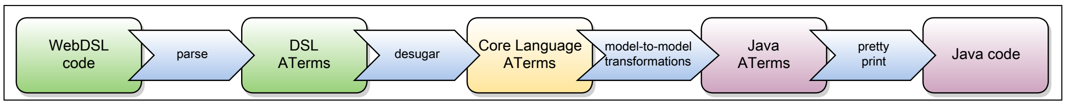


Figure 5.1: Basic flow of the WebDSL compiler, generating java code from DSL code

A basic overview of the compilation process of WebDSL is shown in figure 5.1. A WebDSL program is taken as input. This code gets parsed by a parser generated from the WebDSL syntax definition in SDF. When parsed, the program is represented as abstract syntax tree in *annotated term* (ATerm) format [32]. Using the Stratego/XT program transformation language and toolset [2], the tree of ATerms is then traversed applying various *rewrite rules*. Rewrite rules transform one ATerm into another. By applying these rules to a whole tree, a program can be transformed into another model, which is what happens at the desugar stage: the WebDSL ATerms are transformed into ATerms complying to the core language. Now, other rewrite rules take care of the core language ATerms, transforming them to the desired Java code blocks (Stratego/XT supports the use of concrete syntax in its rewrite rules, which are automatically transformed into their ATerm equivalent). The resulting java classes (represented by the Java ATerms) are then *pretty printed* into plain Java files ready for compilation by a Java compiler.

5.3 A Similar Approach for this DSL Extension

This section discusses the development process we applied for the search DSL extension. This approach is similar to that from past WebDSL development. The set of features to become part of the language were obtained by studying the solution space for programming patterns and common practices. The language is directly put into practice by development of a web application that heavily relies on search features, broadening our perspective of features to be supported and their priorities. Implementation of the language relies on using core languages on which we later put syntactic abstractions (the interface of the language).

With respect to core language development, a distinction is made between the configuration aspects of site search and the specification of constraints, retrieval and presentation of search results.

5.3.1 Imperative Searching

Having studied the problem and solution space of internal site search, the semantics of the domain can be grouped into roughly 4 types. The first type covers configuration aspects, which deal with the selection of searchable data and the transformation of this data into searchable documents in a search index. The second type of semantics deal with the specification of search constraints for restricting the set of data being searched. Then, the third type of semantics relate to the actual retrieval of data. These include the projection of results (i.e., in what representation should the searched data be retrieved), and the retrieval of data *about* the search itself. And finally, there are semantics dealing with the user interface in which search constraints can be specified and search results are presented.

Concerning the latter, WebDSL's template and action language with built-in building blocks for forms, input controls and the rendering of data should fulfill these needs. The data to be presented and facilities used within such user interfaces will be retrieved using aspects from the second and third type of semantics. There is no need to add new linguistic abstractions for the presentation of search user interface elements.

Language features are to be introduced for the other types of semantics. Instead of inventing new core languages, we tried to reuse existing components from the WebDSL language, when suitable. We found WebDSL's *native Java interface* to be a good candidate to start development of the core language for the semantics dealing with searching and retrieval. This language feature adds support to embed Java code within WebDSL application code. This can be useful to interface with existing Java libraries or classes that offer functionality which is not (yet) offered by WebDSL. We will use this language feature to develop an interface of classes and methods, to be used for the implementation of the search/retrieval semantics of site search. By developing an efficient and expressive library of classes and methods, its interface will become the core language on which we can later put syntactic sugar. With Java being an imperative programming language, it will match the stateful characterics of search: the set of search constraints within a search session, or more generally a browse session, should be seen as the state of that session. This state might change incrementally by user actions such as changing the query terms, selecting a category from presented facets or changing the sorting of currently viewed collection.

Another advantage of using native Java is a gain in the implementation cycle for each newly added abstraction. New features become usable by adding Java classes and/or methods to the search library. In order to expose this functionality in WebDSL app code, we only needed to add the signature of the relevant class or method to the native Java definitions. WebDSL uses this to support type checking on embedded Java code. By using Java directly from within WebDSL code, we could create the implementation of the DSL's features, without the need to extend the compiler with syntax definitions and rewrite rules beforehand, which we usually would do when adding new language features. Also, there is no need to rebuild the WebDSL compiler, which is mostly a time consuming job.

5.3.2 Declarative Configuration

Regarding the configurational semantics, the applicability of WebDSL's native Java interface is limited and in fact not suitable. The implementation framework, Hibernate Search, offers two ways for definition of the mapping between POJ objects and index documents (we will refer to this as the *search mapping*). One way is by adding annotations to the Java classes, fields and/or properties. Various types of annotations and parameters make it possible to specify which data should become searchable and how the data should be normalized for searching. As WebDSL abstracts away from the Java classes generated from entity definitions, its native Java interface does not support the use of Java annotations. Besides using these, Hibernate Search also provides a programmatic API to specify the search mapping. However, specification of search configuration is pre-eminently declarative and we would not expose an imperative language in the eventual search DSL.

Instead, we choose to reuse another existing WebDSL language component, namely *property annotations*. These annotations are bound to entity properties and are used to customize a property, for example to assign a default value on initialization of an entity or to treat a property as unique identifier (which are used in URLs) or name to be displayed when the entity is viewed. Property annotations seem to be an ideal place to mark data as being searchable. This is because data units in WebDSL are expressed as entities, which will therefore be the primary units to be retrieved at search time. A document within the search index will therefore represent a WebDSL entity. The data within an entity is represented by entity properties, which in turn can be used as search constraints, i.e. entity properties are the input for search fields in a document. As we will learn, annotations may become large when more options become available for setting up the searchable data, degrading the readability of entity definitions. This makes them less attractive for use as the eventual 'syntactic sugar' for search configuration. However, they are a reusable component already supported by the compiler, suitable for use as core language for searchable data specfication. Table 5.2 summarizes the core languages used for the various concerns of internal site search.

Concern	**Core language**
Searchable data configuration	*Entity property annotations*
Searching/Retrieval	*WebDSL Native Java + API to be developed*
User interface and presentation	*Existing WebDSL template/action language*

Table 5.2: Core languages

5.3.3 WebLib: A Demo Application Built Side-by-side

A demo application, named *WebLib*, was developed to test newly added abstractions during the development of the core languages. WebLib is a simple digital library in which we integrate site search features. Working with a demo application allows quick feedback on the interface of the library. Furthermore, the demo application is a great tool to acquire

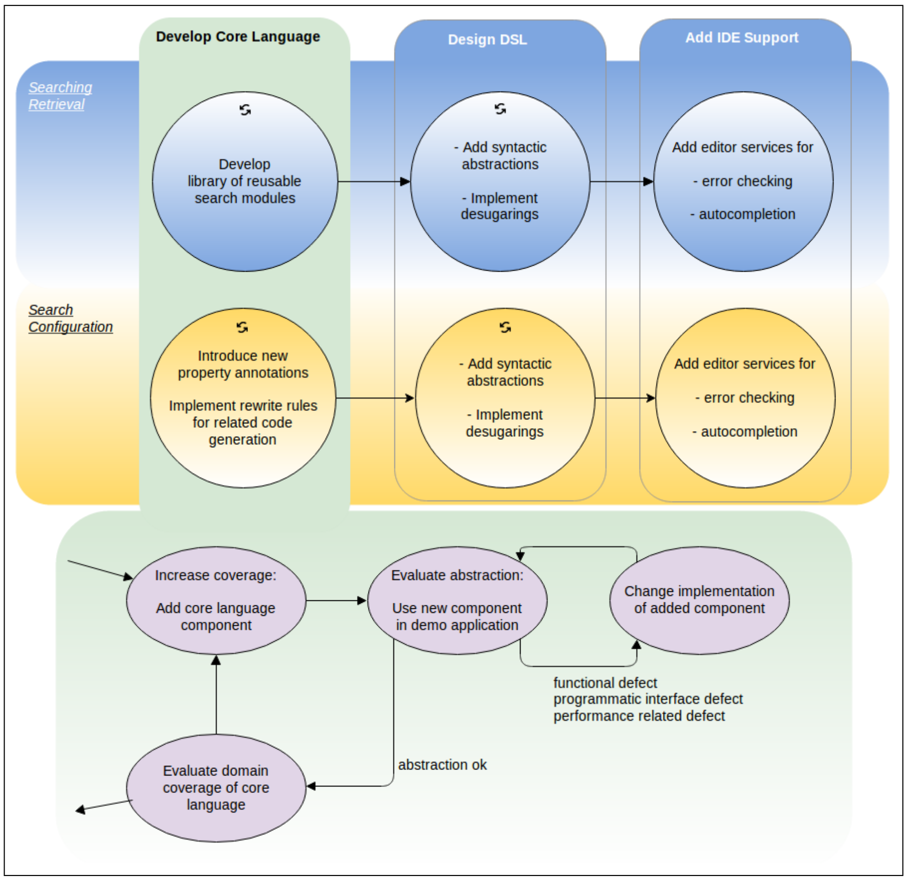

Develop Core Language
Design DSL
Add IDE Support
Searching Retrieval
Search Configuration
Develop library of reusable search modules
- Add syntactic abstractions
- Implement desugarings
Add editor services for
- error checking
- autocompletion
Introduce new property annotations
Implement rewrite rules for related code generation
- Add syntactic abstractions
- Implement desugarings
Add editor services for
- error checking
- autocompletion
Increase coverage:
Add core language component
Evaluate abstraction:
Use new component in demo application
Change implementation of added component
functional defect
programmatic interface defect
performance related defect
abstraction ok
Evaluate domain coverage of core language

Figure 5.3: Iterative process applied during (core) language development

insight in the coverage of the domain, because it partly reflects typical needs regarding the targeted domain. This will help to prioritize future feature(s) to be implemented.

Applying this pattern in an iterative fashion, the interface of our core language library (i.e. the framework of classes, methods and their parameters to be used as core language for searching/retrieval) evolves over time. Changes to the interface and implementation of this library may be required during later development cycles in order to better match domain concepts and relations between them, or to improve conciseness of the library by minimizing the number of method calls and arguments needed to perform some task. Similarly, newly added features for searchable data configuration are evaluated using WebLib. Figure 5.3 shows an overview of the development process, with core language development magnified.

Data Model

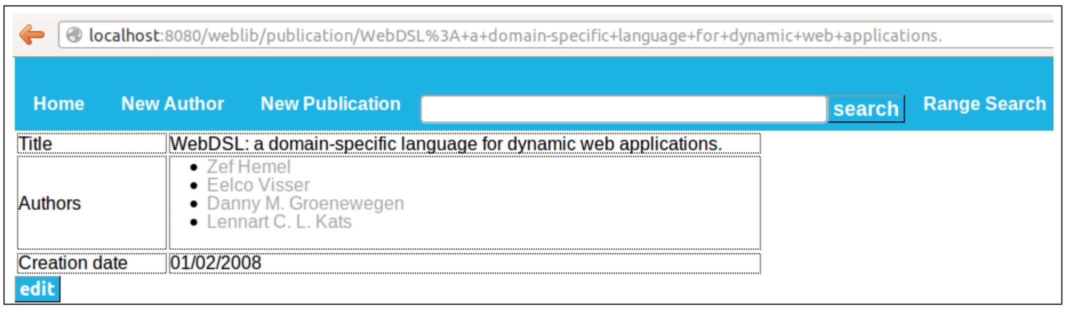
The data model of WebLib is fairly simple. It mainly consists of two entities: `Publication` and `Author`. A publication may have one or more authors, and an author may have zero or more publications. Besides this relational data, both entities own relevant data properties.

These include a title, description, release date and number of citations for the Publication entity and a name for the Author entity. While, of course, the number of citations might be derived by introducing a '*cites*'-relation between Publication entities, we choose to model it as 'dumb' data property for the sake of simplicity. Listing 5.4 shows the WebDSL app-code of this data model.

```
entity Publication {
  title        :: String   ( id, name )
  authors      -> Set<Author>
  description  :: WikiText
  releaseDate  :: Date
  nOfCitations :: Int
}

entity Author {
  name         :: String (id, name)
  publications -> Set<Publication> (inverse = Publication.authors)
}
```

Listing 5.4: WebLib: data model

An initial data set was created by importing publications and authors from a snapshot of DBLP XML records, yielding about 1.3M publications and 0.9M authors in WebLib. Not all modeled data was available in the provided DBLP XML records, we therefore generated random data for release date, number of citations and we gave some publications a description. A basic interface allowed us to view, add and edit publication and author data. As search features were added during the development of the core languages, WebLib was extended to adopt these features in order to evaluate the core language. An impression of WebLib is shown in Figure 5.5.


localhost:8080/weblib/publication/WebDSL%3A+a+domain-specific+language+for+dynamic+web+applications.

Title	WebDSL: a domain-specific language for dynamic web applications.
Authors	• Zef Hemel • Eelco Visser • Danny M. Groenewegen • Lennart C. L. Kats
Creation date	01/02/2008

Figure 5.5: WebLib: publication view

The next two sections give an overview of the features that were implemented during the core language development iterations. In Section 5.4, we discuss implemented features concerning searching and retrieval. Section 5.5 discusses features that relate to configuration aspects of search. The order in which the features are discussed within each section reflect the order of the implementation iterations.

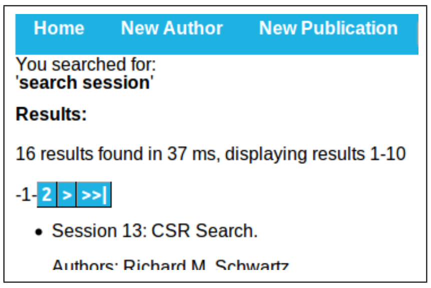
5.4 Search and Retrieval: Introducing the Searcher Type

As discussed in Section 2.4.2, WebDSL already offered limited search features. Only string compatible properties were supported for search and searching was done using one of the three search methods by providing a query and optionally a result starting offset and limit on the result size. These methods return a list of entities matching the query, but did not allow to obtain other information such as the number of results a search query matches. This information is needed in order to provide pagination elements to browse over result pages. It is also not uncommon to display the execution time of a search. Also, when more types of constraints will become available in the eventual search language, these constraints are likely to be presented to the user during browsing. Together with the fact that multiple refinements may be performed after an initial search query, having an object that keeps track of all constraints and provides access to data about a search would be ideal.

It is therefore one of the first additions we introduced in WebDSL: the notion of a *searcher* type. Each entity being searchable in WebDSL gets accompanied with a searcher specific to that entity. That is, a searcher for entity `X` will have a list of instances of `X` as results and holds information relevant for search associated with entity `X` (for example the configured search fields). Generic search functionality, not bound to entity specifics, was later moved to a super class from which all entity searchers inherit. The searcher class was optimized to allow method chaining by returning the searcher instance itself on setter-methods. By using the searcher type in WebLib (through native Java), we were able to show information like the search execution time and pagination controls on the search result page (see Listing 5.6 and Figure 5.7).

```
define template searchResults(query : String, page : Int){
  var searcher := PublicationSearcher().query( query ).setOffset( ( page-1 ) * 10 ).setLimit( 10 );
  var results  := searcher.results();
  var size     := searcher.count();
  var exectime := searcher.searchTime();
}
```

Listing 5.6: WebLib: Using a searcher instance to perform search



You searched for:
'search session'

Results:

16 results found in 37 ms, displaying results 1-10

- Session 13: CSR Search.

Authors: Richard M. Schwartz

Figure 5.7: WebLib: result page with pagination and data about the search

5.4.1 Serializability

One purpose of a searcher is to keep track of the search constraints, without the need to reconstruct the searcher in the app-code for each page, template, action or function. Normally, a web developer should come up with some encoding in order to share the search constraints between page requests. Now that we introduced the searcher abstraction, it was important to have support for using a searcher as page argument, without requiring a WebDSL user to encode the constraints: he should just be able to use a searcher as page/template argument like any other data element. There was no support yet for passing native Java objects as page arguments, because it requires an interface to encode and decode the object from/to an URI-compatible string representation. For this reason we created such an interface. It requires a class to implement 2 methods: `toParamMap` and `fromParamMap`, which are used to serialize and deserialize an instance of that class respectively. These method should return/accept a map of key-value pairs (both Strings) making it able to reconstruct an instance of the implementing class. WebDSL takes care of proper transformation between these maps and URI-strings. The Searcher class was extended with these methods which enables use of searcher instances in template and page parameters.

5.4.2 Field Selection and Query Types

At this stage all search queries were processed by Lucene's query parser, producing a Lucene query from a given string. It searched *all* fields of an entity by default using a text query. Field selection and query type could be adapted according to the Lucene query syntax. On the one hand, this allowed the developer of a WebDSL application to modify a user entered query in such a way that it would be parsed as the desired query type on possibly a subset of search fields, while it also exposed this ability to the users of the web application. The downside is that queries containing a syntax error would not be processed. Ordinary users of a web application, unaware of this syntax, could accidentally break the search process. Other users being aware of this syntax could abuse this feature to perform unwanted searches (for example resource intensive fuzzy searches).

To deal with this problem, we extended the searcher class to have more flexibility in the specification of constraints, without the need to use Lucene's query syntax. We first added methods to set the targeted search fields. We later also added the ability to boost fields on query time. Using a float number (default 1.0), fields can be assigned a weight to prioritize their importance during result ranking. In WebLib, we boosted the title field to gain more weight compared to the description (Listing 5.8).

```
var searcher := PublicationSearcher()
                .fields(["title","description"])
                .boost("title", 10.0)
                .query( query )
                .setOffset( ( page-1 ) * 10 )
                .setLimit( 10 );
```

Listing 5.8: WebLib: Field selection and boosting

We also added control to disallow Lucene's query syntax (default: allowed). However, if the query syntax is disallowed, we (as developer of a web application) would be unable to differ from the default text query type, since we cannot use the Lucene syntax. For that to work, we needed to add new query methods to the searcher interface supporting other types of queries like range and phrase queries. We also added a static method that escapes Lucene syntax symbols such that they are treated as ordinary characters as part of the query. Furthermore, it should be possible to combine multiple queries. For example, we might want to search publications in WebLib matching some keyword (query 1) within a specific date range (query 2) from a specific author (query 3). As will be shown in Chapter 6, we extended the searcher class to support this by having additional methods for grouping and combining queries into boolean queries.

5.4.3 Faceting

As we developed our search extension, a newer version of Hibernate Search was released that added support for faceting. Faceting could be performed on any search field in the index, returning the top *n* terms of a search field that appear in the result set of the current search. By selecting a facet, existing search constraints are expanded with an additional facet constraint. Unfortunately, it turned out that faceting did not work well on fields holding multiple values[1] [2] (which is the case for collection properties, like `Publication.authors` in WebLib). This required a new implementation of Hibernate Search' faceting in order to fix these issues. We therefore decided to partially move to another library for faceting which works on Lucene indexes: Bobo Browse[3]. This library had no issues handling fields with multiple values. We kept range faceting performed by Hibernate Search, because of its integrated support for date and numeric types (single valued fields by definition).

5.4.4 Suggestion Facilities

Suggestion services like spell checking and serving type-ahead suggestions will help a user during his search for information. Although Hibernate Search did not offer suggestional functionality, Lucene provided a contributed spell check module that can take an existing search index or plain text file as input for its suggestions. To make this functionality available in WebDSL, we extended our search library with a `SearchSuggester` class that provides methods for retrieving spell suggestions. Providing one or more search fields and the number of suggestions to be retrieved, the search suggester uses the associated suggestion-index to find the targeted suggestions. Type-ahead support was introduced later as there was no such implementation available for Lucene.

Type-ahead Suggestions Implementation

In order to support type-ahead suggestions, we base our implementation on the spell check class as provided with the Lucene library. The `AutoCompleter`-class takes the terms from

[1] `https://hibernate.onjira.com/browse/HSEARCH-726`

[2] `https://hibernate.onjira.com/browse/HSEARCH-776`

[3] `https://github.com/javasoze/bobo`

a source index as input for suggestion index creation, similar to the spell check class. It creates a document for each term that appears in a given search field of a source index (a single term may contain whitespaces, depending on the analysis bound to the field). The original term is stored in a search field (not tokenized, letter casing preserved), such that it can be retrieved at the actual moment of suggesting. In case the original term contains whitespaces, the term is splitted into tokens. This enables multi-token terms to be suggested when only a single token within the term matches the prefix entered by a user. In order to search the suggestion index, prefix-fields are created and added to each document. It does so by constructing prefixes of length 1 to 10 (controllable with a constant `MAX_PREFIX_LENGTH`) for each token in the original term. The prefixes are indexed in lower case. Lastly, it adds a field that stores the frequency of the term in the original source index. The frequency is used as simple heuristic for popularity. By sorting the suggestions on their frequency in the source index, more popular terms appear higher in the suggestions list. When requesting suggestions (see Figure 5.9) for an input *i* of length *l*, the `AutoCompleter` class searches the index using the prefix-fields. It does so by searching the prefixes of length *l*, and in addition prefixes of length *l-1* and *l-2* (in case *l* >2). A suggestion document matches whenever at least 1 prefix matches. Completion suggestions are thus still provided in case the user makes one or two mistypings.

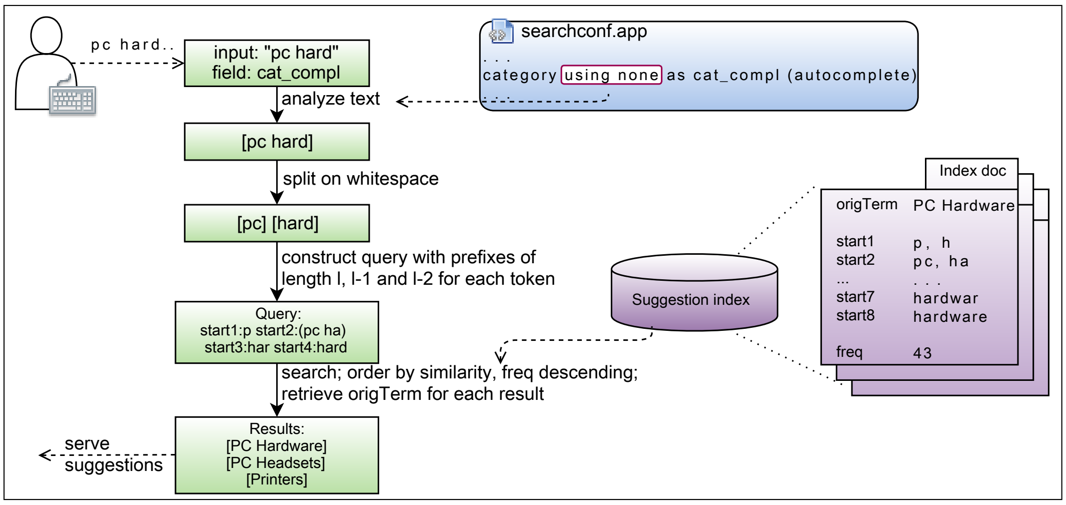


Figure 5.9: Constructing type-ahead suggestions

5.4.5 Sorting and Highlighted Result Summaries

In later stages, we added support for controlling the order of results. By default, results are sorted by relevance using Lucene's scoring algorithm, which is based on the Vector Space Model [26]. The searcher interface was extended with methods to sort on any search field available. A mapping between the search fields and their type of data is required in order to sort the results with Lucene. The WebDSL compiler was extended to include this information to the entity specific searcher classes that are generated. This way, we can sort the publication in WebLib on number of citations and creation date.

