Plants Search Engine

Purnachandra Kanagala

Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Computer Sciences Commons

Recommended Citation
Kanagala, Purnachandra, "Plants Search Engine" (2012). All Graduate Plan B and other Reports. 111.
https://digitalcommons.usu.edu/gradreports/111
Plants Search Engine

by

Purnachandra Kanagala

A report submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Approved:

_______________________  _______________________
Curtis Dyreson           Dan Watson
Major Professor          Committee Member

_______________________
Vladimir Kulyukin
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2012
ABSTRACT

This project presents a Web search engine for North American plant species that provides easy and fast access to plant information from multiple data sources. This project’s overarching goal is to integrate plant information from different sources under a single interface to allow any user to search said information from a universal portal rather than searching in several places. The user interface was designed and customized to help the user search for anything related to North American plants and get the required information within few clicks with the help of the faceted browsing. The user experience is more enhanced by sorting the plants based on their geographical location with the help of Google maps.

(69 pages)
ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Curtis Dyreson. His support and advice made successful completion of this project possible. It was knowledge and expertise in databases that inspired me to take on this project. I would also like to thank my committee members, Dr. Dan Watson and Dr. Vladimir Kulyukin, for their support and valuable suggestions.

Purnachandra Kanagala
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td><strong>CHAPTER 1</strong></td>
<td></td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>8</td>
</tr>
<tr>
<td>1.1. NEED FOR A PLANTS SEARCH ENGINE</td>
<td>8</td>
</tr>
<tr>
<td>1.2. OBJECTIVES</td>
<td>9</td>
</tr>
<tr>
<td>1.3. REPORT OUTLINE</td>
<td>10</td>
</tr>
<tr>
<td>1.4. UNDERLYING SEARCH ENGINE</td>
<td>10</td>
</tr>
<tr>
<td>1.5. CRAWLING WEBPAGES</td>
<td>12</td>
</tr>
<tr>
<td>1.6. SEARCH</td>
<td>13</td>
</tr>
<tr>
<td>1.7. USER WALK THROUGH</td>
<td>15</td>
</tr>
<tr>
<td><strong>CHAPTER 2</strong></td>
<td>26</td>
</tr>
<tr>
<td>SYSTEM ANALYSIS</td>
<td>26</td>
</tr>
<tr>
<td>2.1. OVERVIEW</td>
<td>26</td>
</tr>
<tr>
<td>2.2. FUNCTIONAL REQUIREMENTS</td>
<td>27</td>
</tr>
<tr>
<td>2.2.1 Basic Search</td>
<td>27</td>
</tr>
<tr>
<td>2.2.2 Range Search</td>
<td>27</td>
</tr>
<tr>
<td>2.2.3 Faceted Search</td>
<td>27</td>
</tr>
<tr>
<td>2.2.4 Location-Based Search</td>
<td>28</td>
</tr>
<tr>
<td>2.2.5 Fetching Data</td>
<td>28</td>
</tr>
<tr>
<td>2.3. NON-FUNCTIONAL REQUIREMENTS</td>
<td>29</td>
</tr>
<tr>
<td>2.3.1 Operating System</td>
<td>29</td>
</tr>
<tr>
<td>2.3.2 Language and Platform</td>
<td>29</td>
</tr>
<tr>
<td>2.3.3 Web Server and Container</td>
<td>29</td>
</tr>
<tr>
<td>2.3.4 Release Management</td>
<td>29</td>
</tr>
<tr>
<td><strong>CHAPTER 3</strong></td>
<td>30</td>
</tr>
<tr>
<td>CONFIGURING THE SEARCH ENGINE</td>
<td>30</td>
</tr>
<tr>
<td>3.1 OVERVIEW</td>
<td>30</td>
</tr>
<tr>
<td>3.2 LUCENE</td>
<td>30</td>
</tr>
<tr>
<td>3.3 INDEX/DATABASE FOR THE PLANTS SEARCH ENGINE DESIGN</td>
<td>31</td>
</tr>
<tr>
<td>3.4 SCHEMA DESIGN</td>
<td>31</td>
</tr>
<tr>
<td>3.5 FIELDS</td>
<td>32</td>
</tr>
<tr>
<td>3.5.1 Predefined Data Types for Numbers</td>
<td>32</td>
</tr>
<tr>
<td>3.5.2 Field Options</td>
<td>36</td>
</tr>
<tr>
<td>3.5.3 Field Definitions</td>
<td>36</td>
</tr>
<tr>
<td>3.5.4 Copying Fields</td>
<td>37</td>
</tr>
<tr>
<td>3.5.5 The Unique Key</td>
<td>39</td>
</tr>
<tr>
<td><strong>CHAPTER 4</strong></td>
<td>40</td>
</tr>
<tr>
<td>INDEXING DATA</td>
<td>40</td>
</tr>
<tr>
<td>4.1 OVERVIEW</td>
<td>40</td>
</tr>
<tr>
<td>4.2 INDEXING USING DATAIMPORTHANDLER</td>
<td>40</td>
</tr>
<tr>
<td>4.3 INDEXING WITH NUTCH</td>
<td>42</td>
</tr>
<tr>
<td>4.4 COMMIT, OPTIMIZE, AND ROLLBACK</td>
<td>44</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1. Simple keyword search using scientific name .......................................................... 15
Figure 2. Simple keyword search using common name .......................................................... 16
Figure 3. Simple keyword search using family name .............................................................. 16
Figure 4. Search for plants in a particular county ................................................................. 17
Figure 5. Search for family and category ............................................................................... 18
Figure 6. Family and category search, along with query for county ...................................... 19
Figure 7. Range based Query on RootDepthMinimumInches .............................................. 20
Figure 8. Example of field query: HeightMatureinFeet ......................................................... 20
Figure 9. Faceted navigation ............................................................................................... 21
Figure 10. Faceted search with bread crumb navigation ...................................................... 22
Figure 11. Google maps integration with searches ............................................................... 23
Figure 12. Google maps integration with search results ....................................................... 23
Figure 13. Google maps integration with bread crumb faceted navigation ......................... 24
Figure 14. Counties listed in a filtered search ....................................................................... 25
Figure 15. Counties marked in Google maps in a filtered search ........................................ 25
Figure 16. Overview of Solr search engine .......................................................................... 31
Figure 17. Faceted Search in Plants Search ........................................................................... 59
Figure 18. Example search query ......................................................................................... 62
Figure 19. Example of refined search query through faceting ........................................... 63
Figure 20. Limiting and increasing facets viewed ............................................................... 64
Figure 21. Example of faceting with Google maps .............................................................. 65
Figure 22. Text of faceting example ..................................................................................... 66
CHAPTER 1
INTRODUCTION

An organism's *name* is the key element to learning more about it. With the name, reference materials or search engines can be used to gather more information about an organism’s aesthetic and beneficial qualities or find out how it can be successfully managed. Naming an organism also builds knowledge of an ecosystem, for instance in documenting biological diversity and monitoring climate change, as well as human impact on the environment.

1.1. Need for a Plants Search Engine

While of vital importance, figuring out the name of an organism can be time-consuming and require an experienced diagnostician or taxonomist. Consider arthropod identification. It is one of the most difficult disciplines in entomology. There are more than a million arthropod species (Myers, 2008). Few taxonomic experts exist, and of those most are highly specialized (e.g., work on only a few families or genera). An expert identifies an arthropod by observing its morphology, which includes body size, shape, proportions, specific features (e.g., antenna length and shape, presence of leg spines, and eye position), colors and shades, and external body textures. Other data can also play an important role. For instance, spatial data can help to isolate the species that commonly occur in an area. But unless a diagnostician is an expert in a group, s/he can make only a small number of identifications per hour.
Though there are many excellent search engines like Google, Bing, Yahoo, etc., they do not index the internal databases of the USDA plant database, called PLANTS, nor other herbarium websites. Our proposed Plants Search Engine solves this problem by specifically indexing all plant data available and organizes it such that the user can search for plants based on their characteristics and geographical location.

Plants Search collects plants data from various websites like PLANTS. The acronym for Plant List of Accepted Nomenclature, Taxonomy & Symbols, and PLANTS is a database of North American flora. This website provides standardized information about vascular plants, mosses, liverworts, hornworts, and lichens of the U.S. and its territories. PLANTS works with entities such as Flora of North America, International Composite Alliance, Global Gymnosperm Tree of Life Consortium, and others to gather data. PLANTS follow international plant nomenclature standards (ICBN) by linking occurrences with specimens whenever possible.

Common standards for encoding taxonomic descriptions, such as the Description Language for Taxonomy (DELT), exist. So the taxonomic descriptions of many species have also been incorporated into Plants Search Engine.

1.2. Objectives

The goal of the research described in this proposal is to make finding the name of an organism easier, faster, and more enjoyable. The search is directed by taxonomic characters that distinguish taxa. We call the system Plants Search Engine, or Plants Search for short. Plants Search utilizes information about the locality, terrain, and other
geographically determined data to enhance and direct the search. In the development phase, Plants Search Engine focused on USDA plants database, also called as PLANTS.

1.3. Report Outline

This report outlines the step-by-step process of implementing Plants Search Engine. This process includes configuring the Solr search engine by adding configurations, setting up Solr cores for different kinds of data, designing the schema for various Solr cores, indexing the data from various sources, performing a simple keyword search, a faceted search, and distributed search. These search types all search from all the Solr cores and location-based faceting using Google Maps.

1.4. Underlying Search Engine

Plants Search is built upon a Solr search engine for all of its indexing and searching purposes. Solr is an open source enterprise search platform from the Apache Lucene project. Its major features include powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document (e.g., Word, PDF) handling. Thus, it provides distributed search and index replication. Solr is written in Java and runs as a standalone full-text search server within a servlet container such as Apache Tomcat or Jetty in the case of Plants Search. Solr uses the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it easy to use from virtually any programming language. Solr's powerful external configuration allows it to be tailored to almost any type of application.
without Java coding, and it has extensive plugin architecture when more advanced customization is required. To configure Solr, a configuration file must be written specifying the request handlers, specifying the velocity templates for processing the requests and responses from the browser to Solr, specifying the facets, and specifying the components that a Solr search engine should have.

Setting up Plants Search Engine itself involves the process of gathering plant data from data sources, processing them according to stated requirements, indexing them, and performing searches. Data sources are of two types, structured relational databases and pure textual data. Indexing structural data involves normalizing the data according to the requirements and indexing it to Solr using SQL queries.

The next step in the process includes developing the index type and schema for Plants Search. Plants Search uses a single combined index that contains all the data in a single index with a unique plant ID for each record in an index. Designing the schema for the index should be done such that it can accommodate lots of fields for describing plant characteristics, hierarchical classification, and geographical location. Data types for those fields must be chosen in a way that search efficiency is improved. To enhance number-based searches, a Solr predefined field type called TINT is used. TINT is a Solr predefined data type used to store any type of number. Moreover, it enhances the range of searches. It should be noted that for some fields wherein a sort on the numbers is needed, another data type called SINT is used.

Additionally, unlike a generic database, the field type should be specified to the fields in an index. The field type specifies whether the data should be stored or indexed,
or if it is a multivalued field. Field type “stored” means data for this field is eligible for searching, “multivalued” means that the field has more than one value, and “indexed” indicates that the search and sort can be performed on that field. Based on the requirements, different field types for different fields are selected.

1.5. Crawling Webpages

The other type of data that is to be indexed into a Solr search engine is the textual data from data sources like Flora of North America. This tends to be pure textual data from webpages such as that found at the Flora of North America website, http://fna.huh.harvard.edu/. Flora of North America builds upon the cumulative wealth of information acquired since botanical studies began in the United States and Canada more than two centuries ago. Recent research has been integrated with historical studies, so currently Flora of North America (FNA) is a single-source synthesis of North American floristics. FNA has the full support of scientific botanical societies and is the botanical community's vehicle for synthesizing and presenting this information. The Flora of North America Project treats more than 20,000 species of plants native or naturalized in North America north of Mexico, about 7% of the world's total. Both vascular plants and bryophytes are included.

Indexing data from FNA involves crawling their webpages using the Nutch web crawler. Nutch is an open source web crawler written in Java that automatically crawls all the hyperlinks of a given websites. Control over the crawl process is achieved by placing the URL seeds, depth of the links, number of threads, regular expressions to fetch only certain parts of the webpage, and restricting the file types to index by placing a filter on
the file types. Data crawled from the Nutch is in the form of segments, which is then analyzed using stemming and tokenization classes of Solr indexed into a core of Solr search engine.

To include both structured data and non-structured data in the index, Solr cores are used for storing the indexed data having different schemas. Searching is performed on both the cores, using a distributed query, and results are listed together. Each Solr core holds an index and a Solr configuration. Nearly all interactions with Solr are targeted at a specific core.

1.6. Search

The faceted search, which is the Solr feature that sets it apart from other search engines, is employed to enhance the user experience. Faceting is an interaction style wherein users filter a set of items by progressively selecting from only valid values of a faceted classification system. Faceting enhances search results with aggregated information over all the documents found in the search, not the entire index. A faceted search can be implemented by specifying the fields on which faceting should be performed in the Solr core configuration. Another way to do this is by using facet queries, in which a complex query is specified as a facet in the Solr configuration. Facets along with their respective facet counts for a search query appear on the left pane of the webpage. To select a facet, the user clicks on the required facet link inside the pane. New results based on the facet are displayed on the webpage. A user can select n number of facets, and all the selected facets are displayed in bread crumb style at the top of the page. Clicking on a particular facet in the facet bread crumb deselects the facet, and the results
are displayed accordingly. Location facets are also added for location-based searches, wherein the user can facet on the required locations by clicking on the geographic location markers (typically counties) plotted on the Google map.

We implemented a web-based user interface for Plants Search Engine to display the results for the search queries and to handle the responses and requests from a browser to Solr and vice versa. Velocity Template Language and velocity template engine are used for developing the web interfaces. The input parameters from the browser are passed to the Solr by the velocity template engine. Then, Solr performs the search on the index, Solr returns the results to the Velocity template engine, and the template engine processes the response and gives the HTML.

PLANTS data is classified into two types. One is the hierarchical classification of the plant and includes details like kingdom, subkingdom, super-division, division, class, subclass, order, family, genus, and species. The second type of data is characteristic data includes duration, growth habit, native status, active growth period, fall conspicuous, flower color, foliage color, foliage texture, fruit color, fruit conspicuous, growth rate, height, leaf retention, lifespan, shape and orientation, bloom period, etc. Plants Search Engine uses all these data and presents a clean and intuitive interface with the help of faceted search to get the required plant species within a few clicks. Apart from the above two types of data, one other type of data Plants Search uses is the normal geographical location of the plants. As stated above, this information is present in the PLANTS database. With the help of Google maps, the user can select the county of choice and get the list of known plants in that particular county.
As of now, Plants Search Engine includes only the plant species of the United States of America. Additionally, the data that is currently indexed is not complete data. Many plants do not have all the fields completed. Added to that, a lot of other fields like extinct status, pests and diseases information, climatic data, and uses are not included as of now. The PLANTS database is continuously working on collecting and organizing such data. As PLANTS gets the data, Plants Search Engine will index the updates. Also, currently, Plants Search Engine does not index images, or rich text documents like pdfs.