We also tackled the presentation of search results. As mentioned in the solution domain chapter, Lucene includes a contributed module for extracting fragments from a text (tailored to a given query) and highlighting of the matched terms. As a searcher object is being used to keep track of search constraints, we added highlight methods that return a selection of fragments with the constraint values highlighted. It surrounds the matched terms with (HTML) tags that can be provided as method arguments. However, the fragments that are returned will contain these (HTML) tags, which in turn will get filtered (encoded) by WebDSL if outputted as ordinary text. This problem can be worked around by outputting the text as-is and by filtering the original text prior to highlighting. Since we don't want to expose this flow of actions to the developer, we later added a built-in WebDSL template that can be wrapped around other elements such that it highlights everything that is put between the brackets, taking a searcher and search field (to base highlighting on) as arguments (discussed in section 6.5.2).

5.4.6 Search Filters and Namespace Abstraction

Apache Lucene (and Hibernate Search) adopt the notion of *filters*. A filter is actually a search result encoded into a bit-set, one bit for each document. By caching the filters, the result set does not need to be calculated over and over again. Filters are efficiently applied as bitmask and do not affect the scoring of documents. This is useful when constraints not contributing to a document's relevance are to be applied repeatedly. The notion of filters was also added to the search DSL.

These filters are also used internally for another abstraction we introduced: *search namespaces*. With this feature, the value of an entity property can act as namespace identifier for search. At search time, the searcher interface allows to set the namespace such that it only targets entities that hold a specified value in the namespace property. For example, if we extend the `Publication` entity in WebLib to have a property 'language', acting as namespace identifier, we would be able to search for publications in a particular language using the namespace abstraction. The namespace abstraction also manages the suggestion indexes such that spell and type-ahead suggestions are returned only for the targeted namespace. We did not use this feature in WebLib, as it was later added to fulfill the requirements of *Reposearch*, which required suggestion retrieval to be scoped to a single namespace (see Section 7.1).

However, our first implementation of search namespaces needed to be replaced with a different implementation. We first created multiple indexes using *sharding* as provided by the Hibernate Search library, one for each search namespace. It turned out that the number of shards could not be adapted at runtime, and therefore there was a hard limit on the number of namespaces. Unfortunately, we could not just set this number to a very high number. By profiling a deployed WebDSL application, we saw that the number of threads, open file descriptors and memory usage grew with the number of unused, but configured shards. Furthermore, our shift to Bobo-Browse for faceting forced us to reconsider the implementation of search namespaces using shards. It turned out that code changes to Bobo-Browse were needed in order to work with sharding in Hibernate Search. For this reason, we reimplemented search namespaces to use a single search index with filtering at search time. If used, suggestional indexes are still created for each namespace. This happens dynamically, so there is no preconfigured limit on the number of search namespaces.

5.5 Configuration: Support All Property Types

The limited existing implementation of search had shortcomings with respect to the data types that were supported to become searchable: only String compatible types were allowed. At the beginning of the DSL extension process, existing web applications built with WebDSL already showed the need for search support on other types. One example is search in YellowGrass, a tag-based issue tracker build with WebDSL. Issues were only searched on their title and description (both being string compatible properties for the entity `Issue`), but not on comments. Comments are modeled as collection property for `Issue`. So in order to search issues on texts in comments, the textual properties of entity `Comment` should be searchable in the scope of the targeted type `Issue`. In other words: the search fields of the *embedded* property type (`Comment`) should be embedded in the indexed document of the *embedding* entity `Issue`.

5.5.1 Embedded Property Types

Similarly for WebLib, we wanted to be able to search for `Publications` using the name of the author. We therefore extended the WebDSL compiler to allow embedded property types to become searchable (i.e. all properties of type reference or composition, see Section 2.1). An optional parameter `depth` was added to the property annotation to control the level of embedding. With depth set to 1 (the default value used when `depth` is not specified), it only indexes searchable, non-embedded typed properties of the embedded entity type. Increasing the depth one more level will also include the embedded typed (searchable) properties of the embedded entity type, and so on.

5.5.2 Text Analysis and Named Search Fields

Adding support for embedded search fields was not sufficient to cover index configuration related functionality. At this stage, all textual data was analyzed using the default Lucene analysis, optimized for English. Other recall/precision improving techniques, such as the stemming of tokens, could not be specified yet. Since SOLR provides a powerful framework of analyzer components, being adopted in Hibernate Search, we also adopted this framework into the search DSL. The specification of an analyzer is fully declarative. An analyzer consists of at least one tokenizer and optionally one or more token and/or character filters that comply with SOLR interface. For this part of the search language, we designed new language constructs (not elaborating on a core language). The analyzer definitions are defined at the top level, similar to entity definitions. The searchable annotations were extended with an optional `analyzer` argument to differ from standard analysis. The analyzer definition syntax was later extended to support performing different analysis at query and indexing-time for a single search field. Also, some analyzer components may use file based resources like synonym and stop-word files. We dedicated a project directory for including these files (`project-path/search/analyzerfiles`).

Sometimes it may be required to index the same data multiple times using different analyzers for different purposes. In WebLib, we wanted to use publication titles both for searching and type-ahead suggestions. The default analyzer would suffice for searching, but not for providing suggestions because we wanted the whole title, not the single tokens within, to act as suggestion candidate. Where we previously allowed only one searchable annotation on an entity property, the search field was identified by the name of the entity property. We extended the WebDSL compiler to allow multiple searchable annotations on a single entity property. To distinguish one search field from another, an argument `name` was added to searchable annotation. If this argument is not used, the name of the entity property will act as name of the search field.

5.5.3 Numeric and Date Properties

Other data types in WebDSL that need special treatment are Date/DateTime/Time and Integer/Float. These types need to be preprocessed in such a way that they are useful for search. We extended the compiler to instruct Hibernate Search to preprocess the DateTime types by transforming them to a String representation that can be used for sorting and range queries. Numeric data is, by default, indexed as numeric field for more efficient sorting and execution of range queries.

5.5.4 Management of Search Indexes

Changes, additions and deletions of data are automatically propagated to the search index by Hibernate Search. Changes to the search configuration, like changes to search fields or analyzers, may require the index to be rebuild. For this reason, we added a module `IndexManager` for performing such tasks. It has static methods for rebuilding search indexes for a single, multiple or all entities. This includes methods for rebuilding suggestion indexes where the entity, field and namespace can be specified. Renewal of faceting index readers (used by Bobo-Browse) and optimization of search indexes can also be performed. Except for rebuilding the search indexes, WebDSL manages the renewal of suggestion indexes, index optimization and renewal of facet index readers automatically. Usage of the Index-Manager would typically be used to force early renewal of suggestion indexes and facet readers, which are normally renewed every 12 hours and every 15 minutes respectively.

5.6 Towards a DSL for Internal Site Search

As the domain coverage of the search-related functionality increased during the development iterations, we came to a point where most important search features were implemented and usable through core language constructs. Next step in the development process (Figure 5.3) was the design of the DSL syntax. At the configurational aspects of search, we introduced new top level definitions for specification of searchable data, called search mappings. Search mappings describe the selection and mapping of searchable data in a concise, declarative way. Property annotations, the core language equivalent of search mappings, would quickly become large and fragmented over the entity definitions. We designed other language constructs for the initialization, modification and retrieval of search constraints, results and search meta-data. The WebDSL compiler is extended to transform the ATerms that flow out of the newly added syntax. During the *desugar* stage of the compiler, the ATerms are transformed into core language equivalents (that is, native Java for search and retrieval; property annotations for search configuration).

When the compiler finishes desugaring, the desugared program (represented by a tree of ATerms) is further transformed into a tree representation of the eventual Java program to be compiled (Figure 5.1). Prior to adding syntactic sugar, the compiler was extended with transformation rules that weave search aspects into the Java code and configuration files to be generated. For example, for each mapping of an entity property to a search field, the compiler will add an Hibernate Search annotation (`@Field`) to the Java class field which gets generated for the associated entity property. Analyzer definitions are generated once and are referred by other generated Hibernate Search field annotations. For each searchable entity, a `Searcher` class will be generated, which is an extension of the generic `AbstractSearcher` providing search functions and data specific to the entity (such as the default search fields and a method for the retrieval of results having the appropriate return type). Just to give you a grasp of how these transformations look like, simple examples of Stratego code responsible for desugaring and code generation are listed in Listing 5.10 and 5.11 respectively. In the following chapter we will dive into the concrete syntax of the search language.

```
field-to-boost:
   ( QuerySearchField(fld, Some(QueryBoost(e_boost))), e_done ) -> exp|[e_done.boost(e_fld, e_boost)]|
   with e_fld := <field-to-string> fld
```

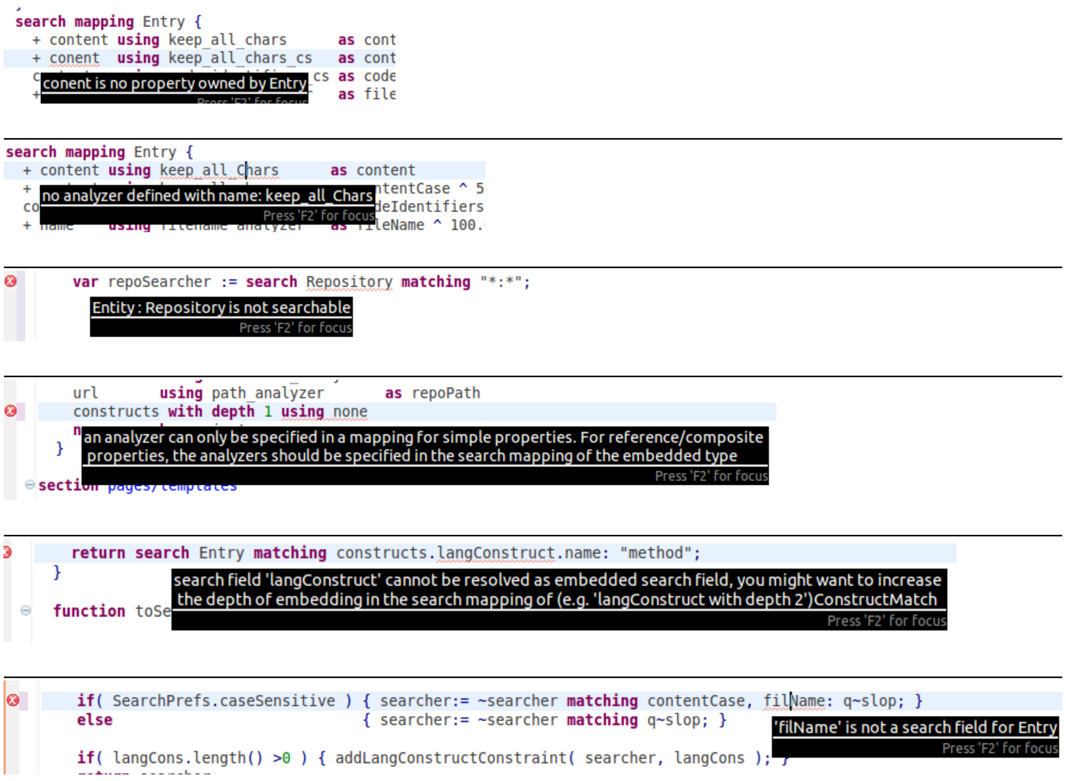
Listing 5.10: Stratego rule invoked during desugaring: it transforms a field-boost as part of a search expression into a core language construct (WebDSL native Java)

```
to-numeric-field-anno:
   SearchableAnno(sa-args) -> anno|[ @org.hibernate.search.annotations.NumericField(forField = "~field-
     name") ]|
   where <fetch(?Numeric-Argument())> sa-args
       ; <fetch(?SA-Argument("name", field-name))> sa-args
```

Listing 5.11: Code generation by transforming a core language ATerm (left hand side) to a Java ATerm expressed in concrete syntax (right hand side)

5.6.1 Editor Services

The Spoofax language workbench allows to add editor-services for reference resolving, code folding, code completion and error checking. You get code highlighting and code folding for free as it can be derived from the SDF syntax definition (both can be customized). Error checking and code completion were mainly implemented during the last stages of the search DSL extension, after the design of the search DSL syntax. We have added various checks on search configuration and expressions for searching. The errors reported adopt language and domain concepts. Figure 5.12 show examples of these when using WebDSL in the integrated development environment Eclipse. Code completion is currently limited to analyzer definition templates, and completion for names of the SOLR analyzer building blocks.



```
search mapping Entry {
  + content using keep_all_chars      as cont
  + conent  using keep_all_chars_cs   as cont
  c                                cs as code
  +                                   as file
```

conent is no property owned by Entry

```
search mapping Entry {
  + content using keep_all_Chars       as content
  +                                ntentCase ^ 5
  co                               deIdentifiers
  +                                   leName ^ 100.
```

no analyzer defined with name: keep_all_Chars
Press 'F2' for focus

```
    var repoSearcher := search Repository matching "*:*";
```

Entity : Repository is not searchable
Press 'F2' for focus

```
    url        using path_analyzer        as repoPath
    constructs with depth 1 using none
    n
  }
section
```

an analyzer can only be specified in a mapping for simple properties. For reference/composite properties, the analyzers should be specified in the search mapping of the embedded type
Press 'F2' for focus

```
    return search Entry matching constructs.langConstruct.name: "method";
  }

  function toSe
```

search field 'langConstruct' cannot be resolved as embedded search field, you might want to increase the depth of embedding in the search mapping of (e.g. 'langConstruct with depth 2')ConstructMatch
Press 'F2' for focus

```
    if( SearchPrefs.caseSensitive ) { searcher:= ~searcher matching contentCase, filName: q~slop; }
    else                            { searcher:= ~searcher matching q~slop; }

    if( langCons.length() >0 ) { addLangConstructConstraint( searcher, langCons );
```

'filName' is not a search field for Entry
Press 'F2' for focus

Figure 5.12: Examples of errors reported when using Eclipse IDE. Compilation also breaks when these errors occur and will be reported in the compiler output.

Chapter 6

A DSL for Search

6.1 Design of the Language

The concrete syntax of the designed search language can be partitioned into mainly four categories. A section is dedicated to each of these categories. These include:

- specification of searchable data
- definition of analyzers
- specification of search constraints
- retrieval of queried data and meta data

Because the search language is mostly designed around core languages that were already part of the WebDSL language (except for the definition of analyzers), there is a core language equivalent for each syntactic sugar introduced. By adding syntactic sugar, the language becomes more expressive with respect to the site search domain. It is not bound to symbols and structures of a host language (as is the case for internal DSLs), increasing the conciseness and readability of the code.

However, introducing new language constructs just for the sake of 'having syntactic sugar' cannot be considered a good idea in general. Green and Petre [10] explain the cognitive implications related to programming language design by introducing the so called cognitive dimension framework. Adding new language constructs would increase the number of lexemes and structures a programmer needs to remember. Because the appliance of particular (including expert) functionality is very limited, we decided to leave this functionality solely covered by core language constructs, in this case, calls to Java methods. This notation is commonly used in WebDSL applications as method calls and function calls share the same notation. In other cases, adding new language constructs leads to an increase in the number of symbols to express a particular task (a method call is shorter), lowering the *terseness* of the language. Keeping such functionality solely accessible through the core language (method calls) relieves a user of the language from remembering additional, possibly lengthier language constructs. Examples include expressions for retrieval of less common search meta-data, advanced hit highlighting and management and retrieval of lists of constraints (like search filters and facets).

We tried to raise *language consistency* by reusing keywords and structures for similar tasks, such that it becomes easier to guess the notation for different objectives for a novice user. Examples of such include the keyword 'matching' applied at query and facet constraints and retrieval of suggestions; the keyword 'with' for filtering and enabling faceting; the structure '*data* from searcher' to retrieve various data from a searcher; a single notation for ranges in query and facet specification; and the same symbol for boosting search fields at indexing and query time. Furthermore, we avoided requiring specific arrangements of language constructs, which would otherwise increase the cognitive overhead for a developer needing to remember specific arrangements of lexemes (a *hard mental operation*).

Finally, the language is designed to be close to the problem domain, i.e. has a good *closeness of mapping*. The notation adopts domain concepts without noise from the base language, and symbols really describe their purpose (i.e. a high *role-expressiveness*).

6.1.1 Syntax Notation

In remainder of this Chapter, newly added language constructs are presented with an discussion of the semantics, accompanied with code examples. The syntax of these constructs are described using the notation as shown in Figure 6.1. All keywords (terminals) start with a lowercase letter. Non-terminals start with an uppercase letter. The meta-syntax operator symbols (`[`,`]`,`(`,`)`,`{`,`}`,`=`,`|`,`+`,`*`) are printed in bold such that they can be distinguished from equal non-terminal symbols that are part of the syntax to be described.

```
Grammar        = Rule+
Rule           = NonTerminal = Rhs
               | NonTerminal |= Rhs //additional Rhs for NonTerminal defined elsewhere
Rhs            = NonTerminal [TypeConstraint]
               | Terminal
               | Rhs Rhs
               | Rhs Repetition
               | {Rhs Terminal} Repetition //sequence of Rhs's seperated by Terminal
               | [Rhs]                   //optional
               | (Rhs)                   //grouping
               | Rhs | Rhs               //alternation
               | Comment
Repetition     = * //repetition: zero or more
               | + //repetition: one ore more
NonTerminal    = Letter<uppercase> Char* //always starts with uppercase letter
Terminal       = Char<lowercase> Char*
TypeConstraint = < Char+ >
Comment        = // Char*
Letter         = //any of the characters in range a-z and A-Z
Char           = //any single character including letters, numbers and symbols
```

Figure 6.1: Meta-syntax we use to express the syntax of the language. All non-terminals start with uppercase letter; all terminals start with lowercase letter.

6.2 Searchable Data Specification

Entities are the main data units to store content in WebDSL. Therefore, we choose to map WebDSL entities as document unit, and entity properties as search fields. If an entity has at least 1 searchable property, possibly inherited from a super class, all instances of that entity type will get indexed. As we explained earlier, we have chosen *property annotations* as core language for search field specification. By adding the searchable keyword to a property's annotations, a search field will be added to the index document for that entity. If at least one search field is defined for an entity (or in one of its super entities), documents will be created and kept up to date for each instance of an entity type. Related searchers and search methods are automatically generated.

As more search features related to the configuration were added, the searchable property annotations were extended with additional arguments in order to override default configuration for a search field. Taking a search field as base, a search field can:

- hold a single or multiple values (when a property is a collection of elements) for the same document.
- be declared on any built-in type, declared entity type or collection of entities.
- be identified by a name which, if not specified, is the name of the entity property and should be unique in the scope of an entity.
- be declared alongside other search fields for the same property (i.e. support for indexing the same data multiple times)
- optionally be accompanied with an analyzer name, which refers to an analyzer definition that specifies how data should be tokenized/normalized. If no analyzer name is specified, it uses the default analyzer, or none depending on the type of property dealing with, more about this in Section 6.3.2.
- be used as source for spell check and completion suggestions
- be boosted with a float number, to control the importance of search fields in the ranking of search results.
- be marked as default search field. The ability to mark search fields as default ones (overriding the default set being *all* search fields of an entity) allows more compact specifications of search constraints.

Search fields can be referenced by their name which, if not specified, is the name of the mapped entity property. In case such property holds one or more references to other declared entities, this name acts as prefix for the *embedded* search fields. Listing 6.2 shows a version of the data model of WebLib with all properties being searchable. Each indexed `Publication` document will now have the following 5 search fields: `title`, `authors.name`, `description`, `releaseDate` and `nOfCitations`. For the `Author` entity, the search fields are: `name`, `pubs.title`, `pubs.description`, `pubs.releaseDate` and `pubs.nOfCitations`.

In this example, the reason why property `Author.publications` is not mapped as search field for `Publication` is because of the parameter `depth` controlling at which level search

```
entity Publication {
  title        :: String      ( id, name, searchable )
  authors      -> Set<Author> ( searchable( depth=1 ) )
  description  :: WikiText    ( searchable )
  releaseDate  :: Date        ( searchable )
  nOfCitations :: Int         ( searchable )
}

entity Author {
  name         :: String           ( id, name, searchable )
  publications -> Set<Publication> ( inverse = Publication.authors,
                                       searchable( depth=1, name="pubs" )
                                     )
}
```

Listing 6.2: Data model of WebLib with all properties being searchable using property annotations

field embedding should stop. By default, this is set to 1, meaning that it only adds search fields that maps to simple type properties of the associated embedded entity.

While the code in Listing 6.2 might still look clean, adding more search field specifications through annotations impacts the readability of the code. Let's say we want author names to be searchable such that people not knowing the exact spelling of a name are still able to find authors by typing how it sounds like, i.e. using phonetic analysis. Moreover, we would like to show author names as auto complete suggestions in search boxes. Now, the specification of the `name` property would look like this:

```
name :: String ( id, name, searchable( default ),
                 searchable( analyzer=soundex, name=name_soundex ),
                 searchable( analyzer=none, name=name_untokenized,
                             autocomplete ),
               )
```

Listing 6.3: Specification of multiple search fields on a single entity property using property annotations

The size of the property annotations now take the overhand, and indentation is already tweaked to let the code look more 'clean'. However, the larger property annotations will cause the property definitions itself to be more fragmented, making it harder to keep the data model organized. For this reason, we designed new language constructs for configuring search fields. As we are dealing with a mapping between an entity and index document, we identify these constructs with the keywords *search mapping*. A search mapping definition is bound to a WebDSL entity and can be embedded within an entity definition, or at the top level by adding the identifier of the entity associated with this search mapping. This allows a developer to unite multiple search mappings into a separate module (allowing separation of concerns) or a developer can keep the search mapping and entity definition together when he prefers locality of the code. The search mapping for the example `Author` entity would become:

```
search mapping Author {
  + name
    name using none as name_untokenized (autocomplete)
    name using soundex as name_soundex
    publications as pubs with depth 1
}
```

Listing 6.4: Search mapping for entity Author

6.2.1 Searchable Data Specification: Syntax

Figure 6.5 shows the basic syntax for search mapping definitions. A search mapping specification (`SearchMappingSpec`) can be placed on each line of a search mapping. Each `SearchMappingSpec` maps an entity property identified by its name `Pid` to a search field with optional configuration parameters, expressed by `MappingParts`. Additionally, one can precede a `SearchMappingSpec` by a `+`, indicating that this search field is part of the default collection of search fields used for searching. The optional *index* keyword is added solely for cosmetic reasons. It allows to express mappings in some sort of natural form, e.g.: `index title using trigram_analyzer as title_trigram boosted to 10`. Finally, the system can be configured to have search namespaces based on a property value. At query time, a namespace constraint can be added which tells the system to only target data in a specific namespace, such as requesting English spell suggestions in case a language property is configured to act as namespace separator.

```
Eid                   = Id //identifier of entity
Pid                   = Id //identifier of property
SearchMappingEmbedded = search mapping { SearchMappingSpec* }
SearchMapping         = search mapping Eid { SearchMappingSpec* }
SearchMappingSpec     = [+] [index] Pid MappingPart* [;]
                      | namespace by Pid [;]
MappingPart           = as Id
                      | (boosted to|^) Float
                      | using Id
                      | ( {Annotation ,}+ )
                      | for subclass Id
                      | [with] depth Int
Annotation            = spellcheck
                      | autocomplete
```

Figure 6.5: Syntax for search mapping specification

The syntax and semantics of the possible `MappingParts` are organized in table 6.6. This table also shows the property annotation equivalents for each search mapping construct. Some configuration options are exclusively for simple type properties (like String, Text, DateTime etc), others for reference type properties, which are (collections of) defined entity types.