1.7. User Walk Through

The simple keyword search can be performed by entering a keyword relating to a plant like its family name, species, flower color, fruit color, shape, etc.

Suppose the user wants to search with a scientific name keyword; the user enters the keyword into the search box and clicks on submit (Figure 1).

Figure 1. Simple keyword search using scientific name.
Similarly, if the user wants to start a new search with a common name of a plant, the user enters the common name into the search box and submits it to the search engine (Figure 2).

![Search: Saguaro](Figure 2. Simple keyword search using common name.)

A user can search with a family name (Figure 3).

![Search: Cactaceae](Figure 3. Simple keyword search using family name.)
To search for plants known to a particular county, the user enters the state code followed by a hyphen and the county name into the search box (Figure 4).

Figure 4. Search for plants in a particular county.
To search for plants that belong to a family and a category, both the family name and the category name are entered into the search box separated by “and”, which gives the results with union of the two keywords on either side of “and”. (Figure 5) A similar search can be conducted with “or”.

Figure 5. Search for family and category.
To filter the results of the above query with county, the user places an “and” followed by the two letter state code followed by a hyphen and the county name after the family and category. (Figure 6)

Figure 6. Family and category search, along with query for county.
To filter the queries based on certain fields, the syntax would be based on the following example: `<fieldname>:[start range TO end range]`. (Figure 7)

![Figure 7. Range based Query on RootDepthMinimuminInches.](image)

Figure 7 shows another example of a field query called HeightMatureinFeet.

![Figure 8. Example of field query: HeightMatureinFeet.](image)
In such cases wherein the user does not know specific keywords to enter, faceted navigation is very helpful. The left pane on the webpage has the list of facets for various characteristics and classifications. Multiple selections can be made and can be filtered with facets.

There are altogether 37 fields to facet on, and the user can select as many as required to get the desired results. The selected facets are shown as the breadcrumb navigation below the search bar. (Figure 10) Clicking on a facet in that breadcrumb deselects it.
Figure 10. Faceted search with bread crumb navigation.

Instead of entering the state and county name to filter the results of keyword search queries and faceted queries, Google maps integration helps the user to click on the particular county and filter the results. For example, if the user searches for the family name “Rosa”. All the markers on the Google map indicates the counties in which the plant records related to “Rosa” are available. (Figures 11, 12, and 13)
Figure 11. Google maps integration with searches.

Figure 12. Google maps integration with search results.
Figure 13. Google maps integration with bread crumb faceted navigation.

To further filter the results based on a county, the user clicks on the county marker on the Google map. The results are listed (Figure 14), and the counties to which the filtered results belong to are marked on the Google map (Figure 15).
Figure 14. Counties listed in a filtered search.

Figure 15. Counties marked in Google maps in a filtered search.
CHAPTER 2
SYSTEM ANALYSIS

2.1. Overview

To understand the desired functionality of the Plants Search Engine, one must understand the system requirements, i.e., identify who will be viewing plant information, understand why they need/want to view plant information, and understand the various procedures that users need to follow to successfully access such information. The best way to visualize a software system is to first identify different objects associated with the system and then seek to understand how they collaborate with each other to accomplish a task. Defining the tasks or activities of each object of a given system is the most crucial step in identifying the requirements of the system. In other words, identifying the objects and studying their relationships with each other is the key step in analyzing the system and its desired behavior. We follow an approach for analyzing the system in question: it is a given that this system’s basic functionality is directly related to analyzing the activities that a user will carry out for searches efficiently.

This chapter focuses on system analysis, which, as stated above, starts by identifying the requirements of the system and the functionality the system will offer. Also, it is helpful to identify different actors associated with the system, user goals, and actors’ roles in the system, and the activities that the user carries out. The system requirements are systematically documented as the functional requirements and non-functional requirements listed in Sections 3.2 and 3.3, respectively.

Both the functional and non-functional requirements together form a strong basis for testing.
2.2. Functional Requirements

This section discusses the process of requirement collection. The most important aspect of requirement collection is to understand the needs of the Plants Search Engine and what it is trying to accomplish. A systematic analysis of the system resulted in the following functional requirements for Plants Search Engine. Each requirement lists an activity associated with the Plants Search Engine, and is included as a part of the services that the Plants Search Engine offers.

2.2.1 Basic Search
This feature allows a user to search with a keyword on all the data including the characteristics, classifications, as well as the geographical information. A user can search for records by placing the query in the search box.

2.2.2 Range Search
This feature allows the user to search for a plant based on data ranges. The numeric fields on which the user needs to be able to perform range searches include information such as height at base age, height mature, pH values, planting density, precipitation levels, root depth, and number of seeds per pound.

2.2.3 Faceted Search
Faceted search, also called faceted navigation or faceted browsing, is a technique for accessing information organized according to a faceted classification system, thus allowing users to explore a collection of information by applying multiple filters. A faceted classification system classifies each information element along multiple explicit
dimensions, enabling the classifications to be accessed and ordered in multiple ways rather than in a single, pre-determined, taxonomic order.

Plants Search Engine implements faceted searches upon plant classification data, characteristic data, and location data. Classification facets include the kingdom, subkingdom, super-division, division, class, subclass, order, genus, family, and species; characteristic facets like growth habit, active growth period, flower color, flower conspicuous, foliage color, foliage texture, fruit color, lifespan, shape and orientation, and bloom period; and finally, location facets like the cities, counties, and even countries.

2.2.4 Location-Based Search
A location-based search conducted with Google maps enhances the search experience for the user. The user enters a search query in the search box, and Plants Search plots all the geographical locations of the obtained plant records. Google maps API V3 is used to plot all the center points of the counties. When the user clicks on a marker on the map, Plants Search fetches all the plants species present in that geographical location.

2.2.5 Fetching Data
Different Web resources require differing methods of fetching data. For example, Flora of North America does not offer clean, structured data. In such a case, a Web crawler called Apache Nutch is used to fetch all the data in all the links of the website. Next, the raw data is analyzed, processed, and fed to the Lucene index as segments.

On the other hand, sites like the PLANTS database offers structural data wherein a manual process is followed to normalize the data and feed it into a database. Then, with
the help of DataImportHandler provided by Solr, all the data is indexed using SQL queries.

2.3. **Non-Functional Requirements**

This section describes the non-functional requirements for Plants Search Engine.

Plants Search is implemented on top of the Solr search engine which is implemented in Java. Following are the implementation constraints.

2.3.1 **Operating System**

The Plants Search Engine can be developed in either a Windows- or Linux-based operating system.

2.3.2 **Language and Platform**

In order to generate the HTML, Plants Search Engine was implemented in Velocity Template language.

2.3.3 **Web Server and Container**

The Plants Search Engine requires a Jetty Webserver, which is an open source HTTP server for Linux and Windows.