Concrete Syntax	*Semantics*
`as` *`Id`* Equivalent: *searchable*(`name =` *`Id`* ...)	overrides the default search field name, or prefix in case of an embedded search field default: pId (identifier of the property to be indexed)
`+` *`Pid MappingPart`* Equivalent: *searchable*(`default` ...)	Using '+' explicitly mark a search field as default search field. default: If no other search field is explicitly marked as being default, all search fields for an entity are default search fields
`boosted to` *`Float`* `^`*`Float`* Equivalent: *searchable*(`boost =` *`FLOAT`* ...) *searchable*(...)`^`*`Float`*	boosts this field at indexing time to value of *`Float`* default: 1.0
`using` *`Id`* Equivalent: *searchable*(`analyzer =` *`Id`* ...)	specifies which analyzer (*`Id`*) to use for preprocessing. When *`Id`* is `none`, the data is indexed untokenized. default: Default built-in analyzer, or an analyzer marked as default. Temporal/numerical types of data are not analyzed by default.
`( {`*`Annotation`*` ,}+ )` Equivalent: *searchable*(`{`*`Annotation`*` ,}+ ...`)	Search mapping annotations to set additional flags to search fields. Possible values are "spellcheck" and "autocomplete", allowing the search field to be used for spell checking or auto completion.
`for subclass` *`Id`* Equivalent: *searchable*(`subclass =` *`Id`* ...)	specifies which subclass to index default: Defined type (or inner type in case of collection properties) of the data element to be indexed

`depth` *`Int`*
`with depth` *`Int`*
 	
Equivalent:
searchable(`depth =` *`Int`* ...)
	specifies the maximum path length to be traversed for search field embedding
	default: 1

all types	simple types only	reference types only

Table 6.6: Search mapping semantics

6.3 Specification of Analyzers

An analyzer describes how a stream of textual data should be transformed into tokens to be added to or queried against a search index. The search language allows one to specify analyzers and refer to them in specification of search fields, as has been done for the search field `name_soundex` in Listings 6.3 and 6.4. An analyzer definition is a top-level definition, meaning that it can be defined anywhere as long as it is not nested. The SOLR framework of analyzer building blocks is used for specification of analyzers. An analyzer definition consists of zero or more character filters, one tokenizer and zero or more token filters. A collections of these filters and tokenizers are supplied with SOLR and are available for use in WebDSL. This set of tokenizers, character and token filters will cover most requirements for tokenization. To name a few, it supports splitting texts into *n-grams* (building tokens of length *n* from adjacent letters); tokenization and token filtering using regular expressions; expansion of tokens with their synonyms; stemming for a variety of languages (transforming words into their root form); phonetic matching; reversal of tokens; lower-casing and many more[1]. In case it lacks needed functionality, a customized tokenizer or character/token filter can be created by implementing the intended interfaces from SOLR. This tokenizer/character- or token filter can then be referenced by using it's fully qualified class name in an analyzer definition.

6.3.1 Specification of Analyzers: Syntax

```
AnalyzerDef     = [default] analyzer Id { AnalyzerBody }
AnalyzerBody    = AnalyzerBodyDef
                | HybridBodyDef
HybridBodyDef   = index { AnalyzerBodyDef } query { AnalyzerBodyDef }
AnalyzerBodyDef = CharFilter* Tokenizer TokenFilter*
CharFilter      = char filter = {Id .}+ [Args]
Tokenizer       = tokenizer = {Id .}+ [Args]
TokenFilter     = token filter = {Id .}+ [Args]
Args            = ( {Argument ,}* )
Argument        = Id = String
```

Figure 6.7: Syntax for analyzer definition

Figure 6.7 shows the syntax for analyzer definitions. The keyword 'analyzer' is followed by the name of the analyzer and its body containing the references to the analyzer components: character filters (optional), a tokenizer and token filters (optional). Each analyzer component is referenced by its class name if the component is provided with WebDSL, i.e. being part of the SOLR analyzer package (`org.apache.solr.analysis`). For simplicity the suffix '`Factory`', which all of the provided analyzer component class names share, can be omitted in analyzer definitions.

If the text to be indexed should be analyzed differently than queries entered by the user, or more general: constraint values, an analyzer may contain a definition for analysis at

[1] `http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters`

indexing time and one for analysis at query time. Lastly, an analyzer can be marked as being the default analyzer for search fields. Search field specifications that lack an analyzer specification will use the default analyzer.

Listing 6.8 shows the definitions of the analyzers named `soundex` and `spell_check`. The `Soundex` analyzer tokenizes the text using *StandardTokenizer*, which uses the *Unicode Text Segmentation* algorithm as is specified in *Unicode Standard Annex #29* [2] for tokenization. Tokens then get normalized by *StandardFilter* and *LowerCaseFilter*. The normalized tokens then get transformed into their phonetic form as is done by the *PhoneticFilter*, configured to use the *RefinedSoundex* algorithm.

```
analyzer soundex{
    tokenizer = StandardTokenizer
    token filter = StandardFilter
    token filter = LowerCaseFilter
    token filter = PhoneticFilter(encoder = "RefinedSoundex")
}

analyzer spell_check{
  index {
    //put shingles in index for length 1, 2 and 3 (suggestions will return =< 3 terms)
    tokenizer = StandardTokenizer
    token filter = StandardFilter
    token filter = LowerCaseFilter
    token filter = ShingleFilter(minShingleSize="2", maxShingleSize="3",
                                 outputUnigrams="true", tokenSeparator=" ")
  } query {
    //interpret user query as a single token (i.e. one shingle)
    tokenizer = KeywordTokenizer
    token filter = StandardFilter
    token filter = LowerCaseFilter
  }
}
```

Listing 6.8: Analyzer definitions for a soundex (phonetic matching) and spell check analyzer

The other analyzer `spell_check` is special in the sense that it treats data at indexing time differently than at query time. It is designed to support meaningful spell checking suggestions on multi-term queries. Tokens extracted from searchable properties are not only indexed as single terms (unigrams), but also as shingles of 2 and 3 tokens. At query time, the input is treated as one single term. The spellchecker component will try to find similar terms in the indexed collection of uni-, bi- and trigram terms. Using the shingle filter, the spell checker can return meaningful phrase suggestions, because it compares the input to single and multi-token terms.

6.3.2 Default analyzer

In search mapping specifications, the analyzer argument is optional. If no analyzer is referenced, the default analyzer is used for properties of textual types. This analyzer is equal to Lucene's `StandardAnalyzer`, which tokenizes text based on the *Unicode Text Segmentation*

[2] `http://www.unicode.org/reports/tr29`

algorithm. It normalizes tokens by removing apostrophes and dots followed by lowercasing the tokens and removal of English stop words.

While this analyzer suffices for most textual data containing natural language, it might not be the optimal analyzer for default use. One can override the built-in default analyzer with any other analyzer definition by inserting the `default` keyword in front of an analyzer definition. Listing 6.9 shows an analyzer to be applied as default one for a Dutch web application.

```
default analyzer stemmed_dutch{
    tokenizer = StandardTokenizer
    token filter = StandardFilter
    token filter = LowerCaseFilter
    token filter = SnowballPorterFilter(language="Dutch")
}
```

Listing 6.9: A default analyzer for a Dutch web application

Analysis of Numerical and Temporal Data

The non-textual property types `Float`, `Int`, `Date`, `DateTime` and `Time` are, by default, analyzed differently. Float and Integer type properties are indexed as numeric fields, which are a special type of search fields in the index. It allows sorting and searching ranges to be performed more efficiently on these fields. Properties of temporal type are first transformed into a String representation of the form `yyyyMMdd` for `Date`; `yyyyMMddHHmm` for `DateTime` and `Time`. These values are then indexed untokenized (i.e. without any analysis). Still, an analyzer can be applied to a temporal property type, for example to only index the year from a `Date` property.

6.3.3 Predefined Analyzers Bundled with WebDSL

A selection of illustrative analyzers are supplied with WebDSL. These include the default analyzer coming in two other flavors: one without stop word filter and one for use with a custom stop word filter. There is also an analyzer for expanding terms with synonyms; one that splits words into trigrams (tokens of length 3); one for matching phonetically similar terms; and an analyzer that performs stemming. The analyzer definitions reside in an app-file included in any new WebDSL project and can be adapted.

6.4 Specification of Constraints

Search and retrieval is done using the new WebDSL type *Searcher*, which is generated for each searchable entity and its child classes. A searcher can be constructed and interacted with using WebDSL's native Java interface, as has been done during the development of the search extension. Native Java is chosen to be the core language on which we put syntactic sugar by designing a searcher interaction language for construction of/interaction with searcher instances. The language is designed keeping in mind that a browse or search session may be dispersed over multiple page requests involving multiple user actions. Furthermore, constraints may be added or changed on various places in WebDSL, like functions, pages, actions and templates. While most common functionality is covered by the searcher interaction language, some functionality is exclusively accessible through Searcher's methods. These include methods for management and retrieval of declared constraints values, debug functionality and some expert features. The full list of methods available for the Searcher type is included at the end of this chapter in Table 6.29.

6.4.1 Initialization of and Reference to a Searcher

The language allows to specify constraints on initialization of a searcher or on an already declared searcher captured in a variable (Table 6.10). Initialization of a searcher will set the first constraint, namely the class of entities to be targeted. Additional constraints can be specified directly at initialization or later at other places using a reference of a searcher.

Syntax:

```
SearcherInit    = search Id SearcherPart*
SearcherRefMod = ~ Exp<Searcher> SearcherPart+
SearcherPart    = ConstraintFilter
                | ConstraintDef
                | Offset
                | MaxResults
                | OrderBy
                | FacetDef
                | SearcherAttributes
                | NamespaceConstraint
```

Search language example	*Embedded Java equivalent (core language)*
`var s := search Entry;`	`var s := EntrySearcher();`
`var s := search Entry matching "static query";`	`var s := EntrySearcher().query("static query");`
`var s := ~getBasicSearcher() matching userQuery;`	`var s := getBasicSearcher().query(userQuery);`
`~s limit 20 offset 20;`	`s.setLimit(20).setOffset(20);`

Table 6.10: Syntax: Searcher interaction language

6.4.2 Query Specification

The most common constraints are queries. A set of query types are currently supported and query constraints can be grouped and combined into boolean queries. The `matching` keyword indicates a query constraint and is followed by one or more match groups. A match group optionally starts with an enumeration of search fields followed by a colon, and always consists of a query of supported type. Match groups and queries within match groups can also be grouped using parentheses to allow searching multiple queries for the same search fields and for combining match groups using boolean operators. Table 6.11 shows the syntax definition.

Syntax:

```
ConstraintDef   = matching MatchGroup+
MatchGroup      = [ {QueryField ,}+ :] Constraint
                | [+|-]( MatchGroup )
QueryField      = Field [ Boost ]
Field           = Id
                | ~ Exp
Boost           = ^ Exp<Float>
Constraint      = [+|-] Query
                | [+|-] ( Constraint+ )
```

Search language examples	*Embedded Java equivalent (core language)*
`search User matching name, nickname: query;`	`UserSearcher().fields(["name", "nickname"].query(query));`
`~getMovieSearcher() matching title^10.0,abstract^100.0: (+mustQuery -notQuery);`	`getMovieSearcher().fields(["title", "abstract"]) .boost("title", 10.0) .boost("abstract", 100.0) .must().query(mustQuery); .mustNot().query(notQuery);`
`search Product matching +(title, desc: "tablet") -(~notField: "Apple");`	`ProductSearcher().startMustClause() .fields(["title", "desc"]).query("tablet") .endClause() .startMustNotClause() .field(notField).query("Apple") .endClause();`

Table 6.11: Syntax: Specifications of query constraints

Field Selection

All query constraints are performed on one or more search fields. This set of fields can be enumerated, or left out when the default search fields are to be used (see Section 6.2.1). The targeted search fields can be specified statically at development time using the search field names, or dynamically using a tilde followed by a valid WebDSL expression that is String-compatible. Setting the search fields dynamically allows definitions of more generic templates and functions. Multiple search fields are separated by a comma.

Field Boosts

The order of search results should reflect the relevance within a search context. Per default, the Lucene ranking algorithm based on the Vector Space Model is used for sorting the results. In order to optimize the result ranking, the importance of search fields (relative to each other) can be changed on indexing time (Section 6.2.1). However, index-time boosting has some drawbacks. If the boost value of a field has changed, the search index need to be rebuild. Also, a field boost may not be applicable for all contexts the field will be queried in.

To overcome this, fields can also be boosted at query time by putting a boost suffix to the fields in a field enumeration. While index-time boosting is a little more efficient, query-time boosting enables one to boost fields dynamically. Search fields can be boosted differently in various contexts and might even be set up as user controllable variable in an (advanced) search user interface.

More Examples

As can be observed from Table 6.11, the size of the method call chain grows quickly in the native Java equivalents. The search language abstracts away from the diversity of method calls and replaces them with symbols for grouping, boosting and assignment of occurrences to clauses (groups of queries). Listing 6.12 shows examples of more advanced (combined) queries with explanations of the constraints.

```
//Find Movie entities that match "hello" in title or abstract (with matches in abstract
//being less important than in the title), but it should not match "goodbye" in any of
//the default search fields
var x1 := search Movie matching title^100.0,abstract^10.0: +"hello" -"goodbye";

//Find Movie entities that match "hello" in title or abstract,
//but doesnt match "goodbye" in title or abstract
var x2 := search Movie matching title,abstract: (+"hello" -"goodbye");

//Find Movie entities that match "hello" in the default search fields,
//but doesnt match "goodbye" in any of the default search fields
var x3 := search Movie matching +"hello" -"goodbye";

//Find materials with (name "metal peroxide") OR
//( 'some material that has "metal" in desc AND
//  "peroxide" in title AND OPTIONALLY
//  (a name or alias with "mp" but NOT "magnesium") )
var x4 := search Material
          matching ( descr:+"metal" name:+"peroxide"
                     name,alias:("mp" -"magnesium")
                   ) name:"metal peroxide";

//Searching a negative integer
var x5 := search User matching credit: (-200);
//Searching all except the given integer
var x6 := search User matching credit: -(200);
```

Listing 6.12: Examples of combining queries

6.4.3 Query Types

In Table 6.13, the query types currently supported by the search language are shown. The query types that are explicitly supported are text queries, range queries and proximity queries (also known as phrase queries). More query types are supported implicitly when Lucene's query syntax is set to be allowed. In addition to the explicitly supported types, it allows fuzzy and wild card queries. The Lucene query syntax is allowed by default, and can be disabled through a searcher attribute (see next section). Listing 6.14 shows an example of how to perform a wild card query using the Lucene query syntax, while not allowing this syntax to be used by the end user.

Syntax:

```
Query         = TextQuery
              | ProximityQuery
              | RangeQuery
TextQuery     = Exp
ProximityQuery = TextQuery ~ Exp<Int>
RangeQuery    = RangeOpen RangeLimit to RangeLimit RangeClose
RangeOpen     = [ //including
              | { //excluding
RangeClose    = ] //including
              | } //excluding
RangeLimit    = Exp
              | * //no limit
```

Search language examples	*Embedded Java equivalent (core language)*

`~s matching title:"singleTerm";`
`~s matching title:"multiple terms";`
`~s matching title:userQuery;`
`~s matching title:getSearchTerm();`
	`s.field("title").query("singleTerm");`
	`s.field("title").query("multiple terms");`
	`s.field("title").query(userQuery);`
	`s.field("title").query(getSearchTerm());`

`~s matching "should appear next to each other"~0;`
`~s matching "should appear within 5 tokens"~5;`
`~s matching -userQuery~5`
	`s.phraseQuery("should appear next to each other", 0);`
	`s.phraseQuery("should appear within 5 tokens", 5);`
	`s.mustNot().phraseQuery(userQuery, 5);`

`~s matching amount:[* to 0]; //0 or below`
`~s matching amount:[* to 0}; //below 0`
`~s matching rating:{6 to *]; //above 6`
`//above 0 up to value of uMax (inclusive):`
`~s matching amount:{0 to uMax];`
`//"AA" and up to "AB" (exclusive):`
`~s matching eLabel:["AA" to "AB"};`
	`s.field("amount").rangeQuery(null, 0);`
	`s.field("amount").rangeQuery(null, 0, true, false);`
	`s.field("rating").rangeQuery(6, null, false, true);`
	
	`s.field("amount").rangeQuery(0, uMax, false, true);`
	
	`s.field("eLabel").rangeQuery("AA", "AB", true, false);`

Table 6.13: Syntax: Specification of queries (various types)

6.4.4 Searcher Attributes

Additional settings related to a searcher can be set using *searcher attributes*. Currently, there are 2 settings that can be controlled. With the attributes `lucene` and `no lucene` one can (dis)allow the use of Lucene query syntax (default: allowed). The other attributes `strict matching` and `loose matching` determine if the tokens within a single query *must all* match or if *at least 1 token should* match (default) respectively. This might seem similar to the

```
var prefix := "";

form {
  "Search brands starting with:"
  input( prefix )

  submit action{
    var query := Searcher.escapeQuery( prefix ) + "*";
    var s := search Brand matching name: query;
    return resultPage( s );
  } { "search" }
}
```

Listing 6.14: Wild card query defined in Lucene query syntax while disallowing it to end-users (by escaping the query)

use of boolean operators on `MatchGroup` or `Constraint` constructs. The difference lies in the fact that boolean operators control the occurrence of constraints as a whole, while the ‘matching’ search attribute determines how *tokens within* a single query are to be combined by the query parser.

Syntax:

```
SearcherAttributes = [ {SearcherAttribute ,}+ ]
SearcherAttribute  = strict matching
                   | loose matching
                   | lucene
                   | no lucene
```

Search language examples	*Embedded Java equivalent (core language)*
`~s matching query [strict matching];`	`s.query(query).strictMatching(true);`
`~s [no lucene]`	`s.allowLuceneSyntax(false);`
`~s [loose matching, lucene]`	`s.allowLuceneSyntax(true).strictMatching(false);`

Table 6.15: Syntax: Searcher attributes

6.4.5 Sorting Results

The order of search results retrieved is determined by the ranking algorithm of Lucene. This behavior can be overridden by sorting on the values of search fields. This is useful for contexts in which relevance is related to a quantitative or temporal value, such as product prices and ratings or post-dates of articles. Sorting results using the searcher language (Table 6.16) is similar to other query languages.

Multiple levels of sorting are allowed, meaning that if a set of documents share the same values with respect to ordering based on the first sorting field, the second field will then be used to order the subset, etc. By default the relevance score will always be used as last sorting field in the chain.

Syntax:	

`OrderBy   = order by {SortExp ,}+`
`SortExp   = Field [Direction]`
`Direction = asc | ascending | desc | descending`	

Search language examples	*Embedded Java equivalent (core language)*
`~s matching userQuery order by price asc;`	`s.query(userQuery).sortAsc("price");`

`~s matching userQuery order by rating desc, price asc;`	`s.query(userQuery).sortDesc("rating")`
	`.sortAsc("price");`

`~s order by ~srtFld descending;`	`s.sortDesc(srtFld);`

Table 6.16: Syntax: Sorting search results

6.4.6 Filtering

The searcher language has language constructs to setup *filters*. Filters are different from queries and become attractive when constraints will repeatedly be applied. Filters are not taken into account for result ranking. Most search engines that process queries by combining lists of documents each matching some criteria (posting lists). A filter relieves the search engine from looking up documents in the index and from combining them once a filter has been initialized. A filter should be seen as a result set (documents) of a query, where each bit represents a document in the index. Filters are cached in memory and managed by Hibernate Search which is configured to hold at most 128 megabytes of hard references to filters. The rest of cached filters will be soft references that may be disposed on garbage collection performed by the Java virtual machine.

Syntax:	

`ConstraintFilter = with filter[s] {FieldFilter ,}+`
`FieldFilter      = Field : TextQuery`	

Search language examples	*Embedded Java equivalent (core language)*
`~s with filter isPublished:true;`	`s.addFieldFilter("isPublished", true);`
`~s with filters lang:"EN", ~fld:value;`	`s.addFieldFilter("lang", "EN").addFieldFilter(fld, value);`

Table 6.17: Syntax: Adding constraint filters

The applicability of filters is currently limited to simple field-query pairs (syntax in Table 6.17), where the query is an ordinary text query and thus analyzed using the analyzer bound to the targeted field. When using a filter, the searcher attributes also apply to the filtered query.

6.4.7 Search Namespaces

As described in section 5.4.6, WebDSL is extended with the notion of search namespaces. An entity property can be set up to act as namespace identifier meaning that the set of namespaces for an entity is equal to the set of unique values that exist for the targeted property. By assigning a namespace to a searcher instance, the set of results will be restricted to only contain entities with the specified namespace value. Current search namespace implementation is limited to target a single namespace only, or all namespaces when no namespace constraint is set.

Syntax:	
`NamespaceConstraint = in namespace Exp`	
Search language examples	*Embedded Java equivalent (core language)*

`~s in namespace:"EN";`
`search Article in namespace userSelectedTopic;`
	`s.setNamespace("EN");`
	`ArticleSearcher().setNamespace(userSelectedTopic)`

Table 6.18: Syntax: Adding namespace constraint

6.4.8 Pagination

In order to limit the number of results shown on a result page the starting offset (zero-based) and maximum number of results are controllable. Syntax and examples are shown in Table 6.19.