2.3.4 **Release Management**

Once the code is tested on the local environment, we push testing into the development environment, and then to production.
CHAPTER 3
CONFIGURING THE SEARCH ENGINE

3.1 Overview

Before configuring the Solr schema, we need to know about the schema on which Solr is built. The foundation of Solr is based on Apache Lucene's index. This chapter discusses Lucene Vis-a-Vis the index structure and the schema for Plants Search, including definitions for fields, their data types, and their field types.

3.2 Lucene

Apache Lucene is an open source information retrieval software library. It is a high-performance, full-featured text search engine library written entirely in Java. Lucene itself is just an indexing and search library and does not contain crawling and HTML parsing functionality. At the core of Lucene's logical architecture is the idea of a document containing fields of text. This flexibility allows Lucene's API to be independent of the file format. Text from PDFs, HTML, Microsoft Word, and Open Document documents, as well as many others (except images), can all be indexed as long as their textual information can be extracted.

Figure 16 shows the big picture of how various aspects Solr and a search engine working together.
3.3 Index/Database for the Plants Search Engine Design

Because it is a search engine, Plants Search has an index instead of a database which a normal application would have. Plants Search uses a combined index for its schema; it can also be called an aggregate index which is an index that is conceptually like a single-table relational database schema, thus sharing similarities with some NoSQL databases. It takes different fields for the different document types, and uses a field to discriminate between the types. An identifier field which is unique is shared across all documents in this index, no matter what the type is.

3.4 Schema Design

A key thing to come to grips with is that the queries to support in the Plants Search completely drive the schema design. Conversely, relational databases typically use standard third normal form decomposition of the data, largely because they have...
strong relational-join support. Since queries drive the Solr schema design, all the data needed to match a document, i.e., the criteria, must be in a document matched, not in a related one. To satisfy said requirement, data that would otherwise exist in one place is copied into related documents that need it to support a search.

3.5 Fields

The first section of the schema is the definition of the field types. In other words, these are the data types. This section is enclosed in the<types/> element. The field types declare the types of fields, such as Boolean, numbers, dates, and various text flavors. They are referenced later by the field definitions under the<fields/> element.

Here is the field type for a Boolean:

```xml
<fieldType name="boolean" class="solr.BoolField" />
<field name="Symbol" type="text_general" indexed="true" stored="true" required="true"/>
```

A field type has a unique name and is implemented by a Java class specified by the class attribute.

3.5.1 Predefined Data Types for Numbers

**Tint.** This field is a Solr TrieIntField with a precision step greater than zero. This field type has unbeatable performance on numeric range queries at query time at the expense of a little more disk and indexing time cost.

```xml
<field name="ITISTSN" type="tint" indexed="true" stored="true"/>
```
**Tfloat.** The Tfloat field is used for storing the floating values and best supports the range facets and sorting.

The following fields are of the tfloat type with the parameters *indexed* and *stored* true, as a search is performed on the following fields:

```xml
<field name="pHMinimum" type="tfloat" indexed="true" stored="true" />
<field name="pHMaximum" type="tfloat" indexed="true" stored="true" />
<field name="PlantingDensityperAcreMinimum" type="tfloat" indexed="true" stored="true" />
<field name="PlantingDensityperAcreMaximum" type="tfloat" indexed="true" stored="true" />
<field name="PrecipitationMinimum" type="tfloat" indexed="true" stored="true" />
<field name="PrecipitationMaximum" type="tfloat" indexed="true" stored="true" />
<field name="RootDepthMinimuminInches" type="tfloat" indexed="true" stored="true" />
<field name="SeedsperPound" type="tfloat" indexed="true" stored="true" />
<field name="HeightatBaseAgeMaximuminFeet" type="tfloat" indexed="true" stored="true" />
<field name="HeightMatureinFeet" type="tfloat" indexed="true" stored="true" />
```
**text_general.** The text_general field is used for storing text, usually multiple words or tokens and best supports the full text search queries.

The following fields are of the text_general type with the parameters *indexed* and *stored* true, as a search is performed on the following fields:

```xml
<field name="Symbol" type="text_general" indexed="true" stored="true" required="true"/>
<field name="ScientificName" type="text_general" indexed="true" stored="true"/>
<field name="HybridGenusIndicator" type="text_general" indexed="true" stored="true"/>
<field name="HybridSpeciesIndicator" type="text_general" indexed="true" stored="true"/>
<field name="Species" type="text_general" indexed="text_general" stored="true"/>
<field name="SubspeciesPrefix" type="text_general" indexed="true" stored="true"/>
<field name="HybridSubspeciesIndicator" type="text_general" indexed="true" stored="true"/>
<field name="Subspecies" type="text_general" indexed="true" stored="true"/>
<field name="VarietyPrefix" type="text_general" indexed="true" stored="true"/>
<field name="HybridVarietyIndicator" type="text_general" indexed="true"
```
Comma_seperating. The Comma_seperating, field type as the name suggests is used for storing comma separated words or tokens. This type is normally used for the fields with multiple values and best supports the full text search queries.

The following field is of the Comma_seperating type with the parameters indexed and stored true, as a search is performed on the following fields:

```xml
<field name="statesandcounties" type="comma_seperating" indexed="true"
stored="true" multiValued="true"/>
```

String. The String field is used for storing strings as a whole.

The following field is of the string type with the parameters stored true, as a search is performed on the following fields:

```xml
<field name="URL" type="string" stored="true"/>
```
3.5.2 Field Options

Indexed. This option indicates that this data can be searched and sorted. The only purpose a non-indexed field has is that it is returned in search results.

```xml
<field name="CommonName" type="text_general" indexed="true" stored="true" />
```

Stored. This option indicates that the field is eligible for inclusion in search results. Usually fields are stored, but sometimes the same data is copied into multiple fields that are indexed differently; thus, the redundant fields would not be marked as stored.

```xml
<field name="URL" type="string" stored="true" />
```

Multivalued. This field is used when a field can contain more than one value. Order is maintained from what is supplied at index-time.

```xml
<field name="statesandcounties" type="comma_seperating" indexed="true" stored="true" multiValued="true" />
```

3.5.3 Field Definitions

The definitions of the fields in the schema are located within the `<fields/>` element. In addition to the field options defined above, a field has the attributes listed below.

Let us look at the symbol field in the schema of plants search engine as an example:

```xml
<field name="Symbol" type="text_general" indexed="true" stored="true"
    required="true" />
```

Name. This uniquely identifies the field. There are neither restrictions on the characters used nor any words to avoid, except for score. Symbol is the unique name to the whole schema.
**Type.** This is a reference to one of the field types defined earlier in the schema. Type is the data type of the data that it holds; since the symbols are alphabetic words text_general is chosen.

**Required.** For some fields wherein the values cannot be null, this is set to true. Indexing will fail for the documents with null value in those fields. Since the symbol is similar to the primary key in relational databases, it is made a required field and hence required="true".