Syntax:	

`Offset     = offset Exp<Int>`
`MaxResults = limit Exp<Int>`	

Search language examples	*Embedded Java equivalent (core language)*

`~s limit 10;`
`~s offset 20 limit 10;`
`~s limit 10 offset (page-1)*10;`
	`s.setLimit(10);`
	`s.setOffset(20).setLimit(10);`
	`s.setLimit(10).setOffset( (page-1)*10 );`

Table 6.19: Syntax: Pagination over the result set

6.4.9 Faceted Search

Presenting facets will add organization to a browsable data collection such as search results. We implemented faceting as the presentation of the top *n* terms for some search field within the currently browsed context, where the context is a Searcher instance with its set of constraints (i.e., queries, filters, search namespace, and already selected facets). Retrieval and selection of facets is done using `Facet` instances, another new type. A single facet holds: its value, being the value of the associated term from the search index or a specified range in case of range faceting; its hit count within the currently browsed collection; and selection information that describes if a facet is currently selected with its occurrence type (must/must not/should match).

method	**description**
`getFieldName()`	returns the field name of the facet
`getValue()`	returns the value of the facet
`isSelected()`	returns true when this facet is selected in the searcher
`should(), isShould()`	Set/check the occurrence type. `Should` makes the facet constraint optional, but at least one facet for this field should match. (equal to `matching` ~`f.getFieldName(): f.getValue()`)
`must(), isMust()`	Set/check the occurrence type. `Must` means that a document will only match if and only if this facet does match. (equal to `matching` ~`f.getFieldName(): +f.getValue()`)
`mustNot(), isMustNot()`	Set/check the occurrence type. `Must not` means that a document will only match if and only if this facet does not match. (equal to `matching` ~`f.getFieldName(): -f.getValue()`)
`getValueAsFloat()` `getValueAsDate()` `getValueAsInt()`	helper methods for retrieval of value in appropriate type.

Table 6.20: Methods available in the Facet type

How it works

Any search field can be used for faceting. Be aware that the facet values to be presented in the user interface will be equal to the terms as they appear in the search index. It might therefore be required to add an additional search field solely for faceting that has no tokenizing/normalization applied.

In order to retrieve a list of `Facets`, faceting must first be enabled on a searcher. This is done by specifying which field(s) to use for faceting and how many facets to retrieve at most. In case of range facets, the number of specified ranges denote the number of facets to be retrieve, i.e., one facet for each range. Next, facets can be retrieved from a searcher by using the field name on which faceting was enabled. The retrieved facets can then be used as additional constraints on the facet-enabled searcher, which is similar to adding an additional query constraint, except that no boolean operators (+/-) are allowed. Listing 6.21 is a minimal example of a search page with faceting for product categories.

```
define searchbar(){
  var query := "";
  form {
    input(query)
    submit doSearch() {"search"}

    action doSearch() {
      //construct a searcher and enable faceting on categories.name, limited to 20 top categories.
      //more facets can be enabled by separating field(topN) facet definitions by a comma
      var searcher := search Product matching query with facets categories.name(20);
      return search(searcher);
    }
  }
}

define page search(searcher : ProductSearcher){
  var results : List<Product> := results from searcher;
  var facets  : List<Facet>   := categories.name facets from searcher;

  header{"Filter by product category:"}
  for(f : Facet in facets){
    facetLink(f, searcher)
  }separated-by{" "}

  showResults(results)
}

define facetLink(facet: Facet, searcher: ProductSearcher){
  submitlink narrow(facet){
    if(facet.isSelected()){ "+" }

    output(facet.getValue())
    "(" output(facet.getCount()) ")"
  }

  action narrow(facet : Facet){
    if (facet.isSelected()) {
      searcher.removeFacetSelection(facet);
    } else {
      ~searcher matching facet.must();
    }
    goto search(searcher);
  }
}
```

Listing 6.21: Search page with faceting

Faceting syntax

Table 6.22 illustrates the syntax for faceting and Table 6.20 the available methods for the `Facet` type. The reason for not allowing use of boolean operators on facet selection is mainly transparency. The occurrence operator is maintained in the facet itself. If we would allow boolean operators in a `FacetSelection` construct, the effect when using no boolean operator would become unclear: will it set the facet occurrence to *should* (similar to query specification) or will it respect the occurrence as set in a facet?

Syntax:

```
FacetDef       = with facet[s] {FacetExp ,}+
FacetExp       = Field ( TopN )
               | Field ( {RangeQuery ,}+ )
TopN           = Exp<Int>
RetrieveFacets = Field facets from Exp<Searcher>
MatchGroup    |= FacetSelection
FacetSelection = Exp<Facet|List<Facet>|Set<Facet>>
```

Search language examples	*Embedded Java equivalent (core language)*

`~s with facets authors.nameFull(20), year(10);`
 	
`~s with facets year([* to 2010},[2010 to *]), genre(10);`
 	
`~s with facet ~facetField(cnt);`
	`s.enableFaceting("authors.nameFull", 20)`
	` .enableFaceting("year", 10);`
	`s.enableFaceting("year", "[* to 2010},[2010 to *]")`
	` .enableFaceting("genre", 10);`
	`s.enableFaceting(facetField, cnt);`

`nameFull facets from s;`
`~facetField facets from s;`
	`s.getFacets("nameFull");`
	`s.getFacets(facetField);`

`~s matching price: [lower to upper], facetList;`
 	
`//must is default occurrence of a facet`
`~s matching someFacet.must(); //~s matching someFacet;`
`~s matching someFacet.mustNot();`
`~s matching someFacet.should();`
	`s.field("price")`
	` .rangeQuery(lower,upper)`
	` .addFacetSelection(facetList);`
	
	`s.addFacetSelection(someFacet.must());`
	`s.addFacetSelection(someFacet.mustNot());`
	`s.addFacetSelection(someFacet.should());`

Table 6.22: Syntax: Enabling, retrieving and selection of facets

6.5 Data and Meta-data Retrieval

This section describes how and which data can be retrieved from searcher instances and other facilities. During the design of the search syntax, we decided to add syntactic constructs for the most common ‘getters’ of search (meta-)data, namely the search results, search time, result size and facets. Additionally, we added syntax for retrieval of suggestions and hit highlighting for a given text.

```
Exp          |= RetrievalExp
RetrievalExp  = RetrieveResults
              | RetrieveFacets
              | RetrieveSuggestions
              | ResultSize
              | SearchTime
              | HighlightHits
```

Figure 6.23: Available retrieval expressions

6.5.1 Retrieval of Results, Result Size, Search Time and Facets

The basic syntax format for retrieval of data is by telling *what* to retrieve *from which searcher*. Table 6.24 is self-explanatory.

Syntax:

```
RetrieveResults = results from Exp<Searcher>
ResultSize      = count from Exp<Searcher>
SearchTime      = searchtime from Exp<Searcher>
RetrieveFacets  = Field facets from Exp<Searcher>
```

Search language examples	*Embedded Java equivalent (core language)*
`searchResults := results from s;`	`...:= s.results();`
`var r : List<Post> := results from search Post matching q;`	`...:= PostSearcher().query(q).results();`
`var c : Int := count from s;`	`...:= s.count();`
`var t : String := searchtime from s;`	`...:= s.searchTime();`
`var f : List<Facet> := category facets from s;`	`...:= s.getFacets("category");`

Table 6.24: Syntax: Retrieval of searched data

6.5.2 Presenting Relevant and Highlighted Fragments

The presentation of results can greatly reduce the required effort for an information seeking user. By presenting document surrogates (see Section 3.1.5) with elements relevant to the search context, a user can quickly see why a search result is considered relevant by the system. By providing a text and search field, a searcher can extract the most interesting fragments from the supplied text. It does so by analyzing the given text for query matches (for the query constraints as set in the searcher) using Lucene's Highlighter component. It won't highlight constraints set through a filter, namespace or facet selection, as these type of constraints are valid for all results

When using native Java expressions, the fragment length, number of fragments, fragment separator and surrounding tags for hit highlighting can be adapted using one of the available highlighter methods as shown in Table 6.25. Different method names control the behavior regarding HTML-tag preservation and the number of characters to analyze.

Highlighting Syntax

Common highlight methods that are available through native Java are covered by syntactic equivalents as shown in Table 6.26.

Workaround HTML Filtering

Unfortunately highlighting comes with a caveat. It requires a little trick to workaround HTML filtering which is performed by WebDSL (a feature that prevents HTML and javascript

Highlight Method Signatures

```
$mode$( field : String, toHighLight : String ) : String
$mode$( field : String, toHighLight : String, preTag : String, postTag : String ) : String
$mode$( field : String, toHighLight : String, preTag : String, postTag : String, fragments : Int,
        fragmentLength : Int, separator : String ) : String
```

Mode	**Description**
`highlight`	Highlight all matches for a given text, analyzing at most 50*1024 characters. Might break HTML-tags.
`highlightFullText`	Similar to *highlight*, except that it analyzes the whole text (less efficient for texts larger than 50*1024 characters).
`highlightHTML`	Similar to *highlight*, except that it preserves HTML-tags.
`highlightFullHTML`	Similar to *highlightFullText*, except that it preserves HTML-tags.
Parameter	**Description**
`field`	The search field to be used for finding hits.
`toHighLight`	The text to be analyzed for highlighting/fragment selection.
`preTag`	The tag to be inserted before a hit. default: `<span class="highlightcontent">`
`postTag`	The tag to be inserted after a hit. default: `</span>`
`fragments`	The number of fragments to extract from the given text. default: `3`
`fragmentLength`	The maximum fragment length in characters a fragment should have (last token will be left out if exceeds this value). default: `80`
`separator`	The text to be put between fragments default: `" ..."`

Table 6.25: Highlight methods

injection). The following example shows what happens if we output highlighted text the ordinary way.

outputting highlighted text the ordinary way:

```
output( highlight text: result.text from s )
```

...will result in highlight tags being escaped at runtime:

```
&lt;span class=\&quot;highlightcontent\&quot;&gt;a query match&lt;/span&gt;
```

...while this is what we want:

```
<span class="highlightcontent">a query match</span>
```

Syntax:

```
HighlightHits       = highlight Field : Exp<String> HighlightPart+
HighlightPart       = FromSearcher
                    | HighlightAttributes
                    | Tags
FromSearcher        = from Exp<Searcher>
Tags                = with tags Exp<String> , Exp<String>
HighlightAttributes = [ {HighlightAttribute ,}+ ]
HighlightAttribute  = HTML
                    | full text
```

Search language examples	*Embedded Java equivalent (core language)*
`var hl := highlight title: post.title from s;`	`...:= s.highlight("title", post.title);`
`var hl := highlight cnt: post.content from s [HTML];`	`...:= s.highlightHTML("cnt", post.content);`
`var hl := highlight cnt: post.content from s with tags "<em>", "</em>";`	`...:= s.highlight("cnt", post.content, "<em>", "</em>");`
`var hl := highlight cnt: post.content [full text, HTML] from s;`	`...:= s.highlightLargeHTML("cnt", post.content);`

Table 6.26: Syntax: Result highlighting

In order to workaround the HTML-filtering, the text to be used as input for highlighting should first be rendered to a String such that HTML-tags within the input text are escaped properly. The pre-rendered text can then act as input for highlighting. The result of highlighting should then be rendered *without escaping HTML*. This is done using WebDSL's `rawoutput` template.

In order to not expose this caveat to WebDSL users, we added ready-to-use template definitions to the built-in templates that capture this work-flow. The template `highlight(searcher, field){ elements }` can be put around elements and will only highlight hits in the rendered elements. Another template with signature `highlightedSummary(searcher, field, text)` extracts and outputs at most 3 fragments for the supplied text with the hits highlighted. The definitions of these built-in templates are shown in Listing 6.27.

6.5.3 Retrieval of Suggestions

Both spell and type-ahead suggestions can be retrieved when search fields are configured for that purpose using the `spellcheck` or `autocomplete` annotations as discussed in Section 6.2.1. Suggestion services use the tokens as they are tokenized by the analyzer specified for the search field. Controllable variables for suggestion retrieval are the search field(s) to use as suggestion source, the number of suggestions and, if configured, the targeted search namespace. For spell corrections the similarity, a measure between 0 an 1 based on Levenhstein edit distance, can also be provided (default 0.7). Table 6.28 shows the syntax for retrieving suggestions. The core language equivalents are accessible through static methods.

```
//Tries to highlight the elements inside, not touching the html tags inside
//(highlighter invoked to ignore html tags). If nothing is highlighted, it
//just renders elements
define highlight(s : Searcher, fld : String){
  var rendered   := rendertemplate( elements );
  var renderedHL := if(s != null) //get a single fragment without disposing text
    s.highlightLargeHTML(fld, rendered, "<span class=\"highlightcontent\">", "</span>", 1, 10000000, "")
                    else "";
  if(renderedHL != null && renderedHL.length() > 0) {
    rawoutput( renderedHL ) [all attributes]
  } else {
    rawoutput( rendered ) [all attributes]
  }
}

//Outputs a summary surrogate for the given text 'txt' based on constraints in searcher 's'
//for search field 'fld'. A result surrogate will consist of at most 3 fragments of max 80
//characters seperated by '... '.
//Hits are surrounded by tags <span class="highlightcontent">HIT</span>
define highlightedSummary(s : Searcher, fld : String, txt : String) {
  var decorated    := highlight ~fld: txt from s with tags ("HLOPENTAG","HLCLOSETAG") [HTML];
  var prerendered  := rendertemplate( output(decorated) )
  var tagsfixed    := prerendered.replace("HLOPENTAG", "<span class=\"highlightcontent\">")
                                 .replace("HLCLOSETAG","</span>");

  rawoutput( tagsfixed ) [all attributes]
}
```

Listing 6.27: Definition of the built-in highlight templates

Syntax:

```
Eid                   = Id //Id of entity
RetrieveSuggestions = Eid corrections MatchTerm {SpellPart}*
                      | Eid completions MatchTerm {AutocompletePart}*
MatchTerm             = matching {Field ,}+ : Exp<String>
SpellPart             = MaxResults | NamespaceConstraint | Similarity
AutocompletePart      = MaxResults | NamespaceConstraint
MaxResults            = limit Exp<Int>
NamespaceConstraint = in namespace Exp
Similarity            = similarity Exp<Float>
```

Search language examples	*Embedded Java equivalent (core language)*
`var sgs : List<String> := ...`	`var sgs : List<String> := ...`
`Movie completions matching title_ac: query;`	`MovieSearcher().autoCompleteSuggest(query, "title_ac", 10);`
`Topic completions matching name_ac, alias_ac: query limit 20;`	`TopicSearcher().autoCompleteSuggest(query, ["name_ac","alias_ac"], 20);`
`Topic corrections matching spell: query similarity 0.6;`	`TopicSearcher().spellSuggest(query, "spell", 0.6, 10);`
`Topic corrections matching spell: query limit 5 in namespace "sience";`	`TopicSearcher().spellSuggest(query, "sience", "spell", 0.7, 5);`

Table 6.28: Syntax: Retrieval of spell and type-ahead suggestions

6.5.4 Method Overview for the Searcher Type

Most common functionality of the Searcher type has been discussed during this chapter. Some additional methods for debugging and management and retrieval of constraint values are available that have not yet been explained. Table 6.29 shows all available Searcher methods with explanations of their purpose. The methods are grouped by concern.

Combining queries (boolean queries):

Method	Description	Default
`must()`	Adds a query definition (empty, to be specified) with occurrence set to *must* (AND) to the current clause	
`mustNot()`	Adds a query definition (empty, to be specified) with occurrence set to *must not* (NOT) to the current clause	
`should()`	Adds a query definition (empty, to be specified) with occurrence set to *should* (OR) to the current clause	
`startMustClause()`	Adds a child clause (*must*) to the current clause. The new child clause becomes the current clause	
`startMustNotClause()`	Adds a child clause (*must not*) to the current clause. The new child clause becomes the current clause	
`startShouldClause()`	Adds a child clause (*should*) to the current clause. The new child clause becomes the current clause	
`endClause()`	The parent of the current clause becomes the current clause again	

Field selection:

Method	Description	Default
`defaultFields()`	Sets the search fields for the current query and future queries to the default search fields	
`field(String fld)`	Sets the search field for the current query and future queries to fld	defaultFields()
`fields(List<String> flds)`	Sets the search fields for the current query and future queries to flds	defaultFields()
`getFields()`	Returns the fields set for the current query	
`boost(String, Float)`	Sets the boost value for a field in current and future queries	1.0

Note: On initialization of a Searcher, the fields are set to the default search fields as configured in the mapping. Query-time boosts are 1.0 if not overridden. Changing the fields/boosts with one of these methods will change the search fields/boosts for the current query *and subqueries not yet defined* within a searcher instance. Queries already defined are untouched.

Query specification:

Method	Description	Default
`phraseQuery(Object q, int s)`	Sets the current query to a phrase query q with slop s which controls how many tokens may appear between subsequent query tokens	
`regexQuery(String q)`	Experimental feature for querying using a regular expression q	
`matchAllQuery()`	Sets the current query to just match all documents (useful for faceting over an unrestricted data set)	
`query(Object)`	Sets the current query to a text query	
`rangeQuery(Object min, Object max)`	Sets the current query to a range query from min to max (inclusive)	
`rangeQuery(Object min, Object max, boolean inclMin, boolean inclMax)`	Sets the current query to a range query with in- or exclusive bounds	
`moreLikeThis(String likeText)`	Calls moreLikeThis(...) with minWordLen = 5, maxWordLen = 30, minDocFreq = 1, maxDocFreqPct = 100, minTermFreq = 3, maxQueryTerms = 6.	
`moreLikeThis(String likeText, int minWordLen, int maxWordLen, int minDocFreq, int maxDocFreqPct, int minTermFreq, int maxQueryTerms )`	Will set the current query to an advanced generated query, based on likeText. It does so by trying to extract interesting terms from likeText. The parameters control this term extraction. For explanation on these parameters, see [3]	
`getQuery()`	Returns the main query as String, being the first query set in a searcher	
`escapeQuery(String)`	Escapes Lucene syntax symbols for a given String	

Search attributes

Method	Description	Default
`allowLuceneSyntax(boolean)`	(Dis)allow lucene syntax. Applies to query and filters	true
`strictMatching(boolean)`	All tokens within a query *must* (true) match or at least one *should* match (false)	false

Filters

Method	Description	Default
`addFieldFilter(String fld, String val)`	Adds a field filter	
`removeFieldFilter(String fld)`	Removes filter for field fld	

[3] `http://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/api/contrib-queries/org/apache/lucene/search/similar/MoreLikeThis.html`

`clearFieldFilters()`	Removes all field filters	
`getFilteredFields()`	Get all fields used for filtering	
`getFieldFilterValue(String fld)`	Retrieve value of field filter	

Faceting:

`enableFaceting(String fld, int topN)`	Enables faceting on field fld returning the top-n terms for that field	
`enableFaceting(String fld, String ranges)`	Enables faceting on field fld capturing one or more ranges. Ranges are encoded as String, separated by commas, using the same syntax as RangeQuery (see Table 6.13)	
`getFacets(String fld)`	Retrieves the list of fld Facets (faceting must be enabled first on fld)	
`getFacetSelection()`	Retrieves the list of all Facets that have been selected (i.e. filtered)	
`getFacetSelection(String)`	Retrieves the list of fld Facets that have been selected (i.e. filtered)	
`addFacetSelection(Facet)`	Selects a Facet, i.e. adds the facet as additional constraint	
`addFacetSelection(List<Facet>)`	Selects a list of Facets	
`removeFacetSelection(Facet)`	Removes a single Facet from the selection	
`clearFacetSelection()`	Removes all facets from the facet selection	
`clearFacetSelection(String fld)`	Removes only the fld facets from the facet selection	

Search namespaces:

`setNamespace(Object)`	Sets the search namespace, i.e. restricting results to a single namespace	null
`removeNamespace()`	Removes search namespace constraint	
`getNamespace()`	Returns the namespace constraint currently set	

Pagination:

`setOffset(int)`	Sets the starting offset of results	0
`getOffset()`	Returns the starting offset	
`setLimit(int)`	Sets the (maximum) number of results to retrieve	50
`getLimit()`	Returns the limit	

Sorting:

`sortAsc(String fld)`	Specifies an additional/initial sorting field, sorting results using a field's value in ascending order	
`sortDesc(String fld)`	Specifies an additional/initial sorting field, sorting results using a field's value in descending order	
`clearSorting()`	Resets the sorting to default Lucene ranking	

Result/Search data retrieval:

`results()`	Retrieve the search results as a list of entities of the targeted type	
`count()`	Returns the total number of hits (independent of pagination)	
`searchTime()`	Returns a String representation of the search execution time	
`searchTimeMillis()`	Returns the search execution time in milliseconds (Integer)	
`searchTimeSeconds()`	Returns the search execution time in seconds (Float)	
`getQuery()`	Returns the main query as String, being the first query set in a searcher	

Serialization:

`asString()`	Serializes this searcher to a String	
`fromString(String)`	Constructs a searcher from a String	
`toParamMap()`	Serializes to a map of key-value pairs (required for serialization between page requests by WebDSL)	
`fromParamMap(Map<String, String>)`	Constructs searcher from a parameter map of key-value pairs (used by WebDSL internally)	

Debugging:

`scores()`	Returns a list of floats that represent the score for each result	
`explanations()`	Returns a HTML fragment with detailed score information. Usage: rawoutput(searcher.explanations())	
`luceneQuery()`	Returns a String representation in Lucene syntax of the eventual query that is used for searching	

Miscellaneous:

`reset()`	Resets a Searcher instance, removing all constraints	
`instanceOf(String tp)`	Checks if a Searcher is instance of a given type (type name as String), for example: `s.instanceOf("MovieSearcher")`	

Table 6.29: Overview of Searcher methods

6.6 Index Maintenance Tasks

While data and search index are kept synchronized by Hibernate ORM and Hibernate Search, there are situations in which management of the search indexes may be required. For this reason an index manager is generated that can be used from within the web application for search index-related tasks.

6.6.1 What is Automated Regarding Search Indexes

An index document will be created for each searchable entity that is created and persisted. On creation of such document, all search field values (including embedded ones) are retrieved from the data store and evaluated in case of an expression for a derived property.

Synchronization of Data Changes to the Search Index

Entity changes are also tracked for related changes to searchable data. In case there is a change in the searchable data of an entity, its associated document will be *recreated* and thus *all* search field values are retrieved/evaluated again. Similarly, it does this for all entities for which the system knows it has searchable properties which depend on this changed data. However, this means that the system should know about such dependencies. Searchable entity properties of simple type will work fine, as they are defined by the search mapping. Searchable properties of reference or composite type (i.e. mapping embedded search fields) require the embedded entity to have pointers to the owning (embedding) entity, such that the system can determine which entities have (search) dependencies on a changed entity. In order to update the search documents automatically on a change of an embedded entity the current implementation requires to have an *inverse relation* on one side of the relation between properties. The relation between `Author` and `Publication` in WebLib (Listing 6.2) is such an example. Using the inverse property annotation, a `Publication` entity will get re-indexed if searchable data in one of its author entities changes. If an inverse relation was missing in this case, a `Publication` entity would only be re-indexed when a searchable property value within that entity itself changes. A forced re-indexation from within app-code can be performed by calling the static method `IndexManager.reindex(Entity).`

Automatic Renewal for Suggestions and Faceting

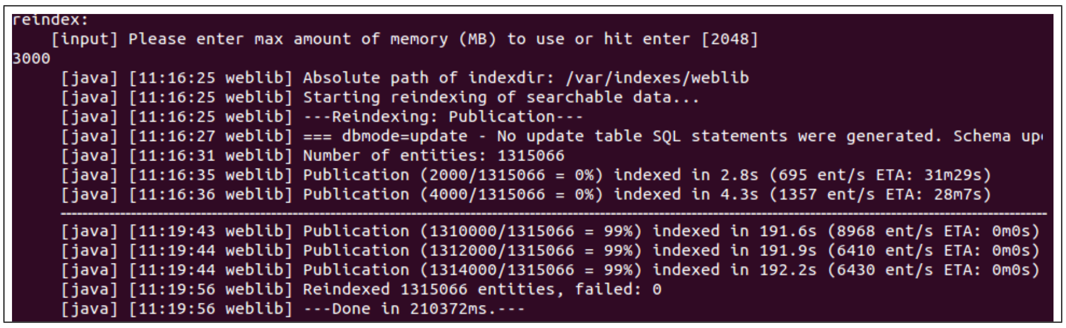
Facet, auto-complete and spell check facilities all use the search index as primary data source. For faceting, special *index readers* are automatically constructed on runtime with the appropriate instrumentation for the search fields used for faceting. These readers are renewed as the set of faceting specifications grow at runtime. Furthermore, they are checked for index changes every 15 minutes, such that changed data is reflected by the index readers used for faceting. Faceting data may thus become out-dated at some time if searchable data has changed. It may take up to 15 minutes at most to see these changes reflected in faceting contexts. An early renewal of the facet index readers can be forced by calling `IndexManager.renewFacetIndexReaders()` from within app-code.