### 3.5.4 Copying Fields

Closely related to the field definitions are copyField directives. A copyField directive copies one or more input field values to another during indexing. A copyField directive looks like this:

```
<copyField source="r_name" dest="r_name_sort" maxChars="20" />
```

This directive is useful when a value needs to be copied to additional field(s) to be indexed differently. For example, sorting and faceting require a single indexed value. Another is a common technique in search systems in which many fields are copied to a common field that is indexed without norms and not stored. This permits searches, which would otherwise search many fields, to search one instead, thereby drastically improving performance at the expense of reducing score quality. This technique is usually complemented by searching some additional fields with higher boosts. At index-time, each supplied field of input data has its name compared against the source attribute of all copyField directives. The source attribute might include a * wildcard, so it is possible the
input might match more than one copyField. If a wildcard is used in the destination, it must refer to a dynamic field. Furthermore, the source must also include a wildcard; otherwise, a wildcard in the destination is an error. A match against a copyField has the effect of the input value being duplicated by using the field name of the dest attribute of the directive. If maxChars is optionally specified, the copy is truncated to this many characters. The duplicate does not replace any existing values that might be going to the field, so one must be sure to mark the destination field as multivalued if needed. For Plants Search Engine, all the fields on which a default search should be performed are copied into a destination field called simpleSearch.

    <copyField source="Symbol" dest="simpleSearch 
    <copyField source="ScientificName" dest="simpleSearch 
    <copyField source="Species" dest="simpleSearch 
    <copyField source="CommonName" dest="simpleSearch 
    <copyField source="PLANTSFloristicArea" dest="simpleSearch 
    <copyField source="Category" dest="simpleSearch 
    <copyField source="Genus" dest="simpleSearch 
    <copyField source="Family" dest="simpleSearch 
    <copyField source="FamilyCommonName" dest="simpleSearch 
    <copyField source="Order" dest="simpleSearch 
    <copyField source="Kingdom" dest="simpleSearch 
    <copyField source="statesandcounties" dest="simpleSearch 


3.5.5 The Unique Key

The `<uniqueKey>` declaration specifying which field uniquely identifies each document. Plants Search Engine schema has symbol as the unique id:

```xml
<uniqueKey>Symbol</uniqueKey>
```
CHAPTER 4
INDEXING DATA

4.1 Overview

The process of getting the data into the system is called indexing or importing. Data from PLANTS database is downloaded as CSV files and added to the local SQL database. Later, using Solr’s DataImportHandler, data from the SQL database is added to Solr’s index. There are two types of data that we indexed into the Plants Search Engine. One is structured data from PLANTS database and the other is textual data extracted from the Flora of North America’s website by crawling with Nutch.

4.2 Indexing Using DataImportHandler

The PLANTS database provides standardized information about vascular plants, mosses, liverworts, hornworts, and lichens of the U.S. and its territories. It includes names, plant symbols, checklists, distributional data, species abstracts, characteristics, images, crop information, automated tools, onward Web links, and references. This information primarily promotes land conservation in the United States and its territories, but academic, educational, and general use is encouraged. PLANTS reduce government spending by minimizing duplication and making information exchange possible across agencies and disciplines.

PLANTS is a collaborative effort of the USDA NRCS National Plant Data Team (NPDT), the USDA NRCS Information Technology Center (ITC), The USDA
National Information Technology Center (NITC), and many other partners. Much of the PLANTS data and design was developed at NPDT, and the Web application is programmed at ITC and NITC and served through the USDA Web form.

Data from PLANTS is downloaded in comma-separated valued files and put into a local MySQL database for indexing into Plants Search Engine. The PLANTS database has about 48,330 accepted taxa within the United States and its minor islands. The data import handler framework of Solr includes a module for importing data known as the DataImportHandler (DIH in short). It is a data processing pipeline built specifically for Solr.

DIH imports data from databases through Java database connectivity (JDBC). It supports importing only changed records, assuming a last-updated date on imports data from a URL (HTTP GET). DIH imports data from files (that is, it crawls files). It imports e-mail from an IMAP server, including attachments which it supports by combining data from different sources. It extracts text and metadata from rich document formats. It applies XSLT transformations and XPath extraction on XML data. Finally, DIH includes a diagnostic/development tool.

DIH is not considered a core part of Solr, so its Java JAR files must be added to the Solr setup for use.

The following are XML snippets showing the DIH configurations.

**Solr-config.xml**

```xml
<requestHandler name="/dataimport"
    class="org.apache.solr.handler.dataimport.DataImportHandler"/>
```
<lst name="defaults">
  <str name="config">data-config.xml</str>
</lst>

</requestHandler>

**Data-config.xml**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dataConfig>
  <dataSource type="JdbcDataSource" driver="com.mysql.jdbc.Driver"
               url="jdbc:mysql://localhost/plants_db" user="root" password="root" autocommit="true"
               readonly="true">
  </dataSource>
  <document>
    <entity name="plants_core_data" query="query">
      <field column="p1.Symbol" name="Symbol"/>
    </entity>
  </document>
</dataConfig>
```

### 4.3 Indexing with Nutch

We use Nutch to build an open source Web search engine based on Lucene for the search and index component. The crawl command for Nutch is:

```
bin/nutch crawl urls -dir crawl1 -depth 3 -topN 5
```

- `-dir dir` specifies the name of the directory to put the crawl in.
- `-threads threads` specifies the number of threads that will fetch in parallel.
- `-depth depth` indicates the link depth from the root page that should be crawled.
• -topN N determines the maximum number of pages that will be retrieved at each level up to the depth.

The following command is used to crawl the webpages of Flora of North America:

```
$ bin/nutch crawl urls -dir crawl -depth 10000 -topN 10000 -threads 100 -solr h ttp://localhost:8983/solr/Nutch
```

The seed specified in the URLs is [http://fna.huh.harvard.edu/families](http://fna.huh.harvard.edu/families).

The following URLs are added in the regex URL filter file to crawl upon:

```
+^http://([A-Za-z0-9]*\.)*efloras\.org\/[florataxon].aspx\/?flora_id=1\&(.+)
+^http://fna\.huh\.harvard\.edu\/[families
+^http://www\\.tropicos\.org\/[namesearch].aspx\/?name=*
+^http://mobot\\.mobot\.org\/[cgi]-bin\/[search]_pick\/?name=*
+^http://www\\.tropicos\.org\/[Name]/*
```

It took eight hours to crawl all the links related to the Flora of North America, and it fetched almost 65,000 records of flora data.

To index the fetched records from Nutch to Solr, the following command is used:

```
bin/nutch solrindex http://127.0.0.1:8983/solr/Nutch crawldb -linkdb crawldb/linkdb crawldb/segments/*
```

Solrindex is an alias for org.apache.nutch.indexer.solr.SolrIndexer,

[http://127.0.0.1:8983/solr/Nutch](http://127.0.0.1:8983/solr/Nutch) is the solr URL, crawldb is the directory for the crawled data, -linkdb crawldb/linkdb specifies the directory of the links that are crawled, and crawldb/segments/* species the list of segments in which the actual crawl data is stored.
Upon successfully executing this command, all the crawl data is sent to Solr for indexing. The schema and the configurations are generated automatically by Solr and made ready for searching.

4.4 Commit, Optimize, and Rollback

Data sent to Solr is not immediately searchable, nor do deletions take immediate effect. Like a database, changes must be committed first. After bulk loading data, a final commit is issued at the end.

Lucene's index is internally composed of one or more segments. Flushing a buffer of indexed documents to a disk creates a new segment. Deletes get recorded in another file, but they go to the disk, too. Sometimes, after a new segment is written, Lucene will merge some of them together. When Lucene has just one segment, it is in an optimized state. The more segments there are, the more query performance degrades. Of course, optimizing an index comes at a cost; the larger the index is, the longer it takes to optimize. Finally, an optimize command implies commit semantics. The optimize command is placed after the commit command. It is recommended to explicitly optimize the index at an opportune time such as after a bulk load of data and/or a daily interval in off-peak hours, if there are sporadic updates to the index. Both commit and optimize commands take two additional Boolean options that default to true:

<optimize waitFlush="true" waitSearcher="true"/> If these are set to false, then commit and optimize commands return immediately, even though the operation has not actually finished yet.
### 4.5 Delta Imports

The DIH supports what it calls a delta import, which is a mode of operation in which only data that has changed since the last import is retrieved. A delta import is only supported by the `SqlEntityProcessor`, and it assumes that the data is time-stamped. It uses a `deltaImportQuery` and `deltaQuery` pair of attributes on the entity, and a delta-import command. A time stamp should be introduced in the SQL's WHERE clause using variable substitution, along with another check if the clean parameter was given to the DIH. Doing so controls whether or not a delta or full import should happen. Here is a concise definition on a fictitious schema and data set showing the relevant WHERE clause:

```xml
<entity name="item" pk="ID" query="SELECT * FROM item WHERE
'${dataimporter.request.clean}' != 'false' OR last_modified >
'${dataimporter.last_index_time}'">
```
CHAPTER 5
SEARCH

5.1 Overview

For obvious reasons, the value of a search engine is in its search capability. This chapter discusses Plants Search’s request handlers, which process the searches, the query parameters to perform the searches, Solr’s Lucene query syntax, and filtering and sorting the data. This discussion demonstrates the viability of Plants Search’s search capability.

The web application uses HTTP to interact with Solr with the help of Solr APIs. The search form submits the form using HTTP, essentially resulting in the browser loading a new URL with the form elements becoming part of the URL’s query string. An example of a solr URL looks like this:

http://localhost:8983/solr/mmirror/plants?q=aegyptiaca&fq=Family%3A%22asteraceae%22

/solr/ is the web application context wherein Solr is installed on the Java servlet engine. After the web application context is a reference to the Solr core named mmirror. The “/plants” is a reference to the Solr request handler. Following the “/plants?” are the set of unordered URL parameters, also known as query parameters in the context of searching. The format of this part of the URL is a separating set of unordered name=value pairs. Text in the URL must be UTF-8 encoded then URL-escaped so that the URL complies with its specification.

In the URL above, Solr interpreted the %3A as a colon and %2C as a comma. The most common escaped character in URLs is a space, which is escaped as either + or %20.
### 5.2 Request Handlers

Searching Solr and most other interactions with Solr, including indexing for that matter, is processed by what Solr calls a request handler. Request handlers are configured in the solrconfig.xml file.

If nothing about the request handler is specified in the URL, it will be redirected to the default request handler. Here is how the default request handler is configured:

```xml
<requestHandler name="standard" class="solr.SearchHandler" default="true">
  <!-- default values for query parameters -->
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="fl">*</str>
    <str name="version">2.1</str>
  </lst>
</requestHandler>
```

The request handlers that perform searches allow configuration of two things: one is establishing default parameters and making some unchangeable, while the other is registering Solr search components such as faceting and highlighting. The configuration of the “plants” request handler is as follows:

```xml
<requestHandler name="plants" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">none</str>
  </lst>
</requestHandler>
```
<int name="rows">20</int> </lst>
<lst name="appends"> <str name="facet">false</str> </lst>
</requestHandler>

5.3 Query Parameters

There are a great number of request parameters for configuring searches, especially when considering components like faceting and highlighting.

**Search Criteria Related Parameters.** The parameters affecting the query are as follows:

- **q:** The user query or just "query" for short. This typically originates directly from user input.

- **fq:** A filter query that limits the scope of the user query, similar to a WHERE clause in SQL. Unlike the q parameter, this has no effect on scoring. This parameter can be repeated as desired. Filtering is described later in the chapter.

- **qt:** A reference to the query type, more commonly known as the request handler, described earlier.

**Result Pagination Related Parameters.** A query could match any number of the documents in the index, perhaps even all of them, such as in our first example of `*:*`. Solr does NOT generally return all the documents. Instead, the user should indicate to Solr with the start and rows parameters to return a contiguous series of them. The start and rows parameters are explained below:

- **start:** (default: 0) This is the zero-based index of the first document to be returned from the result set. In other words, this is the number of documents to skip from the beginning
of the search results. If this number exceeds the result count, it will simply return no documents, but it is not considered an error.

**rows**: (default: 10) This is the number of documents to be returned in the response XML starting at index start. Fewer rows will be returned if there are not enough matching documents. This number is basically the number of results displayed at a time on the search user interface.

**Output Related Parameters.**

The output related parameters are explained below:

**fl**: This is the field list, separated by commas and/or spaces. These fields are to be returned in the response. Use * to refer to all of the fields but not the score.

**sort**: A comma-separated field listing to sort on, with a directionality specifier of asc or desc after each field, for example, r_name asc, score desc.

**Diagnostic Related Parameters.** These diagnostic parameters are helpful during development with Solr.

**Indent**: A Boolean option that will indent the output to make it easier to read. It works for most of the response formats.

**debugQuery**: If true, then following the search results is <lst name="debug"> with diagnostic information. It contains voluminous information about the parsed query string, how the scores were computed, and millisecond timings for all of the Solr components to perform their part of the processing such as faceting. The admin user may need to use the View Source function of the browser to preserve the formatting used in the score computation section.
**explainOther**: If the user wants to determine why a particular document was not matched by the query or why it did not scored high enough, a query is put for this value, such as id:"Group:Dicot", and debugQuery's output will be sure to include the first document matching this query in its output.

**echoHandler**: If true, this emits the Java class name identifying the request handler.

**echoParams**: Controls if any query parameters are returned in the response header, as seen verbatim earlier. This is for debugging URL encoding issues or for verifying the complete set of parameters in effect, taking into consideration those defined in the request handler. Specifying none disables this, which is appropriate for production real-world use. The standard request handler is configured for this to be explicit by default, which means to list those parameters explicitly mentioned in the URL. Finally, all can be used to include those parameters configured in the request handler in addition to those in the URL. Finally, there is another parameter not easily categorized above called *timeAllowed* in which a time limit in milliseconds is specified for a query to take before it is interrupted and intermediate results are returned. Long-running queries should be very rare, and this allows capping them so that they do not over-burden the production server.

### 5.4 Query Syntax (the Lucene Query Parser)

The query parser used for this search engine is called Lucene. It is based on Lucene's old syntax with a few additions explicitly pointed out in this section. The Lucene query parser does have a couple query parameters that can be set. Usually these are not specified as Lucene is rarely used for the user query and because Lucene's query syntax is easily made explicit to not need these options.
**q.op:** The default query operator, either AND or OR to signify if all of the search terms or just one of the search terms need to match. If this is not present, the default is specified in schema.xml near the bottom in the defaultOperator attribute. If that is not specified, the default is OR.

**df:** The default field to be searched by the user query. If this is not specified, the default is specified in schema.xml near the bottom in the<defaultSearchField> element. If such specification does not occur, the search will be an error.

### 5.5 Matching all the Documents

The following syntax: `*:*` matches all the documents indexed into the search engine. The default search for this search engine is matching all the documents.

### 5.6 Mandatory, Prohibited, and Optional Clauses

Lucene has a unique way of combining multiple clauses in a query string. It is tempting to think of this as a mundane detail common to Boolean operations in programming languages, but Lucene does not quite work that way. A query expression is decomposed into a set of unordered clauses of three types:

**Mandatory.** A clause can be made mandatory by adding `+` before the search query:

Example: `+ shrubby Indian mallow`.

This matches only plants with names shrubby Indian mallow.

**Prohibited.** A clause can be prohibited by adding a minus before it as in the following:

-`shrubby Indian mallow`

This matches all plants except those with shrubby Indian mallow.
**Optional.** If the query expression contains at least one mandatory clause, any optional clause is just that—optional. This notion may seem pointless, but it serves a useful function in scoring documents that match more. If the query expression does not contain any mandatory clauses, at least one of the optional clauses must match.

### 5.7 Boolean Operators

The Boolean operators AND, OR, and NOT can be used as an alternative syntax to arrive at the same set of mandatory, optional, and prohibited clauses mentioned above.

The search query “Dicot AND Erigeron” fetches the plant records with the category of the plant as Dicot and the Genus Erigeron, or it can be also given as “+Dicot +Erigeron”. Similarly, the search query “Dicot OR Erigeron” fetches the plant records with an “OR” clause between the category of the plant as Dicot and the Genus Erigeron. To include Dicot and to not include Erigeron, the query would be “Dicot NOT Erigeron”.

### 5.8 Sub-queries

Parentheses are used to compose a query of smaller queries, referred to as sub-queries or nested queries as shown in the following example:

(Dicot AND Erigeron) OR (Perennial AND Herb)

The preceding sub-query is interpreted as documents that must have a name with either Dicot or Erigeron and either Perennial or Herb in its name. So, if there is a plant with a Dicot category and perennial growth Habit, it would match.

### 5.9 Phrase Queries and Term Proximity
A clause may be a phrase query: a contiguous series of words to be matched in that order. In the previous examples, we search for text containing multiple words like Flowers and Yellow, but say we want to match Flowers Yellow (that is, the two words adjacent to each other in that order). This further constrains the query. Double quotes are used to indicate a phrase query, as shown in the following code: "Flowers Yellow". Related to phrase queries is the notion of the term proximity, THAT IS, the slop factor or a near query. In our previous example, if we want to permit these words to be separated by no more than say three words, we could do this: "Flowers Yellow"~3. For the Plants Search Engine data set, this is probably of little use. For larger text fields, this can be useful in improving search relevance. The dismax query parser, which is described in the next chapter, can automatically turn a user's query into a phrase query with a configured slop.

5.10 Wildcard Queries

A plain keyword search looks in the index for an exact match, subsequent to text analysis processing on both the query and input document text (for example, tokenization, lowercasing). But sometimes, a partial match is expressed using wildcards. Text analysis is not performed on the search word containing the wildcard, not even lowercasing. So, to find a word starting with abutilon, tabu* is required instead of Abu*, assuming the index side of the field's type includes lowercasing.

5.11 Fuzzy Queries
Fuzzy queries are useful when the search term need not be an exact match, but the closer the better. The fewer the number of character insertions, deletions, or exchanges relative to the search term length, the better the score. The algorithm used is known as the Levenshtein distance algorithm, also known as the edit distance. Fuzzy queries have the same need to lowercase and to avoid stemming as wildcard queries do. For example:

Mallow~ Notice the tilde character at the end. Without this notation, simply mallow would match only four documents because only that many plant names contain that word. However, Mallow~ matched a huge number of documents. The proximity threshold can be modified, which is a number between 0 and 1, defaulting to 0.5. For instance, changing the proximity to a more mallow~0.7 resulted in 25 matched documents, and it took 174 milliseconds. As with wildcard searches, fuzzy queries also need lowercasing in the query string.

5.12 Range queries

For most numbers in the Plants Search Engine schema, we only have identifiers, so it made sense to use the plain long field type, but there other fields exist. For the length of the RootDepthMinimuminInches duration, we could do a query such as the following to find all of the plant records with root depth longer than 2 inches:

RootDepthMinimuminInches: [2 TO *].

Similarly done on the following fields

Height at Base Age, Height Mature, pH values, Planting Density, Precipitation levels, root depth and number of Seeds per pound.
5.13 **Score Boosting**

The degree to which a clause in the query string contributes to the ultimate relevancy score can be modified by adding a multiplier is called boosting. A value between 0 and 1 reduces the score, and numbers greater than 1 increase it. For example, the search query “FamilyName: Asteraceae^2 OR Fabaceae” fetches the plants with the Family name Asteraceae with double the importance, while the plants with the Family name Fabaceae are of normal importance. The search query “FamilyName: Asteraceae^0.7 OR Fabaceae” fetches data with Fabaceae being considered of more importance and Asteraceae considered of less importance.

5.14 **Existence (and Non-existence) Queries**

To find all the documents within the index, the query is “FamilyName: [* TO *]”, and to negate the above query, a hyphen should be placed before the FamilyName, i.e. “-FamilyName: [* TO *]” which gets the results with no Family Name.

5.15 **Filtering**

Along with the q parameter, additional filter queries can also be applied to filter the search results. Filter queries do not affect scoring, unlike the user query. Filters can be applies by simply using the \textit{fq} parameter. This parameter can be added multiple times for additional filters. A document must match all filter queries and the user query in order for it to be in the results. As an example, say we wanted to make a search form that lets the user search for Genus, not individual plants, and those with the group Dicot and the family Malvaceae.
Let's also say that the user's query string is Indian mallow. In the index, group is dicot, family is Malvaceae. Therefore, a query would be:

\[ q=\text{Indian Mallow}\&fq=-\text{family}\%3A\text{Malvaceae}\&fq=\text{group}\%3Adicot \]

Filter queries have some tangential benefits. For example, they improve performance, because each filter query is cached in Solr's filter cache and can be applied extremely quickly.

5.16 Searching Multiple Fields

The qf parameter is used to tell the dismax query parser which fields are more important and which are less. Handlers, the query parameters can be specified in the URL or in the request handler configuration in solrconfig.xml as follows:

\[ <\text{str name}="qf">\text{CommonName Family Name}^0.8 \text{FamilyCommonName}^0.4</\text{str}> \]

5.17 Sorting

The sorting specification is specified with the sort query parameter. The default is score desc. Score is not a field but a special reference to a relevancy number. desc means descending order, asc is for ascending order. For example, sort=FamilyName desc, score desc will fetch the search results sorted with the FamilyName as well as score.

5.18 Searching with Google Maps

Google maps API v3 is used in Plants Search Engine to plot the counties and cities in which plants are available. Latitude and longitude information about the counties is stored in a JavaScript file in the format of an array. The elements in the array again are
arrays with the first element being the county name and the second element and the third elements being the latitude and the longitude of the center of the county, respectively.

Plants Search Engine generally pulls the plants based on the searches entered in the search box or from the facet selections; along with the search results, it also fetches the county facet information which is stored in the index. This facet information and the JavaScript file which maps the geographic co-ordinate helps in plotting the markers in the Google map. Again, the object called `latlngbounds` is used to contain all the points in a rectangle which makes it easy for the user to get all the plots in one window; otherwise, all the points are scattered, and the user needs to zoom in and out to see all the locations in one window.

The user can click on the marker points to get plants belonging to that geographical location. Again, the faceting searching works here, too. Hence, the user can have multiple options to select from like the classification, characteristics, and the Google maps. Within a few clicks, the required record can be pulled and within very little time.
CHAPTER 6

FACETING

6.1 Overview

Faceting is an interaction style wherein users filter a set of items by progressively selecting from only valid values of a faceted classification system. Faceting enhances search results with aggregated information over all the documents found in the search, not the entire index.

Faceting in the context of the user experience is often referred to as faceted navigation, faceted search, faceted browsing, guided navigation, or parametric search. The facets are typically displayed with clickable links that apply Solr filter queries to a subsequent search.

If we revisit the comparison of search technology to databases, then faceting is more or less analogous to SQL's GROUP BY feature on a column with count(*). However, in Solr, facet processing is performed subsequent to an existing search as part of a single request-response, with both the primary search results and the faceting results coming back together. In SQL, one would need to perform a series of separate queries to get the same information. Furthermore, faceting works so fast that its search response time overhead is almost always negligible.

In Figure 17, faceted navigation of Plants Search is shown. On the left pane, facets like the family, subspecies, variety, category, genus, family, subclass, and class are listed. To search for a plant with the family as Acanthaceae, subspecies as caroliniensis,
and variety as caroliniensis, the user clicks on the family facet Acanthaceae, subspecies facet caroliniensis, and the variety facet caroliniensis to obtain the result. The clicked facets are shown as breadcrumbs on top of the results.

> Family:"acanthaceae" > Subspecies:"caroliniensis" > Variety:"caroliniensis"

![Facet Search in Plants Search](image)

**Figure 17. Faceted Search in Plants Search.**

### 6.2 Field Requirements

The principal requirement of a field that will be faceted on is that it must be indexed; it does not have to be stored. And for text fields, tokenization is usually undesirable. Only on structured data faceting is possible; thus, faceting is possible only on the core which has structured data. Textual data cannot be faceted.

### 6.3 Field Value Faceting

Field value faceting is the most common type of faceting. The example in Figure 17 demonstrates this type of action. Solr, in essence, iterates over all of the indexed terms
for the field and tallies a count for the number of searched documents that have the term. Sophisticated algorithms and caching makes this so fast that its overhead is usually negligible. The following are the request parameters for using it.

**facet.field** To enable faceting for the Solr search engine, the parameter facet is set to the following:

```xml
<str name="facet">on</str>
```

The following are the fields that are set for faceting the plants search engine

```xml
<str name="facet.field">statesandcounties</str>
<str name="facet.field">Family</str>
<str name="facet.field">Subspecies</str>
<str name="facet.field">Variety</str>
<str name="facet.field">Category</str>
<str name="facet.field">Genus</str>
<str name="facet.field">Family</str>
<str name="facet.field">SubClass</str>
<str name="facet.field">Class</str>
<str name="facet.field">SubDivision</str>
<str name="facet.field">Division</str>
<str name="facet.field">SuperDivision</str>
<str name="facet.field">SubKingdom</str>
<str name="facet.field">Kingdom</str>
<str name="facet.field">Duration</str>
```
<str name="facet.field">GrowthHabit</str>
<str name="facet.field">Invasive</str>
<str name="facet.field">ActiveGrowthPeriod</str>
<str name="facet.field">FlowerColor</str>
<str name="facet.field">FlowerConspicuous</str>
<str name="facet.field">FoliageColor</str>
<str name="facet.field">FoliageTexture</str>
<str name="facet.field">FruitColor</str>
<str name="facet.field">LeafRetention</str>
<str name="facet.field">Lifespan</str>
<str name="facet.field">ShapeandOrientation</str>
<str name="facet.field">MoistureUse</str>
<str name="facet.field">CommercialAvailability</str>
<str name="facet.field">ChristmasTreeProduct</str>
<str name="facet.field">PalatableHuman</str>
<str name="facet.field">ProteinPotential</str>
<str name="facet.field">VegetativeSpreadRate</str>
<str name="facet.field">SeedlingVigor</str>
<str name="facet.field">SeedSpreadRate</str>
<str name="facet.field">BloomPeriod</str>
<str name="facet.field">KnownAllelopath</str>
**Facet.sort** This parameter is set to either count to sort the facet values by descending totals or to index to sort lexicographically. For Plants Search Engine, the facet.sort is set to count because the count determines the plants classification more easily than the lexicographical sort, for example, `<str name="facet.sort">count</str>`.

![Search: rose](image)

Figure 18. Example search query.

In Figure 18 a search query for “rose” is made. The first result on the right pane returned is a plant with its symbol ROSE, but it might not be the result that the user wants. So, the facets on the left pane helps the user by showing the family facet Rosacea with highest number of counts, which means that more relevant search results can be found in that facet. Clicking the Rosacea facet can get all the plants belonging to the
rosacea family, as shown in Figure 19. Thus, sorting the facets based on the counts helps in these kinds of occasions.

> Family: "rosaceae"  

**Figure 19. Example of refined search query through faceting.**

**facet.limit** This request parameter limits the number of facet values returned in the search results of a field. To display all the facet values, it should be set to -1. In the case of Plants Search Engine, the value is set to -1. For example, while displaying the county facets, all the counties should be mapped in the Google maps to enable the user facet on the counties. Sometimes it does not make sense to have a large number of facets in the response page. So, a functionality to limit the large number of facets is provided. This functionality could show only the top ten facets, for example. And, the user could see more facets by clicking on Show more or hide by clicking on Show Less.
6.4 Faceting with Google Maps

This feature enables the user to facet on geographical locations by clicking on those locations on a Google Map. The user first makes a search query; next, the required results are obtained by clicking the facets. The corresponding locations of all the plants results obtained are plotted on a Google Map. Figure 21 shows a search query for rose.
The user clicks on facets like the family rosaceae and flowerColor red, and counties of all those results are marked on the Google map.

![Facets: Family:rosaceae, FlowerColor:red](image)

**Figure 21.** Example of faceting with Google maps.

Next, to find the red-colored roses available in Utah County in the state of Utah, the user zooms in the map and clicks on the UT-Utah county marker, as shown in Figure 22. As indicated previously, the map does not just have the marker as UT-Utah County. Instead it maps all the locations that have the current listed plants in the search results. This is to enable the user to facet on the other locations that has the same plants, an important feature for the user.
Figure 22. Text of faceting example.
CHAPTER 7

SUMMARY

This project focuses on implementing a search engine called Plants Search Engine for the flora of the United States of America. Plants Search provides a useful and timely tool to the botanical community, because it allows users to search plants based on their characteristics. Search engines like Google and Yahoo, as well as other well-known engines, do not index the internal databases of websites like PLANTS database, which provides data on plant characteristics that can be used in a search.

Plants Search Engine acquires data from PLANTS and Flora of North America, analyzes it, and indexes it into a powerful search engine called Solr. A web interface is developed on top of Solr to perform search and display the results. The search engine categorizes the plants based on the hierarchical classifications to which a plant belongs, as well as on the characteristics of the plants, such as the color, shape, size, etc. Data from several sources are indexed into the Plants Search Engine making it easier to find the required plant species with just a few clicks and with a lot less time.

The location facet feature allows the user to search plants based on geographical locations, and Google Maps integration with Plants Search Engine makes this feature user friendly. Simply clicking the markers on the map resulting from a search fetches the plants belonging to that location. Range searches help the user identify plants specifically belonging to a particular range.

As of now, Plants Search Engine holds about 47,000 records of structured data from the PLANTS database and 65,000 records from the Flora of North America. Future
work could include indexing data from other countries. The current process of data acquisition could be improved by automatically fetching the structured data through newly developed Web services and delta indexing performed on a daily basis. Features like highlighting the search results and auto suggestions for search queries could also be included in future work. Also the Images, plant guides, other types of documents related to the data are not indexed in Plants Search Engine.
REFERENCES