Both spell and type ahead suggestion indexes are constructed automatically and get renewed if the search index changed. It checks for changes every 12 hours by default. In order to differentiate from this interval, the `IndexManager` offers methods for renewal of the suggestion indexes which are shown in table 6.31.

6.6.2 What is not Automated

In case application code is updated with changes made to the search configuration, an the existing search index that is constructed using older search configuration may become incompatible with the new configuration. A rule of thumb is to rebuild search indexes for an entity when new search fields are added for that entity or when the textual analysis performed at indexing time has changed. WebDSL does not do this automatically, because it renders the search functionality useless during index rebuild. Furthermore, re-indexing is likely to be a resource intensive task, as it will fetch all persisted entities and related data used for searching. It uses the `MassIndexer` of Hibernate Search [4] internally, spawning multiple threads for fetching and indexing.

Reindexing can be initiated from within a running web application, sharing the same servlet container instance (virtual machine). This is done by invoking `IndexManager.reindex()`, although it is recommended to perform an index rebuild in a separate VM with sufficient memory allocated. Initiation of an index rebuild outside a servlet container is done using the ant build file or the `webdsl-reindex` script for *nix platforms only which also fixes file-permissions of the index directory. These files are included in the deployed web applications and should run on same machine serving the web application. The amount of Java heap space to be allocated can be set at the beginning of re-indexation and progress information is printed to the console during this job (Figure 6.30)



```
reindex:
    [input] Please enter max amount of memory (MB) to use or hit enter [2048]
3000
     [java] [11:16:25 weblib] Absolute path of indexdir: /var/indexes/weblib
     [java] [11:16:25 weblib] Starting reindexing of searchable data...
     [java] [11:16:25 weblib] ---Reindexing: Publication---
     [java] [11:16:27 weblib] === dbmode=update - No update table SQL statements were generated. Schema up
     [java] [11:16:31 weblib] Number of entities: 1315066
     [java] [11:16:35 weblib] Publication (2000/1315066 = 0%) indexed in 2.8s (695 ent/s ETA: 31m29s)
     [java] [11:16:36 weblib] Publication (4000/1315066 = 0%) indexed in 4.3s (1357 ent/s ETA: 28m7s)
     ------------------------------------------------------------------------------------------------
     [java] [11:19:43 weblib] Publication (1310000/1315066 = 99%) indexed in 191.6s (8968 ent/s ETA: 0m0s)
     [java] [11:19:44 weblib] Publication (1312000/1315066 = 99%) indexed in 191.9s (6410 ent/s ETA: 0m0s)
     [java] [11:19:44 weblib] Publication (1314000/1315066 = 99%) indexed in 192.2s (6430 ent/s ETA: 0m0s)
     [java] [11:19:56 weblib] Reindexed 1315066 entities, failed: 0
     [java] [11:19:56 weblib] ---Done in 210372ms.---
```

Figure 6.30: Re-indexing

[4] `http://docs.jboss.org/hibernate/search/3.4/reference/en-US/html_single/#search-batchindex-massindexer`

Method	Description	Invoked every
Reindexing search index:		
`reindex()`	Rebuilds the whole search index for all entities, including suggestion indexes. This is a memory intensive job for large datasets. Indexing progress is outputted to console	
`reindex(Entity ent)`	Forces re-indexation of a single entity instance `ent`	
`indexSuggestions()`	Rebuilds the all suggestion indexes	12 hour
`indexSuggestions(List<String> namespaces)`	Rebuilds the suggestion indexes for the given namespaces and the non-namespace aware suggestion indexes	
`renewFacetIndexReaders()`	Forces renewal of facet index readers. useful after data changes to be reflected instantly	15 min
`optimizeIndex()`	Performs optimizations by Lucene to the search index	12 hour

Table 6.31: Overview of IndexManager methods

Chapter 7

Evaluation

In order to evaluate the domain-specific language extension on completeness and expressiveness we developed a demo project WebLib during development. In later stages, with syntactic abstractions implemented, two case studies were performed. In one case study we created a new web application for searching source code repositories. The other case study comprises extensions to an existing large digital library built with WebDSL. We also analyze search-related code from a different, non-WebDSL, web application to illustrate the effectiveness of the search DSL.

7.1 Case Study: Reposearch

During the master's project presented in this thesis, an idea came to mind to have a web-based search platform for the code repositories maintained within SERG (TU Delft Software Engineering Research Group).

Imagine 2 colleague developers Peter and Bob. Bob is currently busy adopting a logging framework into a project. Peter, already familiar with the framework, likes to show how they set up some of the powerful features of that framework during previous projects. Unfortunately, they are currently not behind Peter's desk and there is no easy access to the desired examples. Moreover, he doesn't know the exact files to look for. He vaguely remembers some of the names they used as log profiles.

Now, in order to get the desired code snippets Peter would normally have to download the source code from the repository, requiring him to remember or look up the repository URL. In order to search and view the source code easily, a development environment with search capabilities should be at hand. Depending on the system's hardware and implementation of search within the working environment, it might take another significant amount of time to get the results presented, or no results in case of a typo for which he needs to rephrase the query and start the search again. The idea of Reposearch[1] is to relieve the developer from these setup troubles and assist its users during query formulation by serving

[1] `http://codefinder.org`

code identifiers as type-ahead suggestions. We avoid requiring users to use a special query syntax for project, repository or programming language selection. By typing only a prefix of the targeted identifier and the selection of filters afterwards should direct a user to the intended code fragments.

A code search engine (CSE) should provide easy access to code fragments. Looking up code snippets is useful to developers for a number of reasons. It helps:

- understanding the meaning and working of software artifacts (reverse engineering)
- finding code for reuse (e.g. a previously applied workaround)
- finding usages of particular implementations (e.g. code that is potentially affected by a bug)
- finding the source of errors
- analyzing the impact of a refactoring

A web-based CSE also adds the capability for sharing code searches or snippets by a URL, improving code accessibility and collaboration.

7.1.1 Desired and Available Data

Code snippets are the targeted units of information. Data bound to code fragments include:

- The actual source code itself
- The file containing the code fragment, and therefore
  - The extension of the file they appear in (inferring the language of the source code)
  - The repository the code fragment belongs to
  - Its location within a repository as source code trees are mostly structured
- Repository meta data like the author, revision and creation/modification date

The identifiers of variables, functions, classes or other constructs are terms to be used for searching. Tokenization should guarantee that all occurrences of identifiers are indexed correctly.

7.1.2 Code Files to be the Searchable Units

The desired information are code *fragments*. The type of fragments (e.g. usage of a function or a method declaration) may be different for each search session. Different programming languages have different language constructs and the targeted type of code blocks are thus depending on the language of these snippets. A solution to deal with this variability is to have parsers for each language and treat each language construct individually as possible searchable unit. However, this requires a parser for every programming language to be supported, which implies that the programming language of each element to be indexed should first be classified. Aside from deciding on the interface for language construct extraction, unpopular programming languages are not likely to be supported and if the syntax of a supported language changes (not uncommon within academic projects), the parser definition must be updated too. Or even worse, different parsers for different language versions are needed. Regarding maintenance, this is not a preferred approach.

Luckily, the search terms (code identifiers) are very likely to appear in the desired code snippets. Consequently, the search highlighting facility can be used to provide interesting fragments (based on occurrences of the searched terms) from the source code unit in which they appear, i.e. the data-file containing the source code. For this reason we choose files to become the searchable entities. Other meta data for each file is also modeled in the data model to become searchable.

Listing 7.1 shows a slightly stripped down version of Reposearch's data model definition with some irrelevant entity properties removed for the sake of clarity. Each file from a code repository will be represented by an `Entry` instance, which belongs to a repository (`Repo`), being an `SvnRepo` or `GithubRepo`. Later we also added a `FileRepo` for uploading offline source code using a zip-file. `Repo` entities are bound to a `Project` entity, where a `Project` may hold references to multiple `Repos`.

```
entity Project {
  name    :: String (id)
  repos   -> List<Repo>
}
entity Repo {
  project -> Project (inverse=Project.repos)
}
entity SvnRepo : Repo {
  url     :: URL
}
entity GithubRepo : Repo {
  user    :: String
  repo    :: String
}
entity Entry {
  name        :: String  //file name
  content     :: Text    //file content
  url         :: URL     //URL to repository location of the file
  projectname :: String  //project the entry belongs to
  repo        -> Repo    //reference to the Repo it belongs to
}
```

Listing 7.1: Reposearch: data model

7.1.3 Searchable Data Selection and Analysis

Apart from the `Entry`'s content, its file name and derived file extension may contribute in finding the right code snippet. Indexing file names simply improves recall: if a file name matches a search query it should be included in the results. To improve precision in search results, user's may want to focus on, or filter out specific file types based on file extension. By indexing file extensions in a separate search field, these can be served as selectable facets when viewing results. This way, an information-seeking user is not required to define such constraints during the first query formulation. Instead, he may click the constraint during result presentation from an already constrained set of extensions, i.e. the set of file extensions appearing within the currently viewed result set. Listing 7.2 shows the search field specification for the `Entry` entity.

```
search mapping Entry {
  + content using keep_all_chars      as content
  + content using keep_all_chars_cs   as contentCase ^ 50.0
  content   using code_identifiers_cs as codeIdentifiers (autocomplete)
  + name    using filename_analyzer   as fileName (autocomplete)
  name      using extension_analyzer  as fileExt
  url       using path_analyzer       as repoPath
  namespace by projectname
}
```

Listing 7.2: Reposearch: search mapping

Indexing Source Code

The property `Entry.content` holds the plain content of the original file from the repository. Using the default built-in WebDSL text analysis would be a bad idea, since source code is different from natural language. Tokens (mostly identifiers in this case) may not necessarily be surrounded by white spaces; there is no list of stop words applicable for all programming languages; and symbols, including punctuation symbols, may have special meaning. We therefore decided to use the power of regular expressions for tokenization with preservation of symbols. Let us first look at the analyzer named `keep_all_chars`

```
analyzer keep_all_chars {
  tokenizer = PatternTokenizer(
                pattern="([a-zA-Z_]\\w*)|\\d+|[!-/:-@\\[-‘\{-~]",
                group="0" )
  token filter = LowerCaseFilter
}
```

input	*tokens*
`This.is_a-n0ns3nse example{return 123.456++;}`	`[this] [.]  [is_a] [-] [n0ns3nse] [example] [{] [return] [123] [.] [456] [+] [+] [;] [}]`

Table 7.3: Reposearch: analyzer for preserving all characters except white spaces

This analyzer uses the `PatternTokenizer` allowing a regular expression pattern to be used for tokenization. We constructed a pattern that matches 3 types of tokens. The first type catches identifiers and words: a sequence starting with a non-numeric character followed by zero or more word characters. The second pattern in the disjunction matches sequences of numbers. The last pattern catches single symbols as distinct tokens. After tokenization, the tokens are lowercased such that a user-entered query is not required to match the casing of the tokens in the original source code files.

Depending on the programming language, single tokens produced by this analyzer do not necessarily match lexical tokens (lexemes) of a programming language in question. This is especially true for symbols, where a sequence of symbols might represent a single lexeme, while analyzer `keep_all_chars` splits it into multiple single-symbol tokens. The search results for such lexemes, like the assignment operator (`:=`) in WebDSL which contains two tokens `[:, =]` according to our analyzer definition, should not be dominated with results that have both `:` and `=` somewhere but not near each other. So we need an exact match with respect to term positions as can be done by constructing a *phrase query* with slop set to 0.

Another analyzer, `keep_all_chars_cs`, is identical to `keep_all_chars` except that it respects the casing of characters. Search field `contentCase` uses this analyzer and the field is boosted to improve the score in ranking of entries that also match the casing of the query terms.

Type-ahead Suggestions

Another analyzer `code_identifiers_cs` is used to index identifiers for autocompletion. This analyzer is less aggressive regarding the splitting into tokens. It tries to find the longest sequences of characters that satisfy the character classes: letters; numbers; underscore; dot; or hyphen. Casing of tokens is preserved, thus type-ahead suggestions will show them in original casing. Now you may think: why tokenize differently for autocompletion? We decided to do so because it allows larger sequences, like fully qualified package names in Java, to be suggested. Also, some programming languages allow hyphens to be part of identifiers, like strategy/rule names in Stratego/XT. We wanted these to be suggested completely instead of only partly.

```
analyzer code_identifiers_cs {
  tokenizer = PatternTokenizer(
                pattern="[A-Za-z_]([\\-|\\.]?\\w)+",
                group="0" )
}
```

input	*tokens*

`This.is_a-n0ns3nse example{return 123.456;}`	`This.is_a-n0ns3nse`
	`example`
	`return`

Table 7.4: Reposearch: analyzer for extracting (chains of) identifiers for type-ahead suggestions

Faceting on File Extension and Location

Additional search fields `fileExt` and `repoPath` capture a file's extension and location respectively. Textual analysis of the file name (property `Entry.name`) and repository URL (`Entry.url`) is responsible for extracting this data. For the file names, the analyzer first normalizes file names without extensions by appending '`.(no ext)`'. Then, the sequence of characters following the last occurrence of a dot are captured as token, followed by normalization by lower-casing.

```
analyzer extension_analyzer {
  char filter  = PatternReplaceCharFilter( pattern="^([^\\.]+)$", replacement="/$1\\.(no ext)" )
  tokenizer    = PatternTokenizer( pattern="\\.([^\\.]+)$", group="1" )
  token filter = LowerCaseFilter
}
```

input	*tokens*
`root.app`	`app`
`template.build.properties`	`properties`
`x.JPG`	`jpg`
`readme`	`(no ext)`

Table 7.5: Reposearch: analyzer for extracting file extensions to be used for faceting

Regarding file locations, we first cut off protocol prefixes (e.g. `http://`) and file names. `PathHierarchyTokenizer` is then used which creates multiple tokens from a full path, one for each (intermediate) directory and each being a full path.

```
analyzer path_analyzer {
  char filter = PatternReplaceCharFilter(
                  pattern="(^.+://)(.*)/.*",
                  replacement="$2" )
  tokenizer   = PathHierarchyTokenizer( delimiter="/" )
}
```

input	*tokens*
`http://svn.repo.org/repos/proj/trunk/readme.txt`	`svn.repo.org`
	`svn.repo.org/repos`
	`svn.repo.org/repos/proj`
	`svn.repo.org/repos/proj/trunk`

Table 7.6: Reposearch: analyzer for constructing location tokens for faceting

7.1.4 Searching and Result Presentation

As search namespaces are used, one of the first constraints we can set is the targeted project. This can be a single project or all projects and it is selected by the user clicking the project on the homepage or from the site's top menu (Figure 7.7).

Figure 7.8 and 7.9 show the search form during first query formulation. It comprises an input element that suggests identifiers and file names matching the entered prefix and has controls for pagination and search modes. The WebDSL app-code related to the retrieval of suggestions is shown in Listing 7.10. Using JavaScript Object Notation (JSON), an array of suggestions is fed to a JavaScript function that shows the suggestions on the query input element. This function adds bold tags around the user-entered prefixes that match.

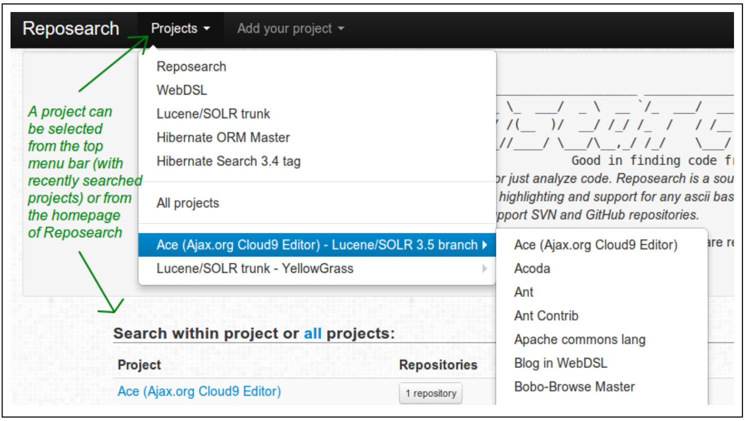

Reposearch
Projects
Add your project
A project can be selected from the top menu bar (with recently searched projects) or from the homepage of Reposearch
Reposearch
WebDSL
Lucene/SOLR trunk
Hibernate ORM Master
Hibernate Search 3.4 tag
All projects
Ace (Ajax.org Cloud9 Editor) - Lucene/SOLR 3.5 branch
Lucene/SOLR trunk - YellowGrass
Ace (Ajax.org Cloud9 Editor)
Acoda
Ant
Ant Contrib
Apache commons lang
Blog in WebDSL
Bobo-Browse Master
Search within project or all projects:
Project
Repositories
Ace (Ajax.org Cloud9 Editor)
1 repository

Figure 7.7: Reposearch: project selection

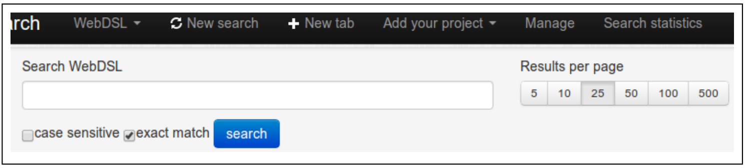


Figure 7.8: Reposearch: search form for first query formulation

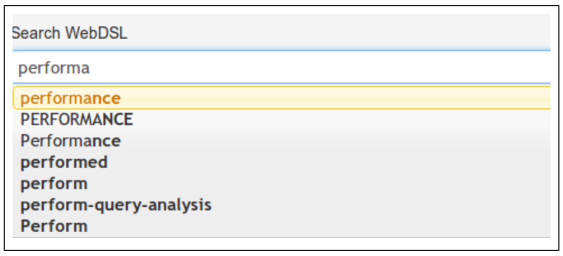


Figure 7.9: Reposearch: project-scoped suggestions while typing

```
service autocompleteService( namespace : String, q : String ) {
  var jsonArray := JSONArray();
  var results := Entry completions matching codeIdentifiers, fileName: q in namespace namespace limit 20;
  for( sug : String in results ) {
    jsonArray.put( sug );
  }
  return jsonArray;
}
```

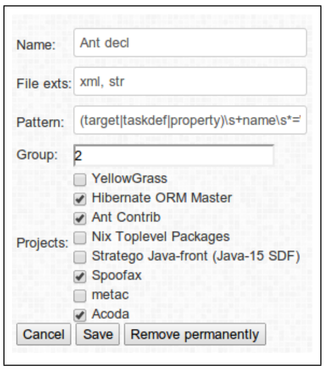
Listing 7.10: Reposearch: retrieval of suggestions

Search preferences are controlled with the check boxes on the search form and a button group for pagination. These preferences are maintained using a *session entity* `SearchPrefs`, which is a temporary entity bound to a browse-session for storing session related data (it expires after a time-out of inactivity). The time-out is set to 1 week. The preferences are retrieved and applied when a search is initiated. Listing 7.11 shows the function for constructing a searcher in Reposearch. The number of results per page are set similarly in the template for showing results.

```
function toSearcher( q:String, ns:String, langCons:String ) : EntrySearcher {
  var searcher := search Entry
                  in namespace ns
                  with facets fileExt(120), repoPath(200) [no lucene, strict matching];

  var slop := if( SearchPrefs.exactMatch ) 0 else 100000;
  if( SearchPrefs.caseSensitive ) { ~searcher matching contentCase, fileName: q~slop; }
  else                            { ~searcher matching q~slop; } //use default fields
  if( langCons.length() >0 ) { addLangConstructConstraint( searcher, langCons ); }
  return searcher;
}
```

Listing 7.11: Reposearch: Construction of searcher

Results are loaded instantly while entering a query. The search form is extended with options for restricting currently viewed result set using faceting for file extensions and file location. We later introduced filters for particular language constructs which are presented similar to the other facets. This feature is currently in prototype state. Its implementation is based on regular expressions and requires preprocessing of search results such that the highlighter extracts the targeted language construct from a result file and not any other fragment with matched tokens. Furthermore, the facets are retrieved using a `ConstructMatchSearcher` instead of the primary `EntrySearcher` that is used for searching and the retrieval of the other facets. Language constructs are defined at the management page of Reposearch by providing a name, valid file extensions, a regular expression, a matching group (denoting the part for matching a query against) and the set of projects for which this filter is enabled as shown in Figure 7.12.

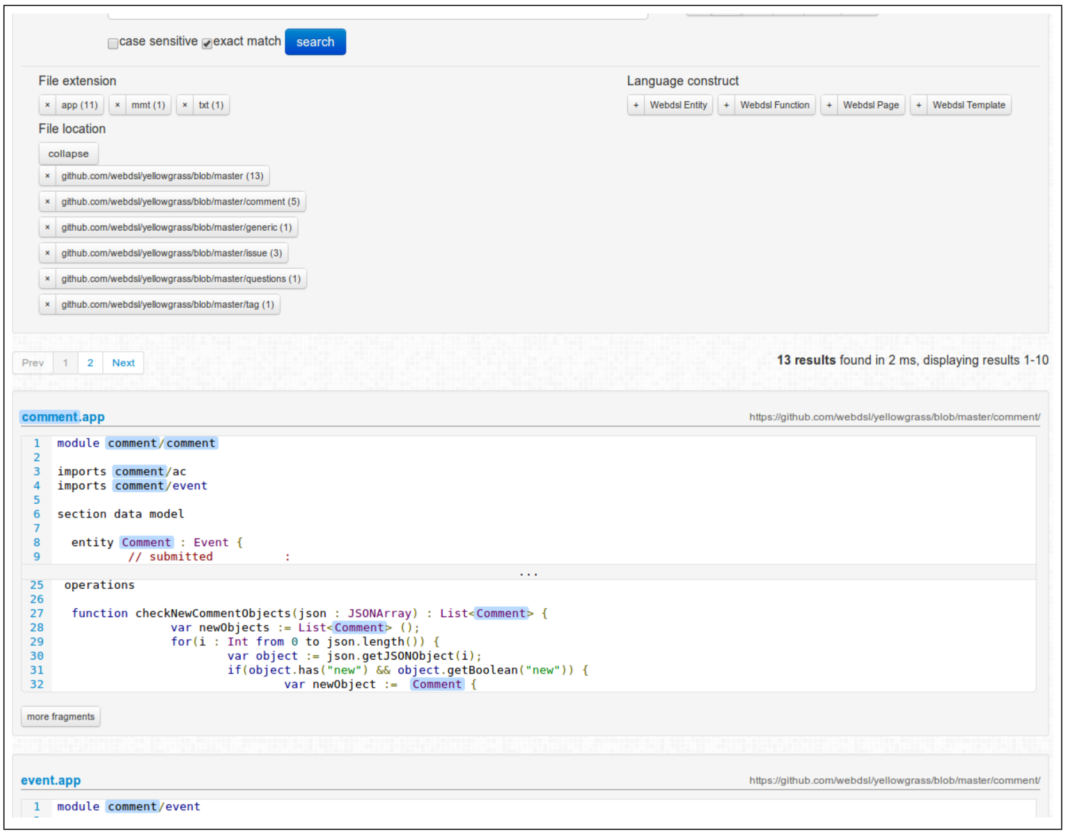


Name: Ant decl
File exts: xml, str
Pattern: (target|taskdef|property)\s+name\s*=
Group: 2
Projects:
[ ] YellowGrass
[✓] Hibernate ORM Master
[✓] Ant Contrib
[ ] Nix Toplevel Packages
[ ] Stratego Java-front (Java-15 SDF)
[✓] Spoofax
[ ] metac
[✓] Acoda

Figure 7.12: Reposearch: definition of language constructs

Extraction and Highlighting of Code Fragment

Results are shown below the search form. Each result consists of a clickable header with file name on the left and location on the right. By clicking the result header, the full file will load in a new page which directly jumps to the first code snippet extracted by the search


case sensitive
exact match
search
File extension
× app (11)
× mmt (1)
× txt (1)
Language construct
+ Webdsl Entity
+ Webdsl Function
+ Webdsl Page
+ Webdsl Template
File location
collapse
× github.com/webdsl/yellowgrass/blob/master (13)
× github.com/webdsl/yellowgrass/blob/master/comment (5)
× github.com/webdsl/yellowgrass/blob/master/generic (1)
× github.com/webdsl/yellowgrass/blob/master/issue (3)
× github.com/webdsl/yellowgrass/blob/master/questions (1)
× github.com/webdsl/yellowgrass/blob/master/tag (1)
Prev
1
2
Next
more fragments

13 results found in 2 ms, displaying results 1-10

comment.app https://github.com/webdsl/yellowgrass/blob/master/comment/

```
module comment/comment

imports comment/ac
imports comment/event

section data model

  entity Comment : Event {
          // submitted          :
                                   ...
  operations

  function checkNewCommentObjects(json : JSONArray) : List<Comment> {
                var newObjects := List<Comment> ();
                for(i : Int from 0 to json.length()) {
                        var object := json.getJSONObject(i);
                        if(object.has("new") && object.getBoolean("new")) {
                                var newObject :=  Comment {
```

event.app https://github.com/webdsl/yellowgrass/blob/master/comment/

```
module comment/event
```

Figure 7.13: Reposearch: result pane with selectable facets, highlighted fragments and pagination controls. Location facets are shown after clicking expand button.

highlighter. Below the result header, code fragments relevant to the search query are shown with clickable line numbers navigating to that line in the full file page. Code fragment extraction from the full file is done using the search highlight facility with fragment length set to 150 characters and number of fragments set to 3.

We needed to take into account that HTML-tags may appear within an indexed file and such tags may even be the targeted code fragments. We therefore cannot use the highlighter in HTML mode, which would ignore HTML tags for highlighting. Consequently, the text we feed to the highlighter component is the unfiltered content of a file (`entry.content`) as shown in Listing 7.14. We instruct the highlighter to surround matches with tags `$OHL$` and `$CHL$`. The resulting String with fragments of code is then filtered internally by rendering it to a String variable. Occurrences of the highlight tags are then replaced with HTML tags to be rendered as-is, i.e. using the `rawoutput` template from WebDSL which does not perform filtering.

More fragments within a single result can be loaded by clicking the button at the bottom of each result. This replaces the result in question with variable `noFragments` set to 10 instead of 3 (used in Listing 7.14). If the user opens the full file by clicking the result

```
var hlField := if( SearchPrefs.caseSensitive ) "contentCase" else "content";

var raw := searcher.highlightLargeText( hlField, entry.content, "$OHL$","$CHL$", noFragments,
      fragmentLength, "\n%frgmtsep%\n" );

var highlighted := rendertemplate( output( raw ) )
                      .replace( "$OHL$","<span class=\"hlcontent\">" )
                      .replace( "$CHL$","</span>" );
```

Listing 7.14: Reposearch: highlighting code fragments

header or a line number, the highlight code is invoked with the number of fragments set to 1 and fragment length set to a million characters at most.

Filtering using Facets

When results are available for an entered query, facets for file extension and file location are loaded into the search form (see Figure 7.13). A user can restrict the result set in 2 ways, namely by including or excluding a set of facets. Each facet is presented as combo-button. Clicking the right button (facet value) will select a facet for inclusion, or re-include a facet in case it has been excluded previously. Similarly, the left button (a cross) will exclude a facet or, in case of a selected facet, remove the facet from the selection. Listing 7.15 shows the related code for displaying the facets. The button state denotes the selection of a facet.

To keep the search user interface clean, the location facets are hidden behind a expand-button. When viewed, location facets are ordered by name. Intermediate directory locations without results are not shown. This is implemented by simply comparing the facet count between each 2 adjacent locations in the list of facets ordered by name.

Some Words About Line Numbering

At the result presentation and when viewing a full file, line numbers are placed left to the source code. These line numbers also facilitate in-page anchors such that a code line can be targeted in an URL. Line numbering was a serious issue during the design of Reposearch. Given the original (textual) content of a file and a fragment contained in this content, we cannot directly determine the line numbers of a supplied fragment. This problem arises when we construct the results on the search page: the highlighter component extracts code fragments from a full file, but does not tell us anything about the positional properties of the returned snippets.

To workaround this issue, we decided to encode the line number information into the code that is persisted in the data store. When an `Entry` is stored, the source code captured in property `Entry.content` is instrumented with line number tokens at the beginning of each line. The added tokens should not be interpreted as searchable tokens by Reposearch. This would affect the scoring and matching of code entries. We did so by extending the discussed analyzer definitions to remove such tokens by a token filter. The search highlighter also ignores these tokens for matching because this uses the analyzer bound to the search field that is provided to the highlighter. Line number information is thus still available in the

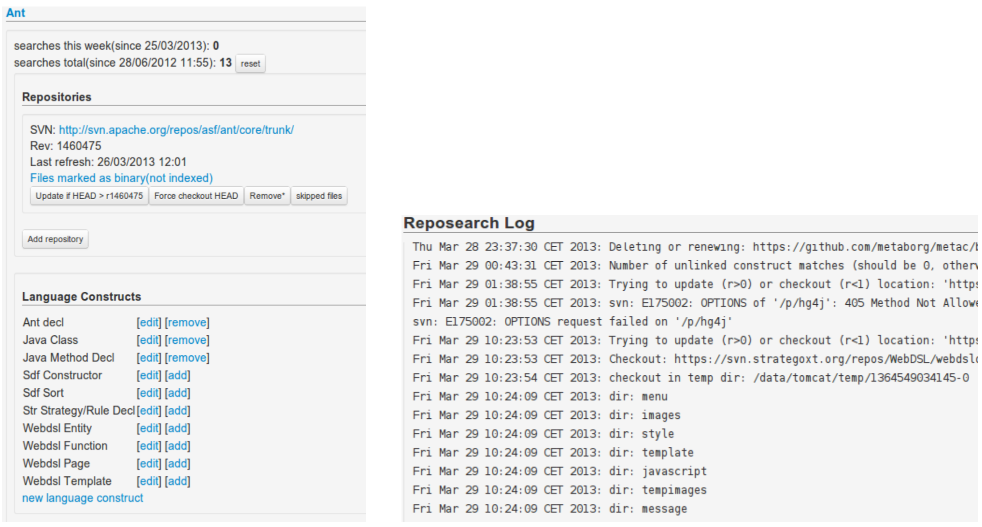
```
...
formEntry( "File extension" ) {
  for( f : Facet in fileExt facets from searcher ) {
    pullLeft { showFacet( searcher, f, ext_hasSel, namespace, langCons )  }
  }
}
...

define showFacet( searcher:EntrySearcher, f:Facet, hasSelection:Bool, ns:String, langCons:String ) {
  if( f.isMustNot() || ( !f.isSelected() && hasSelection ) ) { //use disabled button 'btnOff'
    if( f.isSelected() ) {
      submitlink updateResults( searcher.removeFacetSelection( f ) ) {
        buttonGroup {
          btnOff{ includeFacetSym() }
          btnOff{ output( f.getValue() ) " (" output( f.getCount() ) ")" }
        }
      }
    } else {
      submitlink updateResults( ~searcher matching f.should() ) {
        buttonGroup {
          btnOff{ includeFacetSym() }
          btnOff{ output( f.getValue() ) " (" output( f.getCount() ) ")"}
        }
      }
    }
  } else { //use enabled button 'btn'
    if( f.isSelected() ) {
      submitlink updateResults( searcher.removeFacetSelection( f ) ) {
        buttonGroup {
          btn{ excludeFacetSym() }
          btn{ output( f.getValue() ) " (" output( f.getCount() ) ") " }
        }
      }
    } else {
      buttonGroup {
        submitlink updateResults( ~searcher matching f.mustNot() ) [class="btn"]{
          excludeFacetSym()
        }
        submitlink updateResults( ~searcher matching f.should()  ) [class="btn"]{
          output( f.getValue() ) " (" output( f.getCount() ) ")"
        }
      }
    }
  }
  action updateResults( searcher : EntrySearcher ) {
    return doSearch( searcher, ns, langCons, 1 );
  }
}
```

Listing 7.15: Reposearch: presentation of facets

code fragments we receive from highlighting, and are distilled from the source code at the presentation of results. The required extensions to the analyzers were hidden in the listings we provided during this chapter as it would only add noise, i.e. it was unrelated to the context in which we discussed them.

7.1.5 Management Page

Reposearch is currently designed to have a single administrator account for accessing the management page. From here, the administrator can manage the projects and repositories (Figure 7.16), add new projects (possibly requested by a visitor), view log messages, control the update schedule (Figure 7.17) and change the home page message. We will not discuss the implementation details of these facilities, nor do we review the retrieval/updating of data from the repositories, as it is unrelated to the DSL extension.



Ant

searches this week(since 25/03/2013): **0**
searches total(since 28/06/2012 11:55): **13** reset

Repositories

SVN: http://svn.apache.org/repos/asf/ant/core/trunk/
Rev: 1460475
Last refresh: 26/03/2013 12:01
Files marked as binary(not indexed)

Language Constructs

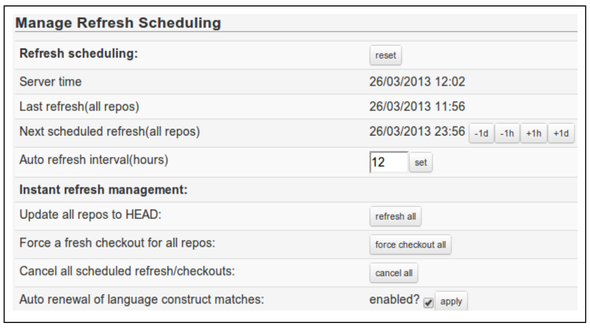
Ant decl	[edit] [remove]
Java Class	[edit] [remove]
Java Method Decl	[edit] [remove]
Sdf Constructor	[edit] [add]
Sdf Sort	[edit] [add]
Str Strategy/Rule Decl	[edit] [add]
Webdsl Entity	[edit] [add]
Webdsl Function	[edit] [add]
Webdsl Page	[edit] [add]
Webdsl Template	[edit] [add]

new language construct

Reposearch Log

```
Thu Mar 28 23:37:30 CET 2013: Deleting or renewing: https://github.com/metaborg/metac/
Fri Mar 29 00:43:31 CET 2013: Number of unlinked construct matches (should be 0, otherw
Fri Mar 29 01:38:55 CET 2013: Trying to update (r>0) or checkout (r<1) location: 'https
Fri Mar 29 01:38:55 CET 2013: svn: E175002: OPTIONS of '/p/hg4j': 405 Method Not Allowe
svn: E175002: OPTIONS request failed on '/p/hg4j'
Fri Mar 29 10:23:53 CET 2013: Trying to update (r>0) or checkout (r<1) location: 'https
Fri Mar 29 10:23:53 CET 2013: Checkout: https://svn.strategoxt.org/repos/WebDSL/webdsl
Fri Mar 29 10:23:54 CET 2013: checkout in temp dir: /data/tomcat/temp/1364549034145-0
Fri Mar 29 10:24:09 CET 2013: dir: menu
Fri Mar 29 10:24:09 CET 2013: dir: images
Fri Mar 29 10:24:09 CET 2013: dir: style
Fri Mar 29 10:24:09 CET 2013: dir: template
Fri Mar 29 10:24:09 CET 2013: dir: javascript
Fri Mar 29 10:24:09 CET 2013: dir: tempimages
Fri Mar 29 10:24:09 CET 2013: dir: message
```

Figure 7.16: Reposearch: Project management (left), log console (right)



Manage Refresh Scheduling

Refresh scheduling:	reset
Server time	26/03/2013 12:02
Last refresh(all repos)	26/03/2013 11:56
Next scheduled refresh(all repos)	26/03/2013 23:56 -1d -1h +1h +1d
Auto refresh interval(hours)	12 set
Instant refresh management:	
Update all repos to HEAD:	refresh all
Force a fresh checkout for all repos:	force checkout all
Cancel all scheduled refresh/checkouts:	cancel all
Auto renewal of language construct matches:	enabled? ✓ apply

Figure 7.17: Reposearch: schedule control on the management page

7.2 Case Study: Researchr.org

In our second case study, our focus was on an existing web application created with WebDSL. The application, Researchr[2], comprises a digital library where users can find, collect, share and review scientific publications. Being a digital library, search is one of the core features of Researchr. Before we empowered Researchr with new search features, search was purely based on the limited search capabilities of WebDSL. As WebDSL lacked support for searching embedded entity properties, it was not possible to search for publications by author names or tags. Textual analysis was done using the default analyzer, not matching inflections of words, hindering possibly relevant results.

Faceting was not present on the search page, but it was implemented for other contexts. Its implementation relied on data model extensions with entities for lists of publications, facet entities for those lists and categories owned by the facet entities. It required a lot of server-side maintenance done by the web application for updating, initialization and reconstruction of these entities on change/addition/removal of a related value. E.g., adding a single tag to a publication triggered a lot of database changes.

7.2.1 Researchr: Data Model

Configuration and the use of search facilities is tightly bound to the data model. We will now review the entities and entity properties most relevant for search in Researchr. These entities include: `Publication`, `Author`, `Tag`, `Bibliography`, `PublicationList`.

`Publications` are the main data elements in Researchr. There are several subtypes all inheriting from this entity, such as `MasterThesis`, `JournalArticle`, `Booklet` and `TechReport`. Basically, each `Publication` instance (Listing 7.18) has a `title`, `abstract` and `year` of publication. Furthermore, each publication has one or more references to `Author`, `Tag` and `PublicationList` instances. A `PublicationList` should be seen as intermediate entity used to group publications that have something in common, for example a `TagPublicationList` (inheriting from `PublicationList`) contains references to publications with the same tag. Similarly a `BibliographyPublicationList` is created for each `Bibliography`. A bibliography is a collection of publications maintained by a `User` or `UserGroup`.

`PublicationList` can be considered a redundant entity if its purpose was solely to provide the collection of Publications. This data can be derived from the existing entities for example by using the Hibernate Query Language (HQL) with a condition denoting the common value (e.g. `from Publication as p where theTag in p.tags`). Another purpose of the `PublicationList` entity was for providing facets when viewing lists of publications, as will be discussed in the next sections.

7.2.2 Previous Search Implementation

The search page on the Researchr website had Lucene syntax enabled. This can be very powerful for users familiar with the syntax, however it leads to empty result sets when

[2] `http://researchr.org`

```
entity Publication {
  key         :: String (id, index(25), collation(unicode_limited),
                         validate(isUniquePublication(this), "that key is already in use"),
                         validate(isValidKeyForURL(key), "Key should consist of letters, digits, : or -"))
  title       :: Text   (name, collation(unicode), searchable)
  authors     <> List<Author>  (index())
  year        :: String
  abstract    :: WikiText (searchable)
  lists       -> Set<PublicationList> (inverse=PublicationList.publications)
  tags        -> Set<Tag>
  aliasOf     -> Publication (optional)
    ...
}

entity Tag {
  key  :: String (id)
  name :: String (name, searchable,
                  validate(this.name.length() > 2, "Tags should have at least three characters."))
    ...
}

entity Author : AbstractAuthor {
  publication -> Publication (inverse=Publication.authors)
    ...
}

entity AbstractAuthor {
  name   :: String := this.alias.name
  alias  -> Alias (inverse=Alias.authors) // spelling of the name
  person -> Person (optional)             // identity of the author
    ...
}
```

Listing 7.18: Researchr: publication entity definition (snippet) in data model

there is a syntax error, which most users are unaware of. Search was also not configured to match inflections of words, e.g. when searching for ‘language‘ it did not match ‘languages‘, which led to absence of potentially relevant results. Furthermore, publications could only be searched by their title and abstract, not by author names, tags and publication year. These properties could not be indexed in the scope of the `Publication` entity, because indexing was limited to simple entity properties.

Another problem was that search often delivered duplicate results. This was caused by publications that have aliases, meaning that multiple publication are considered the same. This could happen if, for example, the same publication is entered by different users or by automated import of Researchr. In that case, one publication is considered the root publication and others aliases of that root.

7.2.3 Previous Faceting Implementation

Entity `PublicationList` was also used for implementation of a limited faceting interface. When a list of publications (e.g. a bibliography) was viewed, *facet clouds* were displayed alongside the publications. Depending on the context, facets were presented for publication year, authors, tags, venues and publication type. Faceting was limited in a sense that it only allowed filtering on a single facet. You could not filter a list of publications on multiple facets of the same type, nor could you combine multiple facet types. This was due to its implementation in which the list of publications for a facet were preconstructed. Multiple `PublicationListFacets` were constructed and persisted for each `PublicationList` instance, one for each type of facet: one for tags, one for publication year, etc. In turn, each `PublicationListFacet` instance contained a list of `Category` entities, one for each facet value. This means that a `Category` entity was persisted for each possible value of year, author, tag, etc appearing in a single publication list. The `Category` entity holds references to the publications that share the category's value in the context of the associated PublicationList. The category entity was used to retrieve and serve the associated publications after a facet value was clicked. Unfortunately, this implementation required a lot of automated maintenance. Entities needed to be kept up to date in case of a change in any of the related data. For instance, publication lists were updated when the set of contained publications changed. Assigning a tag to a publication triggered updating the `TagPublicationList` associated to that tag, and the reconstruction of all the `PublicationListFacet` and `Category` instances related to that `TagPublicationList`. Moreover, all facet-related entities were updated for all publication lists in which the newly tagged publication appeared. This resulted in many database updates and queries: not resource friendly.

7.2.4 A New Search Mapping and Flexible Searching

Our focus in this case study was to improve the search interface of Researchr, and to replace current faceting implementation by a more dynamic implementation using the new faceting features in WebDSL.

We started by moving the searchable property annotation to top-level search mappings into a single file `searchmapping.app`. For the `Publication` entity, we created additional search fields in order to support searching publications on more data than just the title and abstract. We marked the following search fields to become the default ones for searching publications: title, abstract, authors, tags, year and venues. This makes the search flexible in a sense that one can search the publications for some author, optionally for a specific year, or by using the name of a venue. The query 'berners lee 2009' searches for publications by Tim Berners Lee published in 2009, 'sigir 2011' for publications in proceedings of the SIGIR conference 2011 and 'oopsla steven fraser' for searching publications in proceedings of any of the OOPSLA conferences with Steven Fraser as author. The search mapping for `Publication` and `AbstractAuthor` are shown in listing 7.19.

Instead of using the default textual analysis, we extended this analyzer to reduce tokens into their root form, i.e. stemming. The library of SOLR token filters bundled with WebDSL offer various implementations to perform stemming. These include stemming algorithms expressed in the Snowball language designed for this purpose [24], such as the *Porter stemming* algorithm for English [25], the less aggressive *kStem* algorithm [18], a minimal stemmer implementation (`EnglishMinimalStemFilter`) and dictionary based stemmer using Hunspell[3]. We compared the behavior of these implementations by creating a

[3]`http://hunspell.sourceforge.net`

```
extend entity Publication {
  venue            :: String := this.venue()
  pubType          :: String := this.type()
  includeInResults :: Bool := (aliasOf == null)
}

search mapping Publication {
  + title    using publication_content
    title    using spell_check as spell (spellcheck)
  + abstract using publication_content
  + authors  with depth 2 //depth 2 in order to match on authors.alias._id
  + tags     with depth 1
    lists    with depth 1 //for restricting to a publication list context
  + year     using year
    pubType  using none
  + venue    using venue_value
    includeInResults
}

search mapping AbstractAuthor {
    person
    alias
    nameTag using none as name;
  + nameTag using standard_no_stop
}
```

Listing 7.19: Researchr: search mapping for Publication and AbstractAuthor entities

small WebDSL application[4] that has the English dictionary indexed. It shows which terms are considered equal for the variety of implementations. By evaluating using various words and derived inflections, it turned out that Porter2 stemming and the dictionary based Hunspell implementations were too aggressive for our purpose: they considered terms with different semantics to be equal. The word 'animate' demonstrates this quite well. As can be observed in Table 7.20 this term gets mixed with semantically different terms like 'animal' and 'reanimate'. We choose to go for kStem as it was capable of matching most important inflections of words, while not matching completely different words semantically.

In order to remove the duplicates search results for publications having aliases, the property `includeInResults` was added to the `Publication` entity. This boolean value is indexed and used to *filter* (not query) the search results, because this constraint will be applied for almost any search. A base `PublicationSearcher` applying this filter is obtained using a function `basePubSearcher` (Listing 7.21).

7.2.5 Facet Clouds Using New Search Capabilities

Aside from improving the search page of Researchr, another goal was to replace the faceting implementation with the built-in faceting facility of WebDSL. In order to use this facility, the requirement is that the currently browsed collection of publications is a collection that can be retrieved using a (`Publication`)`Searcher`. For example, when viewing the publications of a single author, we should be able to constraint on the author using a search field of the `Publication` entity. Similarly, other search fields are used for different constraints like

[4]`https://github.com/Elmervc/wordsearch` (links to live application)

Porter2 (Snowball)	**kStem**	**minimalStem**	**hunspell (OpenOffice dictionary)**
animalize	animaters	animates	animated
animal	animated	animate	unanimated
animalizes	animater		reanimates
animals	animators		reanimated
animated	animating		animating
animalisms	animates		animations
animalism	animate		animation
animality	animator		animates
anime			animate
animally			animatedly
animators			inanimated
animating			
animalized			
animations			
animation			
animates			
animism			
animate			
animes			
animatedly			
animator			

Table 7.20: Wordsearch: comparison stemming algorithms for the word 'animate'

the identifiers of `PublicationLists` in which a Publication appears (embedded search field `lists` in Listing 7.19). We added an entity property to get a `PublicationSearcher` instance with the right constraints for each entity that has collections of publications associated to it, such that this searcher can be used for the retrieval of the publications to be displayed and for the retrieval of facets. The app-code is shown in Listing 7.21. Using search for faceting, all instances of `PublicationListFacets` and `PublicationCategory` become obsolete and do not need to be maintained anymore. Furthermore, facet clouds can be retrieved for any set of constraint, i.e. also for the search page.

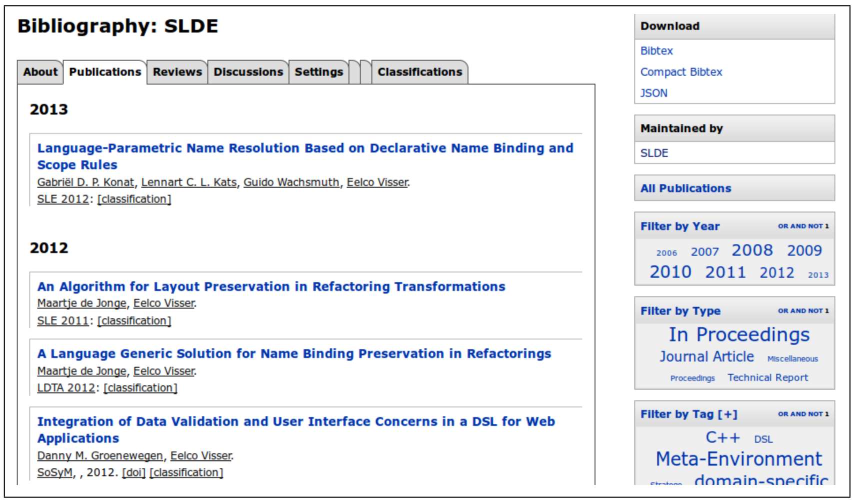
Generic Facet Cloud Templates

In Researchr, clouds of facet values are shown on the right of a publication collection, as is shown in figure 7.22. The original version of Researchr had generic templates for displaying the facet clouds on pages showing lists of publication. Clouds were collapsed by default, and could be expanded to view 40 or all facet values, using different templates respectively. In the search-based version of facet clouds, we reused some of the presentation-related code, for example to display facet values in different font size depending on their degree of presence in the currently viewed collection. The facet clouds are now constructed using a single, more generic template `facetCloudDisplay` (Listing 7.23). While this template is used for viewing a *single* facet cloud, it expects variables for *all* facet clouds for the reason that a change in facet selection, facet combinator or facet cloud size will replace all facet clouds. This is required because each facet cloud shares the same context of constraints,

```
function basePubSearcher() : PublicationSearcher {
  return search Publication with filter includeInResults: true;
}

extend entity Bibliography{
  pubSearcher -> PublicationSearcher := ~basePubSearcher() matching lists.identifier:list.identifier
}
extend entity Tag{
  pubSearcher -> PublicationSearcher := ~basePubSearcher() matching tags.name:name
}
extend entity ConferenceSeries {
  pubSearcher -> PublicationSearcher := ~basePubSearcher() matching venue: this.acronym
}
extend entity Person {
  pubSearcher -> PublicationSearcher := ~basePubSearcher() matching authors.person._id: this.id.toString()
}
extend entity Alias {
  pubSearcher -> PublicationSearcher := ~basePubSearcher() matching authors.alias._id: this.id.toString()
}
```

Listing 7.21: Searchers are retrieved using the ‘pubSearcher‘ property of the entity type denoting the context of a publication collection to be shown

including selection of other facet clouds. For example, a facet cloud showing the publication years will be different and should be updated when a user selects a facet from the author facet cloud, similar to the collection of publications that is shown. The `facetCloudDisplay` template expects a searcher instance with faceting enabled, the list of search fields that are used for faceting in the sidebar, accompanied with a list of occurrences describing how facets are to be combined for each faceting field, and a position denoting index in the field and occurrence lists for the current facet cloud.



Bibliography: SLDE

About | Publications | Reviews | Discussions | Settings | Classifications

2013

Language-Parametric Name Resolution Based on Declarative Name Binding and Scope Rules
Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser.
SLE 2012: [classification]

2012

An Algorithm for Layout Preservation in Refactoring Transformations
Maartje de Jonge, Eelco Visser.
SLE 2011: [classification]

A Language Generic Solution for Name Binding Preservation in Refactorings
Maartje de Jonge, Eelco Visser.
LDTA 2012: [classification]

Integration of Data Validation and User Interface Concerns in a DSL for Web Applications
Danny M. Groenewegen, Eelco Visser.
SoSyM, , 2012. [doi] [classification]

Download
Bibtex
Compact Bibtex
JSON

Maintained by
SLDE

All Publications

Filter by Year OR AND NOT 1
2006 2007 2008 2009 2010 2011 2012 2013

Filter by Type OR AND NOT 1
In Proceedings Journal Article Miscellaneous Proceedings Technical Report

Filter by Tag [+] OR AND NOT 1
C++ DSL Meta-Environment domain-specific

Figure 7.22: Researchr: viewing a collection of publications (a bibliography in this case) with facet clouds in a sidebar

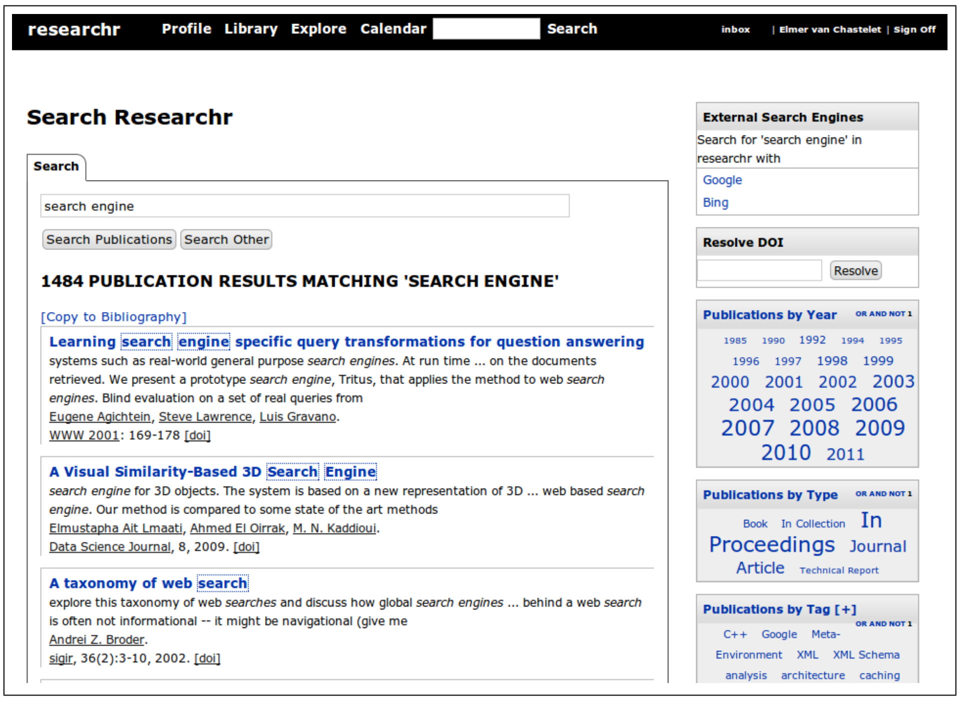
```
define span facetCloudDisplay( fields : List<String>, occurrences : List<String>, searcher : Searcher, pos
      : Int){
  var field   := fields.get( pos );
  var occur   := occurrences.get( pos );
  var heading := headingFromField( field );
  var facets  := getFacets(searcher, field);
  var minHits := minHits( facets ); //used for varying font sizes
  var maxHits := maxHits( facets ); //used for varying font sizes

  //shows heading of cloud with facet combinator options (AND/OR/NOT/1) and controls for
  //viewing a bigger[+]/smaller[-] facet cloud
  facetCloudDisplayHeading( fields, occurrences, searcher, pos, facets.length, heading, occur, field )

  //shows the 'cloud' of facet values
  block[class="cloud"]{
    for( f : Facet in facets order by facet.value asc ) {
      block[class="cloudTag"]{
        container[class:= facetFontLimit( f.getCount(), minHits, maxHits )]{
          if ( f.isSelected() ){
            submitlink changeSelection( f )[class="facet-link", title=f.getCount()+" matches"] {
              output( f.getValue() )
            }
          } else {
            submitlink changeSelection( f )[title=f.getCount()+" matches"] {
              output( f.getValue() )
            }
          }
        }
      }
    } separated-by { " " }
  }

  action changeSelection( facet : Facet ){
    if( facet.isSelected() ){
      searcher.removeFacetSelection( facet );
    } else {
      addFacetSelection( searcher, facet, occur );
    }
    //replace the list of publications (on the left) and facet clouds in sidebar (on the right)
    replace( publicationList, selectedCategory( searcher, 1) );
    replace( facetClouds, facetClouds( fields, occurrences, searcher ) );
  }
}

function addFacetSelection( s : Searcher, f : Facet, oc : String ) {
  case(oc){
    "should" { ~s matching f.should(); }
    "must"   { ~s matching f.must(); }
    "one"    { s.clearFacetSelection( f.getFieldName() ); ~s matching f.should(); }
    "mustnot"{ ~s matching f.mustNot(); }
  }
}
```

Listing 7.23: Researchr: template definition for displaying facet clouds

Besides having the ability to select multiple facets, possibly from different clouds, we also added control over *how* facets should be selected, which is where the occurrence variables are used for. Per facet cloud, a user can control if only one ('1') or multiple ('OR','AND','NOT') facet values can be selected at a time. This switch is placed on the right of each facet cloud header in Figure 7.22. If set to 'OR', at least one of the selected facet values should match a publication in order to appear in the list; 'AND' requires a publication to match all selected facet values; and 'NOT' will exclude all publications that match at least one of the selected facet values.

7.2.6 Updates to Search Page

As part of this case study we first improved access to the search page by adding the search input field to the top menu bar of Researchr. Previously, one had to click the top navigation bar opening a search menu where one could navigate to a search form targeting a specific type, namely publications, authors, tags, conferences, journals, groups and bibliographies. Search in the menu bar now directs to the publication search by default. From there one can target other types for search. Besides the improved analyzer and flexibility in matching various data bound to publications, further improvements were made to the search result page.



researchr Profile Library Explore Calendar Search inbox | Elmer van Chastelet | Sign off

Search Researchr

Search

search engine

Search Publications Search Other

1484 PUBLICATION RESULTS MATCHING 'SEARCH ENGINE'

[Copy to Bibliography]

Learning search engine specific query transformations for question answering
systems such as real-world general purpose *search engines*. At run time ... on the documents retrieved. We present a prototype *search engine*, Tritus, that applies the method to web *search engines*. Blind evaluation on a set of real queries from
Eugene Agichtein, Steve Lawrence, Luis Gravano.
WWW 2001: 169-178 [doi]

A Visual Similarity-Based 3D Search Engine
search engine for 3D objects. The system is based on a new representation of 3D ... web based *search engine*. Our method is compared to some state of the art methods
Elmustapha Ait Lmaati, Ahmed El Oirrak, M. N. Kaddioui.
Data Science Journal, 8, 2009. [doi]

A taxonomy of web search
explore this taxonomy of web *searches* and discuss how global *search engines* ... behind a web *search* is often not informational -- it might be navigational (give me
Andrei Z. Broder.
sigir, 36(2):3-10, 2002. [doi]

External Search Engines
Search for 'search engine' in researchr with
Google
Bing

Resolve DOI
Resolve

Publications by Year OR AND NOT 1
1985 1990 1992 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011

Publications by Type OR AND NOT 1
Book In Collection In Proceedings Journal Article Technical Report

Publications by Tag [+] OR AND NOT 1
C++ Google Meta-Environment XML XML Schema analysis architecture caching

Figure 7.24: Researchr: new search page with hit highlighting, result size, pagination (not visible), faceting, and did-you-mean functionality (not visible)

Faceting

Where in previous version Researchr, facet clouds were only presented when viewing pre-defined publication lists, they are now also shown on the search page for narrowing the result set further (both when searching publications and conferences).

Result Count and Improved Pagination

Using the new type searcher, we are now able to get meta-data about the search, like result size. We added the number of results on top of the result list and improved pagination at the bottom to respect this number. That is, only show browsable result pages instead of only controls for previous and next result page, where the next page could possibly be empty (prior to this case study).

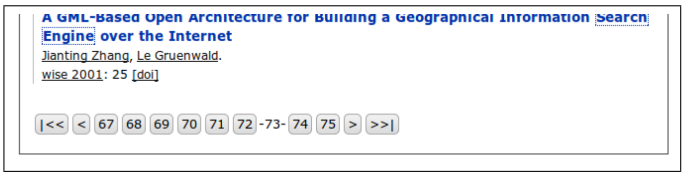

A GML-Based Open Architecture for Building a Geographical Information Search Engine over the Internet
Jianting Zhang, Le Gruenwald.
wise 2001: 25 [doi]

Figure 7.25: Researchr: pagination buttons only show browsable result pages

Highlighted Search Terms and Summaries

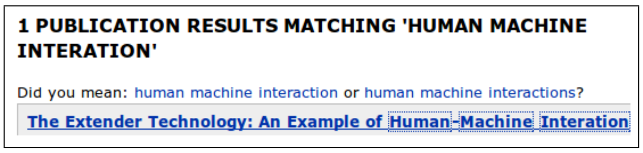
Prior to this case study, publication search results were presented by showing the title, authors, year of publication and, if applicable, the context in which it has been published. We improved these summaries, or *result surrogates*, by making them biased to the search query using the search highlighter component as can be observed from the result page in Figure 7.24. We added fragments of a publication's abstract which were not displayed previously. Listing 7.26 shows the application code related to result highlighting. Using the built-in highlighting templates discussed in Section 6.5.2, only the abstracts are summarized (at most 3 fragments of 80 characters maximum), other data is highlighted and displayed completely.

```
define span searchResult(pub : Publication, searcher : PublicationSearcher) {
  container[class="title"]{
    highlight(searcher, "title"){ output(pub) } //webdsl built-in template
  }

  abstractFragments(pub, searcher)

  highlight(searcher, "authors.nameTag"){ //webdsl built-in template
    if(pub.authors.length > 0) {
      container[class="authors"]{ outputAuthorsComma(pub.authors) ". " }
    }
  }
  ...
}

define abstractFragments(pub : Publication, searcher : Searcher){
  if(searcher != null && pub.abstract != null && pub.abstract.length() > 0){
    container[class="abstract"]{
      highlightedSummary(searcher, "abstract", pub.abstract) //webdsl built-in template
    }
  }
}
```

Listing 7.26: Researchr: presenting search results using highlighting component of WebDSL

Spell Suggestions

In case no or only a small result set matches a search query, chances are that the user made a typographic or spell mistake. In that case, we offer the user 2 possible spell corrections using the new spell check facility from WebDSL. Here, we use the publication titles as source for the spell check facility (see the search mapping in Listing 7.19), which stores terms containing 1, 2 or 3 adjacent tokens using the analyzer `spell_check` for which we already showed the definition in Chapter 6 (Listing 6.8).



1 PUBLICATION RESULTS MATCHING 'HUMAN MACHINE INTERATION'

Did you mean: human machine interaction or human machine interactions?

The Extender Technology: An Example of Human-Machine Interation

Figure 7.27: Researchr: suggesting corrections in case of small or no result set. In this case, the 'small' result set contains a publication with an erroneous title

7.2.7 Comparing Code Size

During this case study, we enriched and improved the search features of Researchr. By replacing the faceting implementation with the faceting abstractions added to WebDSL, some entity definitions and functions became obsolete. Search configuration has been moved from inline entity property annotations to a separate file containing the search mapping and analyzer definitions.

The expressiveness and conciseness of a programming language are hard to quantify. Most evaluations of programming languages compare the number of lines of code (LOC), tokens or statements for a program or algorithm implemented using a variety of programming languages, and use this value to express relative conciseness between programming languages. In our case, we are talking about a complete web application, where the focus is on the DSL extension for search in WebDSL. We performed a comparison on code size before and after the extensions made to Researchr. The comparison illustrate that the DSL extension requires less code to enable more search-related features. Search is now adaptable to the web application's context and has additional features for which there were no abstractions previously. With a (slightly) smaller code base we have a more feature rich application as can be observed in Table 7.28. The numbers represent lines of code and number of tokens in app-files of the complete code base of Researchr. Measurements are performed using *Source Code Line Counter* [5]. Blank lines, comment lines and (lines with only) delimiters are excluded in these measurements.

Currently we are further shrinking the code base by replacing the usage of the `PublicationList` entity with search (not reflected in Table 7.28). Source code related to the creation and maintenance of these entities will become superfluous. We expect the server load to decrease by this change, because the modifications/reconstruction of `PublicationList` entities due to any related change to a Publication entity is not required any more.

[5] `https://code.google.com/p/sclc/`

	Codelines*	**Tokens***
Before	15137	107445
After	15099	107042

Table 7.28: Researchr: Comparison on code size of the complete code base before and after case study.

7.3 Comparison to an Ordinary Java Web Application

In this section we compare code snippets from a real life open source web application developed in Java. We explicitly picked an application that uses Hibernate Search, because it already abstracts away from data store-search index synchronization and translation between index documents and Java entities, significantly reducing boilerplate code.

We studied the code base of *Eureka Streams*[6], a free, open source enterprise social networking platform developed by Lockheed Martin. Although search functionality in Eureka Streams is limited (e.g., it lacks hit highlighting, spell correction and faceting), the size of the program code responsible for search features is significantly larger than what we will discuss in this brief comparison. The purpose of this section is to illustrate the reduction in boilerplate code and gain in expressiveness by using the search language in WebDSL.

7.3.1 Mapping Class Fields to Search Fields

Data model entities are implemented as Java classes. Through the use of annotations, Hibernate Search lets one specify the mapping to search fields in the index documents including options on how to index the data. Listing 7.29 shows a snippet from the `Person` class. Three fields (`lastName`, `preferredName` and `dateAdded`) are mapped to equally named search fields. The highlighted `@Field` annotations contain parameters for indexing options, the field name and analyzer. Obviously, an analyzer applying stemming is used for tokenizing the last and preferred name. The `dateAdded` field is indexed without tokenization. It has a `@DateBridge` annotation on the field, informing Hibernate on how to transform the date into a String representation that will be added to the search index.

We created a mapping in WebDSL that is equal to this Java specification, shown in Listing 7.31. The highlighting colors denote the links between equal specifications. It shows that the notation of the search DSL is visibly more compact and expressive, and that there is no need to specify the translation from Date to Strings (as the WebDSL compiler is extended to handle this for us). The WebDSL code also includes the WebDSL counterpart `stemAnalyzer` of the analyzer that is used by Eureka Streams for which Listing 7.30 shows the implementation.

[6]`http://en.wikipedia.org/wiki/Eureka_Streams`

```
/**
 * The last name of this Person.
 */
@Basic(optional = false)
@Length(min = 1, max = MAX_FIRST_NAME_LENGTH, message = LAST_NAME_MESSAGE)
@Field(name = "lastName", index = Index.TOKENIZED,
// search is using text stemmer, so we need to index searchable fields with it
analyzer = @Analyzer(impl = TextStemmerAnalyzer.class), store = Store.NO)
private String lastName;

/**
 * The preferred name of this Person.
 */
@Basic(optional = false)
@Field(name = "preferredName", index = Index.TOKENIZED, store = Store.NO,
// analyzer
analyzer = @Analyzer(impl = TextStemmerAnalyzer.class))
private String preferredName;

/**
 * The date the user was added into the system, defaults to the current time, indexed into search engine.
     Note, for
 * the date to be sortable, it needs to be either Index.UN_TOKENIZED or Index.NO_NORMS.
 */
@Column(nullable = false)
@Field(name = "dateAdded", index = Index.UN_TOKENIZED, store = Store.NO)
@Temporal(TemporalType.TIMESTAMP)
@DateBridge(resolution = Resolution.SECOND)
private Date dateAdded = new Date();
```

Listing 7.29: Mapping to search fields in the Person entity class using Hibernate annotations

```
 ...
public class TextStemmerAnalyzer extends Analyzer
{
 ...
  public TokenStream tokenStream(final String fieldName, final Reader reader)
  {
      TokenStream tokenStream = new StandardTokenizer(reader);
      TokenStream result = new StandardFilter(tokenStream);
      result = new LowerCaseFilter(result);
      result = new StopFilter(result, StopAnalyzer.ENGLISH_STOP_WORDS);
      result = new EnglishPorterFilterFactory().create(result);
      return result;
  }
}
```

Listing 7.30: Implementation of the analyzer that is used for text searches in Eureka Streams

```
search mapping Person {
  lastName using stemAnalyzer
  preferredName using stemAnalyzer
  dateAdded
}

analyzer stemAnalyzer{
  tokenizer = StandardTokenizer
  token filter = StandardFilter
  token filter = LowerCaseFilter
  token filter = StopFilter
  token filter = SnowballPorterFilter(language="English")
}
```

Listing 7.31: Specification of search fields in WebDSL corresponding to the Java specification of Person

7.3.2 Specification of Search Constraints

As has been done in the previous mapping example, we will only discuss a minimal fragment of the code that is responsible for searching. Here, we take the backend implementation for searching persons by a prefix of their name. In this code snippet (Listing 7.32) four steps can be distinguished:

- ⬜ Escaping the user entered query for special characters meaningful to the query parser
- 🟦 Conditionally adding a constraint for filtering read-only entities
- 🟩 Specification of the actual query
- 🟧 Limiting the number of results to be retrieved

We appended the code associated to the `buildQueryFromNativeSearchString` call (in red), as this is part of query specification. This code is responsible for translating the query constraints into a representation accepted by Hibernate Search and can be considered boilerplate code. The method `getGroupVisibilityClause` is left out for the sake of fragment size, but it is similar to the conditional constraint, except that it adds a constraint based on the rights of the user that instantiated the search.

Again, we created a semantically similar WebDSL implementation shown in Listing 7.33. In this specific case, we are constructing a prefix-query for which the search DSL has no syntactic abstraction yet (see Section 6.4.3). We therefore escape the user query before we add the asterisk symbol, denoting a wild card query for the Lucene query parser. Comparing to the String construction in the associated Java code, query constraints in the search DSL are better readable. We modeled the conditional constraint for retrieval of read-only streams as a filter, as this constraint should not be taken into account for relevance ranking (which is the case for the implementation in Eureka Streams).

This example concludes this comparison, and this chapter. In the next chapter, we will look into other existing DSLs that target the domain of internal site search.

```
  ...
// build a search string that includes all of the fields for both people
// and groups.
// - people: firstName, lastName, preferredName
// - group: name
// - both: isStreamPostable, isPublic
// Due to text stemming, we need to search with and without the wildcard
String term = escapeSearchTerm(inRequest.getPrefix());
String excludeReadOnlyClause = excludeReadOnlyStreams ? "+isStreamPostable:true" : "";
String searchText = String.format("+(name:(%1$s* %1$s) lastName:(%1$s* %1$s) preferredName:(%1$s* %1$s
      )^0.5) " + "%2$s %3$s", term , excludeReadOnlyClause , getGroupVisibilityClause(inRequest));

FullTextQuery query = searchRequestBuilder. buildQueryFromNativeSearchString (searchText);

searchRequestBuilder.setPaging(query, 0, maxResults);

// get the model views (via the injected cache transformer)
List<ModelView> searchResults = query.getResultList();

      ...
Query luceneQuery;
try
{
    luceneQuery = inQueryParser.parse(nativeSearchString);
}
catch (ParseException e)
{
    String message = "Unable to parse query: '" + nativeSearchString + "'";
    log.error(message);
    throw new RuntimeException(message);
}

// get the FullTextQuery
FullTextEntityManager ftem = getFullTextEntityManager();

// wrap the FullTextQuery so we have more control over the control flow
ProjectionFullTextQuery projectionFullTextQuery = new ProjectionFullTextQuery(ftem.createFullTextQuery(
        luceneQuery, resultTypes));

// set the result format to projection
List<String> parameters = buildFieldList();
projectionFullTextQuery.setProjection(parameters.toArray(new String[parameters.size()]));

// set the transformer
projectionFullTextQuery.setResultTransformer(resultTransformer);

return projectionFullTextQuery;
```

Listing 7.32: Code responsible for searching Person entities by a prefix of their name

```
var q         := escapeQuery( query );
var prefixQ   := q + "*";
var searcher := search Person matching name, lastName, preferredName^0.5: (prefixQ q)
                              limit maxResults;

if ( excludeReadOnlyStreams ) { ~searcher with filter isStreamPostable: true }
```

Listing 7.33: Searching for persons by prefix in WebDSL

Chapter 8

Other Search DSLs and Future Work

In this master's thesis, a base language for web development is extended with search features using existing software artifacts from the solution domain. Similar examples can be found when studying web-frameworks like Ruby on Rails (*Rails* for short) and Django, using the host-languages Ruby and Python respectively. Search functionality provided with these frameworks is limited and based on the search capabilities of the database systems. More powerful search solutions are available using additional libraries available for Rails and Django. These libraries are examples of a domain-specific language embedded into a host language, called *internal DSLs* [8] or *domain-specific embedded languages* (*DSELs*) [16]. To the best of our knowledge, there are no documented examples of *external* DSLs for search like the DSL presented in this thesis. Our focus will therefore be on examples of embedded DSLs for search and how they differ from our external DSL.

8.1 Internal Search DSLs

Sunspot[1] for Rails is a search library, implemented in Ruby, that is build on top of RSolr (Solr library for Rails). It offers an expressive DSL both for index specification and searching. Sunspot adds abstractions similar to the ones implemented in WebDSL, such as querying (including phrase and range queries), faceting and highlighting. Sunspot also adds support for grouping of results and supports geospatial search. However it lacks abstractions for suggestion services (spell check, autocompletion). Haystack[2], a popular search framework for Django, is another example of an internal DSL for search. Haystack does have abstractions for suggestion services and supports geospatial search, but there is no abstraction yet for grouping of results. A unique feature of Haystack is the support for different search backends. Backend support for SOLR, ElasticSearch, Whoosh and simple SQL-based search is bundled, but one can also define a custom backend.

[1] `http://sunspot.github.io`

[2] `http://haystacksearch.org`

8.1.1 Host Language Dependencies

Abstractions in internal DSLs are implemented conform the notation of the host language by adding a library of new data types, functions/macros/routines, operators, etc., but without introducing new syntax. This limits the extent to which the language can be tailored to the problem domain: it should comply with the existing notation of the host language such as function calls and symbol usage, leading to a DSL with ‘noise’ from the host language. Look at the following Haystack code for indexing a Django model (similar to a WebDSL entity) `Note`:

```python
class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

Compared to search mappings in WebDSL, Haystack requires quite a lot of code only to configure the `Note` model to become searchable for 3 data properties. It requires a class definition for each model to become searchable with a function that returns the model in question and another one for returning the collection of model instances to be indexed when building the index from scratch. The extent of noise exposed by Sunspot with host language Ruby is significantly lower. This is because of the host language Ruby having extensive meta-programming facilities allowing the design of a concise internal DSL [9, 7]. Sunspot’s search configuration can be embedded in a model’s specification by adding a searchable block (line 7-10):

```ruby
 1 class Blog < ActiveRecord::Base
 2   has_many :posts
 3   has_many :comments, :through => :posts
 4
 5   attr_accessible :name, :subdomain
 6
 7   searchable :include => { :posts => :author } do
 8     string :subdomain
 9     text :name
10   end
11
12   # Make sure that includes are added to with multiple searchable calls
13   searchable(:include => :comments) {}
14 end
```

If we compare the specification of search fields with search mappings in WebDSL, we see that Haystack and Sunspot require to specify a field type for indexing. Index options and analysis (tokenization) are associated to these field types. In the example code snippets a `CharField` type in Haystack is similar to the `text` type in Sunspot. If data needs to be indexed differently than what the built-in types provide, one needs to escape to lower level code. In Sunspot, using SOLR as search backend, the underlying SOLR XML-schema can be adapted with new types and custom analyzers. In case of Haystack, supporting different search backends, one is required to define a custom backend. This can be done for example by adapting the definition of an existing one to fit the non-standard requirements. In contrast to these internal DSLs, the search DSL for WebDSL has language constructs to make custom analyzer definitions. Also, the search field attributes are all optional at search configuration, following the *convention over configuration* design paradigm more closely. Only providing a property name in a search mapping, the WebDSL compiler determines the type of the property and applies the default (conventional) index options for that type (see Section 6.3.2).

Another example, more related to the interface of the `Searcher` class we developed for use as core language, is the *query builder DSL* in Hibernate Search. This is not a complete search DSL, but an embedded language for the construction of Lucene queries and the management of facets in Hibernate Search. It allows method chaining similar to the interface of the Searcher class we developed for WebDSL, but is more sophisticated. Its fluent interface is implemented using various interface classes denoting the different contexts one can encounter during query construction. Each context-interface only has the allowed methods publicly accessible. Using code completion from an IDE one can easily obtain the available query construction methods with an interface that is close to natural language. Listing 8.1 shows how to construct a keyword query (a query without tokenizing the query) on multiple fields with the `name` field boosted.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity( Myth.class ).get();
Query luceneQuery = mythQB.keyword()
    .onField("history")
    .andField("name")
      .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

Listing 8.1: Query construction using Hibernate Search Query DSL

8.1.2 Static Error Checking

Internal DSLs take less effort to develop than external DSLs, because an internal DSL elaborates on a host language which already offers a notation and tool support (editor, debugger, compiler/interpreter). Logically, this puts restrictions on the extent to which these facilities can be tailored to the domain, as we already showed was the case for the notation of the discussed internal search DSLs. A different issue is the static verification of program consistency: a program may be valid for the host language, compiling successfully, however it may be configured incorrectly leading to errors popping up at runtime. Similar to how search configuration relates to an application's data model, in general, language aspects of a DSL or programming language depend on other language aspects that may be provided in various ways (as part of the language, base/host language or by an application programming interface). This hinders the ability to perform static (edit- and compile time) checks affecting different language aspects. For this reason, internal DSLs are often limited to the static verification performed by the host language, such as type checking in statically typed languages. Checks for consistency between different language aspects would require to build static verifiers for the framework of components which the DSL interacts with, i.e. a framework that was never intended to have static verification between its components.

Furthermore, reported errors that relate to internal DSL elements are often in the vocabulary of the host language, containing implementation details which is what the DSL tries to hide in the first place. Domain tailored error messages adopting domain concepts, like in our external search DSL (Section 5.6.1), are unlikely to be reported statically by internal DSLs. In case such language has the ability to report domain-friendly messages, these will likely be limited to checks performed at runtime and it requires these checks to be part of the DSL implementation. The range of checks and the details included in reported messages will further depend on the (meta-)data that is available at the moment of execution, in contrast to static checks performed at compilation/editing which have access to (meta-)data of the complete program and without hindering runtime performance if the required data needs to be retrieved or calculated.

8.2 Future work

Ideally, we would have added all search features we came across during the development of the DSL. We think that the search DSL in its current state offers most search features that are common in web applications. The work that we will now present was considered lower priority during the project. It primarily includes additional domain abstractions and improvements on existing integrated features.

8.2.1 Additional Domain Abstractions

Distributed setup

The base language WebDSL currently offers no abstractions to configure a distributed setup for running a web application on multiple machines. This would involve supporting multiple machine instances with possibly multiple database servers. The same holds for the search backends. Multiple machines may each run their own search engine and maintain their search index. Implementation involves synchronization between indexer and search engines on multiple machines which will depend on the way distribution is set up, i.e. by *sharding* or by *replicating* search indexes.

Named queries

A ‘Searcher‘ instance may be configured to hold multiple constraints of various types: queries, filters/namespace, selected facets. The latter two can already be managed using their associated search field names. However, ordinary query constraints are currently merged into a single query, where only the constraint value of the first configured query (treated as the main query) is memorized and accessible by calling `searcher.query()` (read-only). This hinders the management of *multiple* query constraints which a searcher may hold and requires a developer to maintain additional query constraints in additional template/page variables. A simple example is a search form that allows users to optionally enter query words that should *not* match the search results. In order to present a summary of the criteria set by the user, or allowing to redefine the entered criteria would require to store the additional query constraints in separate variables alongside the searcher instance.

In order to support managing multiple query constraints, the ‘Searcher‘ class should be extended to deal with an (optional) additional parameter, the subquery identifier, in the query-definition and related getter-methods. Similarly, the search language should allow binding these identifiers to the `Constraint`-constructs (Table 6.11). A next step would be to make fields and boosts manageable through a subquery identifier. Being an illustrative example, the syntactic sugar could be designed like this, where a main and not-query are bound to identifiers:

```
search Publication matching title, description: (userQuery@mainq -notQuery@notq);
```

Extended namespaces support

The search namespace abstraction is currently limited to target *all* or *one* namespace in a search context. This restriction relates to the implementation of suggestion facilities of the search language, which maintains separate suggestion search indexes for each namespace. Targeting multiple namespaces on facilities not depending on these suggestional search indexes can be achieved easily by adding additional filter constraints. More work is needed for adaptation of the suggestion services. Instead of having one index for each namespace, it would be better to share a single index for suggestion purposes for all namespaces like how it is done for ordinary search. Even better would be to have the data required for suggestion retrieval integrated into the main search. This would also replace the snapshot-based character of current implementation, where suggest indexes get renewed regularly. Newer versions of Lucene (4.0 and onward) add this support by a `DirectSpellChecker` component which constructs suggestions using the primary search index. Unfortunately, Hibernate Search does not yet support this version of Lucene.

Result grouping

The interface of the search language currently allows to retrieve results in order of relevance, or by sorting on one or more search field values. Not uncommon is to present results in groups that share a particular property. An example is the presentation of publications in

Researchr. Here, publications are grouped by the year of publication. This is achieved by configuring the searcher to order the results by the year field, additional WebDSL code then iterates the results and splits them into groups. This feature is not (yet) offered by Hibernate Search. Simple grouping of results can be implemented in a similar way as is done in Researchr. A syntactic abstraction (... `group by` *field*) can be wrapped around this implementation. Instead of retrieving a list of entities, a list of groups would then be retrieved. This implies adding a `Group` type to WebDSL which supplies access to the list of member entities and to the value they have in common.

Customized ranking

For some cases the default relevance ranking and the ability to sort results by field values may not give the intended order of results. This is the case, for example, when importance of results *partly* relates to a temporal property, such as a publication date for news articles. In this case, only sorting by date would give a newer article with only one match a higher score, while an older, more descriptive article with respect to the query terms would end up low in the ranking. For this to work, the scoring of documents must be adapted to reflect the extent to which they are important in the context of applying the search. In this case, a more recent publication date should give more weight than an older publication date. It should still be combined with the relevance ranking. A first step to support more advanced ranking is to look how this can be done using Lucene or Hibernate Search. Further investigation is needed on how to integrate this feature with the data model and expression language in WebDSL. Or possibly new language constructs may be required for accessing data held by search fields in the search documents.

Support more query types natively

In its current state, the range of query types assigned a domain abstraction in the search language is limited to text, proximity and range queries. Falling back to Lucene query syntax allows more types of queries. Supporting additional types of queries natively will reduce the needed effort for using these types. Other commonly used queries are fuzzy and wild-card queries and regular expressions. The latter is already partly supported using the Searcher methods.

Spatial Search

Abstractions for spatial search are not yet integrated in the search DSL. Geographic locations are not (yet) supported by WebDSL. Introduction of new types to support locations in the WebDSL data model will be one of the first steps in supporting spatial search. Then, this type of data should get supported in the configuration aspects of the search language (search mappings). The Hibernate Search version currently used as backend for search facilities (version 3.4.2) does not support spatial queries. Support is added in Hibernate Search 4.2.0 which implies upgrading Hibernate Core, the object-relational mapper, from 3.6.2 (current) to 4.1.9 or later. The latter upgrade requires migration work affecting various parts of the code generation and static source code and configurations in WebDSL. As part of this master's project we already made an attempt upgrading to Hibernate Core and search version 4. Unfortunately it turned out to be more effort to get things working, which made us decide to postpone this migration. It is recommended to continue this work not only for easier integration of spatial search, but also for improved performance and continued support from JBoss and its community. Adding support for spatial search without upgrading Hibernate would involve instrumenting the indexing process (which is supported by Hibernate Search) to support geographical data. At search time, a contributed Lucene library for spatial search is to be used to query the spatial data.

Syntactic support for spatial search would involve adding spatial specific constraint types, like searching for items that match a location within a radius around a given location, and ordering results with the nearest items appearing first. When spatial data gets adopted in WebDSL's type system, it is likely to support expressing areas around a geographic location, which in turn can be used to express geographic boundaries for search.

8.2.2 Improve Integration with Access Control

WebDSL has a flexible, integrated language for access control. By means of access control rules, elaborating on the WebDSL expression language, access to pages and templates can be controlled [12]. Using these rules, WebDSL handles visibility of page elements and navigation links automatically. For instance, when access control is set up to deny access to view particular products (e.g., the `Product` entities where a property `visible` property is set to false), data bound to these entities will not be displayed. This also applies to the collection of entities retrieved using search, which is also where the integration of access control and search stops. Other data retrieved from a `Searcher`, such as result size, facets and suggestions currently do not respect access restrictions modeled using access control rules.

In the current implementation, access control rules can be used to assure specific search filters are set that represent the access control restriction. In the former example, this means that the property `Product.visible` should be mapped to a search field which is to be used as search filter field for a Searcher instance. This way, access restrictions will get applied *during* the execution of the search by the system, resulting in correct result sizes, result collections and facets to be retrieved. However, taking existing access control rules as a base, this approach requires that these rules can be mapped to search filters at the moment of searching, implying that the data used by access control rules is mapped to search fields. Derivation of search field mapping for automatic access control weaving into search would be complex, if not impossible, due to range of checks and data to be accessed in access control rules. Determining the feasibility of such integration requires further investigation. Another problem comes along in the retrieval of suggestions. Current implementation is limited to query *all* of the available terms for a field, or a subset of these when search namespaces are used. One cannot limit these sets further with additional criteria. This would require a solution, also applicable for targeting multiple search namespaces, which uses the primary search index for the retrieval of suggestions, allowing additional constraints to be set for controlling access.

Chapter 9

Conclusions

We successfully created a usable domain-specific language for internal site search, integrated as sublanguage in WebDSL. The language offers features to facilitate search functionality based around a web application's data model (i.e. different from web search which is based around crawlable web pages). We have identified several concerns, each covering a range of activities typically encountered when implementing search into a (web) application, independent of the engine powering the actual search. We extended the base language WebDSL to allow expressing a substantial part of the variable elements linked to these concerns.

Index document construction

This includes the selection of data a document should hold, and the tokenization and normalization of this data. We integrated declarative *search mapping* and *analyzer* constructs to allow the selection of data to become searchable and specification of transforming the (textual) data into, optionally normalized, tokens. By elaborating on the analyzer framework supplied with Apache Lucene/SOLR, a wide range of tokenization and normalization requirements can be fulfilled. Search mappings are expressive and adopt the *convention over configuration* paradigm by requiring as little as the name of the entity property in order to map that property to a search field.

Specification of search constraints

A new type *Searcher* is added that covers actions and administration of data related to a search session (or generally speaking, browse session). The searcher interaction sublanguage allows the formulation of constraints through queries of various types, filters and faceted search. Searcher instances can be created and adapted at different elements of a WebDSL application and can easily be exchanged between page requests. This fits the stateful character of a browse session, where a query formulation phase is often followed by one or more reformulation or refinement phases.

Retrieval of data

The searcher type and sublanguage support pagination, ordering and retrieval of search results and other search-related data including facet values, result size and search execution time. It also supports the retrieval of query-biased result summaries with query hits highlighted.

User interface

WebDSL templates are to be used for the design of the search user interface. User-aiding search features have been added to WebDSL allowing a rich user interface during browsing and searching. I.e., serving type-ahead and spell/typographic suggestions, faceted search and result summary extraction.

DSL development was driven by programming patterns and common practices in the solution space of the problem domain, similar to previous WebDSL development. Domain abstractions were added iteratively, where the DSL was in a usable state after each addition, allowing evaluation of the DSL by means of an application being developed in parallel. Implementation differed between aspects of internal site search, but mainly elaborated on using core languages which were already part of the base language. Existing support for embedding target language code (Java) in WebDSL enabled efficient implementation for features covered by the Searcher type. The resulting interface of the Searcher type serves as core language for the syntactic sugar being designed in a separate phase afterwards. Similarly, property annotations were already part of the data model specification language in WebDSL, and serve as core language for the search mapping constructs introduced later. Editor services were implemented last, resulting in a useful language with tool support.

Evaluation by means of case studies showed that WebDSL is capable in facilitating search for web applications serving large collections of data. Features such as faceted search, multi-term *did-you-mean*-suggestions, auto-completion, and the extraction and highlighting of textual fragments from search result could all be established using the extended WebDSL language. The DSL abstracts away from search engine specific implementation details which require significantly more (boilerplate) code. Other search DSLs designed for web frameworks show a similar feature-set as the DSL presented in this thesis. The main difference is that these search languages are implemented as *internal* DSLs, using the notation of a general purpose language. Our *external* DSL excels in expressiveness, conciseness and the ability to check for application consistency statically (involving cross-concern checks) with errors reported in a dialect that adopts domain concepts.

Bibliography

[1] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Trans. Internet Techn.*, 1(1):2–43, 2001.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[3] Andrei Z. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[4] Charles Consel. From a program family to a domain-specific language. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 19–29. Springer, 2003.

[5] Charles Consel and Renaud Marlet. Architecture software using a methodology for language development. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP 98 Held Jointly with the 7th International Conference, ALP 98, Pisa, Italy, September 16-18, 1998, Proceedings*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194. Springer, 1998.

[6] Qing Cui and Alex Dekhtyar. On improving local website search using web server traffic logs: a preliminary report. In Angela Bonifati and Dongwon Lee, editors, *Seventh ACM International Workshop on Web Information and Data Management (WIDM 2005), Bremen, Germany, November 4, 2005*, pages 59–66. ACM, 2005.

[7] H. Cunningham. A little language for surveys: constructing an internal dsl in ruby. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX*, New York, NY, USA, 2008. ACM.

[8] Martin Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.

[9] J. Freeze. Creating dsls with ruby. http://www.artima.com/rubycs/articles/ruby_as_dsl.html, March 2006.

[10] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A cognitive dimensions framework. *J. Vis. Lang. Comput.*, 7(2):131–174, 1996.

[11] Stephan Greene, Gary Marchionini, Catherine Plaisant, and Ben Shneiderman. Previews and overviews in digital libraries: Designing surrogates to support visual information seeking. *JASIS*, 51(4):380–393, 2000.

[12] Danny M. Groenewegen and Eelco Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In Daniel Schwabe, Francisco Curbera, and Paul Dantzig, editors, *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*, pages 175–188. IEEE, 2008.

[13] David Hawking. Challenges in enterprise search. In Klaus-Dieter Schewe and Hugh E. Williams, editors, *Database Technologies 2004, Proceedings of the Fifteenth Australasian Database Conference, ADC 2004, Dunedin, New Zealand, 18-22 January 2004*, volume 27 of *CRPIT*, pages 15–24. Australian Computer Society, 2004.

[14] Marti A. Hearst. User interfaces and visualization. In *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[15] Marti A. Hearst, Ame Elliott, Jennifer English, Rashmi R. Sinha, Kirsten Swearingen, and Ka-Ping Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49, 2002.

[16] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

[17] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[18] Robert Krovetz. Viewing morphology as an inference process. In Robert Korfhage, Edie M. Rasmussen, and Peter Willett 0002, editors, *Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Pittsburgh, PA, USA, June 27 - July 1, 1993*, pages 191–202. ACM, 1993.

[19] Mark Levene. *An introduction to search engines and web navigation*. Wiley. com, 2011.

[20] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, feb 1966.

[21] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[22] Rajat Mukherjee and Jianchang Mao. Enterprise search: Tough stuff. *ACM Queue*, 2(2):36–46, 2004.

[23] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[24] Martin F. Porter. Snowball: A language for stemming algorithms. Published online, October 2001. Accessed 9 April 2013 17.00h.

[25] Martin F. Porter. The english (porter2) stemming algorithm. Published online, September 2002. Accessed 9 April 2013 17.15h.

[26] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[27] Bran Selic. A systematic approach to domain-specific language design using uml. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*, pages 2–9. IEEE Computer Society, 2007.

[28] Ben Shneiderman, Don Byrd, and W. Bruce Croft. Clarifying search - a user-interface framework for text searches. *D-Lib Magazine*, 3(1):1–15, jan 1997.

[29] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[30] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[31] Anastasios Tombros and Mark Sanderson. Advantages of query biased summaries in information retrieval. In *SIGIR 98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*, pages 2–10. ACM, 1998.

[32] Mark G. J. van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

[33] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[34] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer.