Database Support for Minority Language Policy Research

Rick Daniel

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

This project covers the development of a system intended to help minority languages researchers by providing a search website for reports produced by the European Charter for Regional or Minority Languages. The source reports are presented in PDF documents and mostly not searchable by document viewers. The built system reads the documents using OCR and converts the contents into a searchable texts, by means of storing them in a database and indexing them in a search server. The built system was delivered as a proof of concept for a document database system. It reduces manual work drastically and provides extensible nature for developers to build ideal document database system.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rick Daniel*)

Acknowledgements

I would like to thank my supervisor, Kyriakos Kalorkoti, for his guidance and supervision during the project development, also his teaching on the Computer Algebra course,
Rob Dunbar (Head of Celtic and Scottish Studies of the University of Edinburgh), for his inputs on the planning of the project,
Graham Dutton, for his helps on the DICE server configuration,
Dave Cochran, for his guidance on scientific writing,
Don Sannella, for his advice as my personal tutor,
M. Arapinis, D. Aspinall, I. Diakonikolas, B. Franke, K. Heafield, S. Maneth, R. Sarkar, and K. Vaniea, for their teachings on the courses I took in my one year of education here,
Papa, Mama, brothers, and sister, for their supports and prayers,
And of course, Lembaga Pengelola Dana Pendidikan Republik Indonesia, who made it possible for me to pursue my postgraduate study here.

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

This report covers the development process of a system built to solve a problem raised by the School of Literatures, Languages, and Cultures of University of Edinburgh. The researchers from the school's Department of Celtic and Scottish Studies needed a system to help them with minority languages research. The problem is covered in more details in Chapter 2. The system is designed to convert and store research documents in a searchable database. Naturally the system is also expected to have an interface for the researchers to perform searches on the documents, as well as administrator-related features.

The development process achieved a system that implements the primary function of the intended system, that is enabling searches on the converted documents. The system proved to reduce manual work drastically. Details of the implementation of the system are covered in Chapter 4. Some unexpected difficulties caused the development process to be delayed and some initially proposed features omitted. This leaves many available improvements of the system. Description of how to access the system is covered in Section 3.5.

1.1 Report Outline

The rest of the report covers the details of the project:

- Chapter 2 covers the basis organisation and data set intended for the system, the problems, the ideal solutions of the problems, and the achieved solutions.

- Chapter 3 covers the tools and framework used in the system.
- Chapter 4 covers the details of implementation of the system, how to use it, and difficulties found during the development process.
- Chapter 5 concludes the report and discusses possible future work.

Chapter 2

Background

This chapter covers the background of the project, particularly the Council of Europe's European Charter for Regional or Minority Languages, from which the problem arises, and the problem itself. Also it states the scope of the project, which includes the ideal solution of the problem and the deliverable solution given the time and resource constraints.

2.1 Council of Europe's European Charter for Regional or Minority Languages

The Council of Europe is an organization in Europe, which focuses on human rights, democracy, rule of law, cultural identity and diversity, and social challenges among others. It was founded in 1949 and headquartered in Strasbourg, France. The Statute of the Council of Europe states that "the aim of the Council of Europe is to achieve a greater unity between its members for the purpose of safeguarding and realising the ideals and principles which are their common heritage and facilitating their economic and social progress" [20].

One of the aims of the Council of Europe is "the protection and promotion of the wealth and diversity of Europe's cultural heritage" [17]. It is protecting and promoting the regional and minority languages of European countries through ratifying the European Charter for Regional or Minority Languages. It goes along with the Council of Europe's focus on human rights by protecting linguistic rights.

The European Charter for Regional or Minority Languages (ECRML) was opened in 1992 and has been since ratified by 25 states. The Charter covers 79 minority languages from the ratifying state parties [8]. It monitors and evaluates how the state parties apply the Charter and makes recommendations on their policies. For the monitoring and evaluation, there are three types of reports we are concerned about:

1. State Periodical Reports

   The state parties have to submit a periodical report about their policies and actions as their commitment to the Charter. The reports are submitted every three years. There are outlines for the initial and periodical reports laid out by the Council of Europe.

2. Committee of Experts' evaluation reports

   The Committee of Experts examines the state reports submitted by the state parties. It addresses problems and questions found on the reports and evaluates them directly by visiting the states. It then gives its own evaluation reports based on the examination and evaluations. It submits the reports to the Committee of Ministers.

3. Committee of Ministers' Recommendation

   The Committee of Ministers receives the evaluation reports and decides to publish them. It then makes recommendation reports for the states about necessary actions on their policies regarding the Charter.

2.2 Problems and Scope

Researchers in the Department of Celtic and Scottish Studies from the School of Literatures, Languages, and Cultures of University of Edinburgh use the reports of ECRML, for example to help them in drafting legislation. They face problems in finding needed information from numerous documents. In order to extract information about something, they manually open each document where the information may be found and search it. However, most of the documents cannot be searched in a document viewer because they are malformed. This makes searching certain information a tedious task to perform manually.

The aim of the project is to build a system that enables the users to perform searches on the documents of the Council of Europe's reports. Before it can be used as a searching tool, the system needs to process the documents into a searchable database. The intendend workflow of the system for enabling search on one document is by parsing the contents of the documents, storing the parsed contents into a database, and indexing the stored contents. These processes are carried out by the system after an administrator provides the document. After the document has been processed into the system, the users can use the web interface to carry out a search. Detailed workflow and the structure of the system are covered in Chapter 4.

An ideal solution to the problem is a system that performs well on every aspect of the workflow and reduces manual work drastically. A system with the following specifications would automate most of the work and solve the problem completely:

- It processes the input documents and extracts metadata information automatically. The administrators of the system should be able to provide only the original documents and leave the system to determine the type of the document, country of origin, and year and cycle of publication. No further involvement of the administrators should be needed after the original documents have been fed to the system.
- It parses every detail of the documents automatically, including accompanying figures, tables, and charts. In cases where the documents cannot be parsed normally, it needs to use optical character recognition (OCR) to parse them. It should also reports any problems found in the documents to the administrators.
- It extracts the contents and stores them accordingly, by separating each chapter of the documents into different database entries. It needs to detect where each chapter starts and ends, extract the contents of each chapter, and determine the minority languages mentioned in each chapter.
- It indexes each searchable component of the documents, such as section numbers, section titles, document type, country of origin, year and cycle of publication, and minority languages mentioned. It needs to make sure every search query can be performed in short time.
- It provides an intuitive search interface and displays the results accurately. The users should be able to access the original document and get into the specific page where the result is found.

In order to build the ideal system as mentioned above, we need significant resources and time. The scope of this project is to build a system as close as the ideal one within the time and resource constraints. With one developer and around eight weeks of development time, we deliver a system with following accomplishments:

- It receives the original documents as input. However, the administrators need to provide its metadata. This is due to irregularities among documents which disallow the system to determine exactly where to find the metadata. This does not require an extra step of work for the administrators as they are required at the same time the administrators upload the documents.

- It parses the documents well using OCR. The high amount of malformed documents and inaccurate normal parsing force the system to use OCR for all documents. This approach gives better parsing result but takes longer time and more computing resources. The text contents are read just fine, however, the accompanying figures, tables, and charts may not be read correctly. For the case of the Council of Europe's reports, it is acceptable as there are not many figures and charts involved.

- It can determine and separate sections of only one type of report, the Committee of Experts' report. This is due to these reports being similar in structure and given higher priority, as they are the ones that the minority languages researchers focus on. The State Periodical Report is more irregular because it is provided by every country separately. The Committee of Ministers' Recommendation report does not need to be separated into sections as there is only one section for each document. However, the system provides an interface for the administrators to separate the sections manually. It also enables the administrators to edit the automatically separated sections of the Committee of Experts' report.

- It indexes each searchable component of the documents well. However, a document is only indexed properly given accurate section separation and metadata.

- It provides two modes of search. The homepage shows one search field and enables the users to perform basic search using arbitrary strings queries. Basic search processes the search query and finds the results in the contents of the documents. The second mode of search is advanced search, where the users can narrow the search by filtering section, document type, country of origin, year and cycle of publication, or minority languages.

Thus we have delivered a proof of concept system which was the main intention of the project proposal. The technical executions and visual details of the delivered system are covered in Chapter 4.

Chapter 3

Tools

This chapter covers the tools used for the system. Each section covers relevant tools for certain function and compares them to each other. Comparisons of strengths, weaknesses, and familiarity to the developer are considered as decision factors. The stakeholders of the project also require all tools used to be free and open-source. The implication is that this project is also released as an open-source project (the repository can be accessed in [32]).

3.1 Document Parser

There are many tools that can be used to extract contents of PDF documents. Most of these tools are open-source and free to use. We consider tools that conform to our requirements and decide which one to use. In this section we cover pdftotext, PDFxStream, Yomu, PDF::Reader, and Docsplit.

pdftotext

One of the standard tools for extracting texts from PDF documents is pdftotext. This tool comes with many Linux distributions freely. It is often a good choice, being a default tool and has been used by many people. However, this tool is a standalone command line program. Integrating it with our system needs extra steps of work that may consume much of the development time. This tool also cannot parse texts from the malformed documents properly, which is a serious issue for this project (see Section 4.1).

PDFxStream

One of the first requirements that we seek when deciding which document parser to use is that it can handle all types of contents, which are among others texts, images, tables, and charts. Considering that capability, PDFxStream comes up as one of the better choices, with it being used by many companies and government institutions to process billions of documents every year [11]. PDFxStream claims to be able to extract data from PDF documents fast and accurately. Compared to pdftotext, it extracts text 13% faster [12].

However, this tool is not free. In order to get a PDFxStream complete server-side license, we need to buy it for US$5,000, with an additional US$1,000 per year for support and maintenance [13]. This does not comply with our free and open-source policy. This tool also runs on a Java or .NET framework, which is not the framework that we choose for this project (see Section 3.4).

Yomu and PDF::Reader

Yomu and PDF::Reader are open-source Ruby gems for extracting texts and metadata from PDF documents [35, 22]. Because we use Ruby on Rails for our web framework (see Section 3.4), both tools are easy to integrate. However, both tools also cannot parse malformed documents.

Docsplit

Docsplit is a tool for extracting texts, pages, images, and metadata from documents. It is available as a command line tool and Ruby gem. It uses GraphicsMagick, Poppler, Ghostscript, Tesseract, and pdftk as underlying libraries [5]. A Ruby gem [1] is available as a wrapper for it to be used with Ruby on Rails web framework (see Section 3.4).

One feature that makes Docsplit the best choice for the project is its optical character recognition (OCR) capability, which is made possible using Tesseract [34]. OCR is the only available automatic solution for the malformed documents problem. In the malformed documents, the texts cannot be converted properly because of missing encoding information. Using OCR, the documents are parsed not by reading its embedded text contents, but by treating the documents as images and converting the texts in the images into machine-readable texts. Tesseract is an open-source OCR software, and is one of the most accurate [34, 6].

Using OCR means using more computing resources and longer time. In this project, documents are inserted into the system only once at the beginning. So, the long parsing time is experienced only at the beginning, which is before the system goes live. This will not affect the intended users directly as after parsing, the texts are stored in a database and can be processed fast.

We decided to use Docsplit as the tool to parse the documents. The decision is based on the reasons that it is capable of parsing the documents properly, free and open-source, easy to integrate with the web framework, while having no significant weaknesses.

3.2 Database

We consider two classes of database management systems to store the documents, non-relational (NoSQL) and relational (RDBMS). The developer is familiar with both NoSQL and RDBMS, having experiences with MongoDB, Redis, BigTable, Memcached, Riak, PostgreSQL, MySQL, and SQLite. There are many options to choose from both classes, but we narrow it down to one representative of each class. We take MongoDB as representative of NoSQL and PostgreSQL as representative of RDBMS, being the most widely used and advanced one from each class. We compare the two of them and decide which one to use.

MongoDB

MongoDB is the most popular NoSQL database management system [3]. As NoSQL, it is highly scalable and flexible, as it does not require schema to define the data [15]. It supports high write rate and high volume of data [14]. However, there are cases of data inconsistency and loss in real world use [25].

In this project, the database acts only as storage. The reports do not get revised, so most of the database entries are only written once. New reports come in cycles of three years, so there is no need for high volume storage. With no requirement of high write rate and high volume of data, PostgreSQL is the more preferred choice, along with other factors explained below.

PostgreSQL

PostgreSQL is one of the most popular open-source RDBMS [3]. It is an advanced RDBMS and is used by around 30% of technology companies, based on 2012 research done by 451Research [26]. It uses schema to define the stored data.

In this project, the structure of data to be stored in the database is already pre-defined, therefore the schema design is useful as we want to make sure all data follow the same structure. The database design is covered in more details in Section 4.2. We also design the database in a relational model, where there are many `JOIN` queries involved among the tables. PostgreSQL, as a relational database, performs three times faster in this sense than MongoDB. PostgreSQL is also proven to outperform MongoDB in memory use, and is the better general purpose database overall [25].

For the purpose of this project, PostgreSQL is the better choice. We decided to use it as the database management system for the reasons stated above.

3.3 Search Server

As mentioned previously, the database in this project acts only as storage engine. In order to have the database entries searchable, we need to index them using a search server. As the data is indexed using a search server, we can customise how to index the data.

Elasticsearch is the most popular search server [3]. It is based on Apache Lucene. Combined with Ruby gem Searchkick, it supports and handles full-text search, result snippets, word-stemming, whitespace, and misspellings [7, 31]. This gives the users flexibility and ease when doing a search.

In this project, we want the users to be able to search the entire extracted texts from the reports. This is made possible by the full-text search capability of Elasticsearch. The custom index enables a faster advanced search, by indexing the needed fields to narrow down the search space. We also want to show the users where in the documents the search results are found. Using the result snippets, we can show parts of the documents where the results are found, including several sentences before and after the results. And finally, we want to give the users flexibility searching English words. For example, searching the word “consistency” should also give results on the words “consistencies” and “consistent”. Extra features like auto-complete, personalisation (for example, showing results most related to a user’s previous search queries), and did-you-mean suggestions also make Elasticsearch the obvious choice.

The Ruby gem availability makes it easy to integrate with the web framework, the popularity makes it easy to discuss problems found with other developers, and its familiarity to the developer makes it ready to use without spending time learning it. We decided to use Elasticsearch as the search server for this project based on the reasons stated.

3.4 Web Framework

We saw that we use various tools to process the documents into searchable entities. Those processes work in the background without the users knowing what happened. What the users see is a web interface capable of performing searches. So we need to be able to provide the users with a web interface and integrate it with the data we processed beforehand. We also have an administrators interface where the administrators can carry out their tasks, which include documents management and users management.

The web framework we choose to delegate such tasks to is Ruby on Rails. We use Ruby on Rails for the following reasons.

Familiarity

The developer of this project has been working with Ruby on Rails for two years. With tight time constraint, experience with the tools is one of the most prioritised factors, as we cannot spend significant time learning new ones.

Integration

The framework is supported by many external libraries called gems. The tools we mentioned before can be integrated easily into the system using various gems.

Ease of development and maintainability

This framework uses the convention-over-configuration paradigm. This means that the developer does not need to specify things already set by the framework. This reduces much work for the developer. It is easy to maintain because it has its own conventions, so when another developer is added into the project, as long as the new developer is familiar with Ruby on Rails, there is no problem with knowledge transfer.

Model-view-controller (MVC) architectural pattern

The MVC separates the system into three core components. The models manage the data and are connected directly to the database. The views display the data to the users. The controllers connect both of them by receiving user input, sending commands to change data to the models, and sending commands to change view to the views. This pattern is widely used and maintains logic separation.

Popularity

As mentioned before in this project's proposal, Ruby (the underlying programming language of Ruby on Rails) has around 116,804 gems [30] and the Ruby on Rails framework is used by many websites, including GitHub, Airbnb, Hulu, and SoundCloud [28]. Therefore, it is a very popular framework and there is a large community behind it where problems can be discussed.

3.5 Web Server and Deployment

As stated in the project proposal, we planned to deploy the system in a University of Edinburgh School of Informatics' server, integrated into the DICE system [16]. However, due to time and access limitations, we could not get the server ready for deployment. We decided to fall back to another virtual server provided by DigitalOcean [4] for a demo purpose to deploy the proof of concept system. The final version of the project will be deployed to the DICE server in the future, complying with the proposal.

The web server is running Ubuntu with Ruby on Rails, PostgreSQL, and Elasticsearch installed, along with other dependencies of the project. The deployment configurations are covered in Section 4.6.3. The deployed system can be accessed in [33]. An administrator access is provided with the email "`admin@seeker.com`" and the password "`adminpassword123`".

Chapter 4

The System

This chapter covers the details of how the developed system works (see Section 3.5 on how to access the developed system). Each section covers each core module of the system (see Figure 4.1). The core modules are the parsing module, the database module, the indexing module, and the web module. The web module is divided into a search module and an administrator module.

The main workflow of the system is divided into two parts. The first one is the process of inserting the documents into the system. The second one is the process of searching the inserted documents. A document is inserted by the administrators through the administrator module to the parsing module, then passed to the database module to be stored, and finally prepared for searching by the indexing module. A user may then perform a search through the web interface that communicates with the database and indexing modules. The overall diagram of the system can be seen in Figure 4.1 and detailed explanations of each component follow in the next sections.

4.1 Document Parsing

The main purpose of this project is turning the reports from the ECRML into a searchable database. In order to do that, the contents of the reports need to be extracted from the PDF documents provided by ECRML's website [18]. Using Docsplit [5], we use OCR to extract texts from the documents. The parsing module takes the URL and information about a document and outputs the parsed texts of the document (see Section 4.5.1 for an example of document insertion).

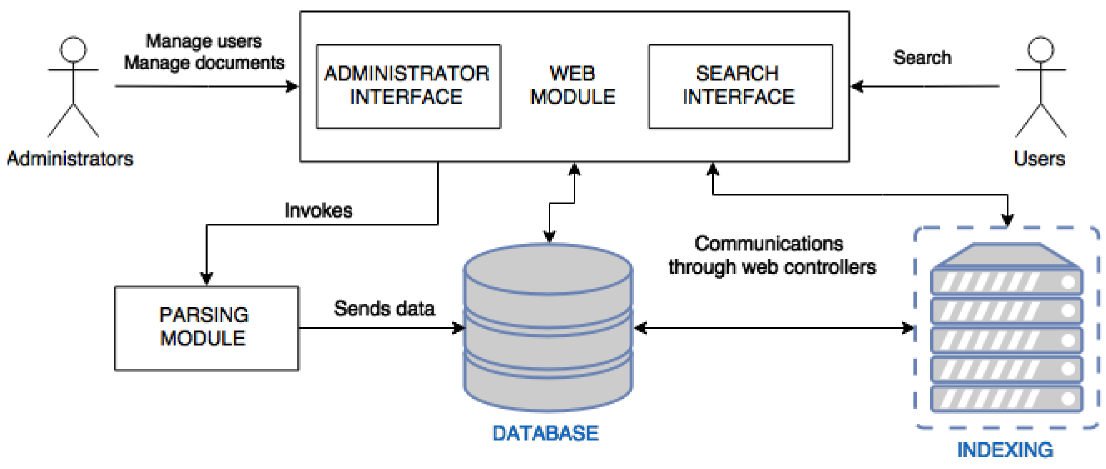


Figure 4.1: System Diagram.

4.1.1 Document Structure

This section covers the structure of the reports. Each type of report retrieved from ECRML's website follows a certain structure, as outlined by the ECRML. To enable an advanced search function, the sections of the documents need to be parsed and stored separately (see Section 4.2.2).

State Periodical Reports

There are two outlines provided by ECRML for state parties to follow for their periodical reports, the initial report outline [9] and subsequent 3-yearly report [10]. However, not every state party follows the outline exactly. For example, the outline states that a section is titled "Part I", but some state parties may write "Part 1", "Part One", "1. Part I", or "Part A" instead, or there are variations of "Introduction", "Introductory", with or without extra words added, and there are sections not written by the submitter in some documents. The resulting reports structures are then easy to read for a human but hard to distinguish for a computer. This makes separating the documents into sections a difficult task.

Committee of Experts' evaluation reports

The Committee of Experts' evaluation reports are submitted by the Committee of Experts itself, so there is less, if none at all, variation in document structure and naming. Therefore, section separation can be done automatically by the system. As this is the type of document that the researchers are most concerned about, the automatic separation helps to reduce manual work significantly.

Committee of Ministers' Recommendation

The Committee of Ministers' Recommendation reports are usually 1-2 pages long, containing recommendations made by the Committee of Ministers regarding the policy of the state party of the document. It does not need section separation as the reports contain only one section. The report is usually included as the second chapter of the corresponding Committee of Experts' evaluation report, as the ECRML publishes the Committee of Experts' evaluation reports after the Committee of Ministers gives their decision to publish them [19].

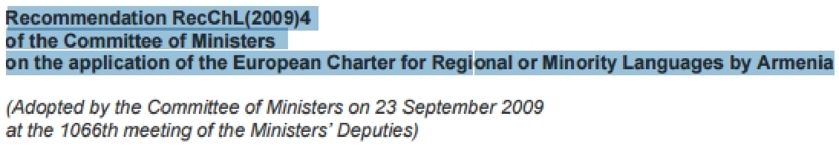
4.1.2 Malformed Documents

The main problem of parsing the documents is that most of the documents on the ECRML's website are broken. A document viewer can only display the document. However, when a reader wants to search the document using the search feature of the document viewer, it results in nothing. Furthermore, when texts of the documents are copied and pasted to another place, it results in gibberish texts containing unreadable symbols, as can be seen in Figure 4.2. The causes are that the documents are missing encoding to map the texts into readable texts, or that the documents use fonts that are not available on the reader's machine. To solve this problem, the system needs to convert the unreadable texts into readable ones. The solution is to use OCR.

4.1.3 Processing the Documents

As stated before, we use the OCR capability of Docsplit to extract the texts from the documents by treating the documents as images and recognising the texts contained in the images. This is done by Docsplit's use of ImageMagick and Tesseract. As Tesseract is one of the most accurate OCR tools available, the results are texts that are accurate compared to the original texts in the documents and can be processed by computers.

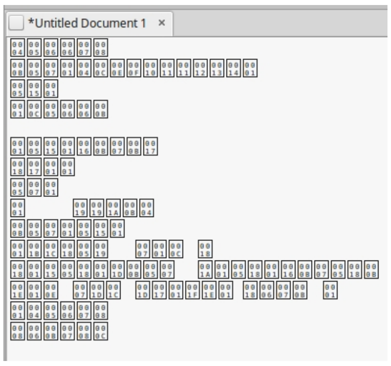
For a document to be inserted into the system, an administrator has to provide the URL to the document and its other information (document type, country of origin, and cycle and year of publication). The parsing module of the system then downloads the document into the the local server and starts parsing it using Docsplit. Due to OCR usage, the parsing process usually takes a long time and high processor usage



Recommendation RecChL(2009)4
of the Committee of Ministers
on the application of the European Charter for Regional or Minority Languages by Armenia

(Adopted by the Committee of Ministers on 23 September 2009
at the 1066th meeting of the Ministers' Deputies)

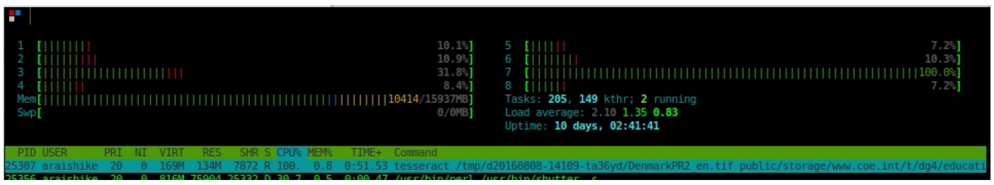
(a) The passage highlighted in a document viewer.



(b) The passage copied and pasted in a text editor.

Figure 4.2: A passage from Armenia's 2nd cycle Committee of Ministers' Recommendation.

depending on the length of the document. For example, a 29-pages report takes about 3 minutes and 20 seconds to parse, utilising single core of the processor at 100% usage (see Figure 4.3). After the parsing process is finished, the parsed texts are then stored to the database by the database module.



```
  1  [||||||||                          10.1%]    5  [||||||                                                     7.2%]
  2  [|||||||||                         10.9%]    6  [||||||||                                                  10.3%]
  3  [||||||||||||||||||||||||          31.8%]    7  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  4  [|||||||                            8.4%]    8  [||||||                                                     7.2%]
  Mem[|||||||||||||||||||||||||||||||||||||10414/15937MB]    Tasks: 205, 149 kthr; 2 running
  Swp[                                          0/0MB]    Load average: 2.10 1.35 0.83
                                                          Uptime: 10 days, 02:41:41

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
25307 araishike  20   0  169M  134M  7872 R 100.  0.8  0:51.53 tesseract /tmp/d20160808-14109-ta36yd/DenmarkPR2_en.tif public/storage/www.coe.int/t/dg4/educati
25356 araishike  20   0  816M 75904 25332 D 30.7  0.5  0:00.47 /usr/bin/perl /usr/bin/shutter -s
```

Figure 4.3: Processor usage of Tesseract (Docsplit's OCR library) when parsing a document. Core number 7 is utilised at 100% usage by the `tesseract` process (see the bottom part of the figure).

4.2 Document Database

As we need to provide the users a system that is fast and reliable, we have to make sure the time-consuming parsing process is not repeated unnecessarily. Therefore, after the texts are parsed, we store them into a database so they can be used and searched over and over again without the need to re-parse them. This section covers the database module, describing the database design and how the parsed texts are stored. The database module takes the parsed texts and stores them according to the database schema.

4.2.1 Database Design

We designed the database to be in relational form, where we can minimise data redundancy. For example, a document is separated into sections, so we create one database entry for the document and one database entry for each section, where each section refers to the document entry. This way we do not need to store the document information in the section entries redundantly, as only one document entry is needed.

Each entry is stored in a corresponding table. Section entries are stored in the SECTIONS table and document entries are stored in the DOCUMENTS table. Other than the two mentioned, we also have tables COUNTRIES where the state parties of ECRML are stored and LANGUAGES where the minority languages covered by the state parties are stored. The relational schema can be seen in Figure 4.4.

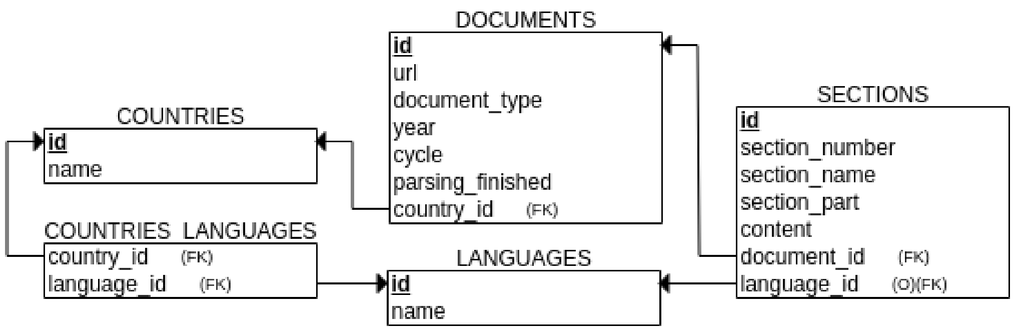


Figure 4.4: The relational schema of the database.

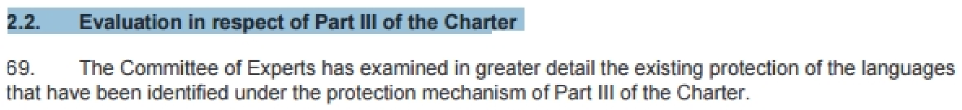
O = Optional; FK = Foreign key

A document entry contains the information that the administrator provides when the document was inserted (see Section 4.1.3). It also has a flag attribute `parsing_finished` to mark whether the time-consuming parsing process has finished or not. A section entry contains its number, name (title), and content. We also have a `section_part` field to store a long section (see Section 4.2.2). Because each report belongs to a state party, we store the `country_id` in the document entry to refer to the country. Each section belongs to a document, we store `document_id` to refer to the document. And as a section may cover a minority language, we optionally store `language_id` in its entry. We simplify the database design by assuming that one section covers at most one minority language, however, it can be changed in future development. Also as seen on [8], a language may belong to multiple countries and a country may have multiple languages, so we create many-to-many relationship between `COUNTRIES` table and `LANGUAGES` table using association table `COUNTRIES_LANGUAGES`.

4.2.2 Storing the Texts

After the parsing module has finished, the parsed texts are then passed to the database module to be stored. The raw texts then need to be separated into sections before they are stored in the database. As mentioned before, only the Committee of Experts' evaluation report documents can be separated automatically. Therefore, the other two types of documents are stored as a document with one section. However, the system has a feature where the administrator can separate the sections manually (see Section 4.5.2).

For the Committee of Experts' evaluation reports, each section starts with a section number followed by section title (see Figure 4.5). Therefore we decide to separate the section by finding the section title patterns and storing the texts between two found titles as one section. This is done after fixing the parsed texts where the section title may come right after the section number, which does not follow the pattern. After finishing separating, we also store the whole document content as one special section called "Full Content" section, to ensure that no data is lost. This is just like what we do to the other two types of document.


2.2. Evaluation in respect of Part III of the Charter

69. The Committee of Experts has examined in greater detail the existing protection of the languages that have been identified under the protection mechanism of Part III of the Charter.

Figure 4.5: Typical section number and title in Committee of Experts' evaluation reports.

Because there is a limitation on Elasticsearch where a string longer than 32,766 characters cannot be indexed, we automate the section storing process by dividing a long section into several shorter ones. This is where we use the `section_part` field on the database as mentioned in Section 4.2.1. This separation does not affect the users and administrators when they search or edit the sections of the documents, as it is made as if the long section were stored as one section entry.

4.3 Search Index

We use the database as storage so that we do not lose the documents data, however we cannot utilise the database to perform a search on the raw data in a reasonable amount of time. This is because PostgreSQL, the database of choice, does not perform as well as Elasticsearch in searching. As stated in Section 3.3, we use the search server Elasticsearch to index the data stored in the database. This index is stored in the local server to ensure fast communication between the web, the search server, and the database. To enable a fast search performance, we need to index appropriate fields. This section covers which part of the data we index.

The obvious data to index is the content of each section. We want the users to be able to search for any text in the documents. As mentioned in Section 4.2.2, there is a limit on the length of a string that Elasticsearch can index. This problem is overcome by the database module instead.

We also want the users to be able to narrow down the search space when using the advanced search feature. Therefore we need to be able to find sections based on the criteria wanted by the users quickly. So we index the fields where each criterion is stored. The fields for this requirement are the section table's `section_number` and `section_name`, the document table's `year` and `cycle`, the country table's `name`, and the language table's `name`. In addition, to prioritise search results on separated sections (see Section 4.5.2), we also index whether the section is "Full Content" or not. All those fields are related to any section entry, and the Ruby gem Searchkick that we use works automatically with Elasticsearch to index the sections based on the fields mentioned (see Section 4.6).

4.4 Web Search Interface

What the users see when using the system is only the search interface. They can perform searches on the inserted documents through the search interface. This is connected to the database and search server through the web framework's controllers (see Section 3.4). There are two main pages the users can see, the search form page and the search result page.

4.4.1 Search Form

The system is designed to be a search engine website for the ECRML reports, so the first page that the users see when accessing the website is a page with the search form. A user can type the search query into the form and the system performs the search for them. The search query is handled in a way that the sections containing all terms in the query are prioritised in the results. The system also prioritises the results from contents not in "Full Section" sections.

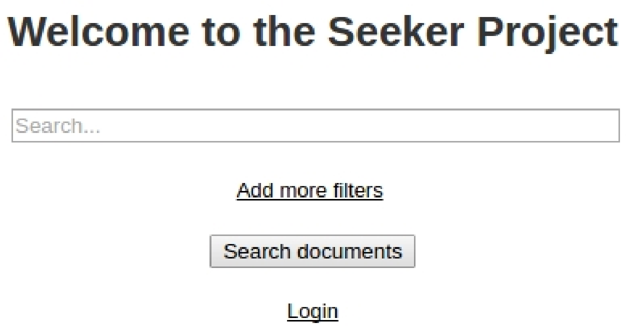


Figure 4.6: The front page of the website. "Seeker" refers to the name of the project.

A user can also perform an advanced search, where they can narrow the search results by adding filters. It is available on the front page by clicking the "Add more filters" link. They can then choose the type of the filter and the content. They can narrow the search results by section number, section title/name, country of origin, language covered, or document year and cycle. For example, they can perform search only on section 2.3 of documents from Austria (see Figure 4.7).

4.4.2 Search Results

After submitting a search query, a user is then presented with the search results for the query. The search results page is opened in a new browser tab in order to preserve the query submitted by the user in the search form page. The search results are presented

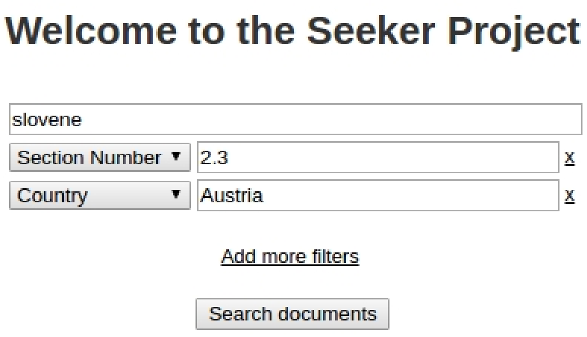
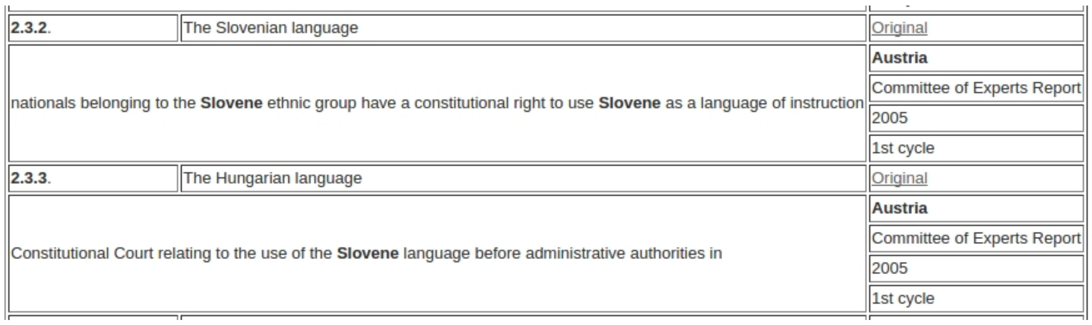


Figure 4.7: Searching “slovene” on section 2.3 of documents from Austria.

as a list of sections that match the query. For example, Figure 4.8 shows the results of the advanced search query from Section 4.4.1.



2.3.2.	The Slovenian language	Original
nationals belonging to the **Slovene** ethnic group have a constitutional right to use **Slovene** as a language of instruction		**Austria** Committee of Experts Report 2005 1st cycle
2.3.3.	The Hungarian language	Original
Constitutional Court relating to the use of the **Slovene** language before administrative authorities in		**Austria** Committee of Experts Report 2005 1st cycle

Figure 4.8: Search results of “slovene”.

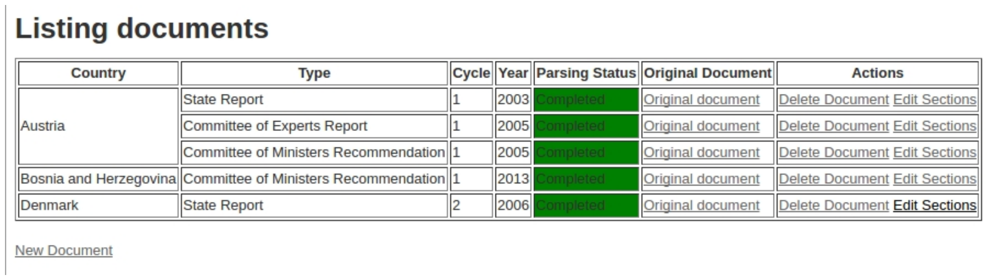
Each entry of the search results shows a section, displaying its section number and title, a link to the original document, information about the document, and a snippet of passage where the search keyword is found. It also highlights the submitted query.

4.5 Web Administrator Interface

The other users of the system are the administrators. They are tasked to manage the documents, which include inserting new documents, providing information of the documents, and maintaining separation of sections of the documents.

4.5.1 Inserting Documents

After logging in into the website, an administrator can access the document management page through a link on the front page. The document management page can be seen as in Figure 4.9.



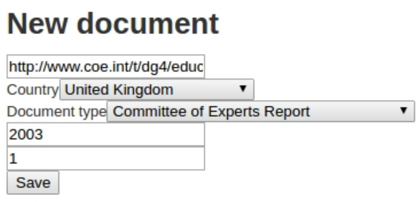
Listing documents

Country	Type	Cycle	Year	Parsing Status	Original Document	Actions
Austria	State Report	1	2003	Completed	Original document	Delete Document Edit Sections
	Committee of Experts Report	1	2005	Completed	Original document	Delete Document Edit Sections
	Committee of Ministers Recommendation	1	2005	Completed	Original document	Delete Document Edit Sections
Bosnia and Herzegovina	Committee of Ministers Recommendation	1	2013	Completed	Original document	Delete Document Edit Sections
Denmark	State Report	2	2006	Completed	Original document	Delete Document Edit Sections

New Document

Figure 4.9: Manage documents page.

An administrator can insert a new document into the system by clicking the "New Document" link on the document management page above, and the system then shows a form to be filled in by them. For example, an administrator wants to add a new Committee of Experts' evaluation report from United Kingdom, they fill in the form as in Figure 4.10. After submitting the information, the system then downloads the document and invokes the parsing module to start performing OCR on the document (see Section 4.1). The document management page shows the new document with its parsing status, which says whether the parsing process is still running or finished already (Figure 4.11). It is a static page where the administrator needs to refresh it manually to see whether the parsing status has been updated.



New document

http://www.coe.int/t/dg4/educ

Country: United Kingdom

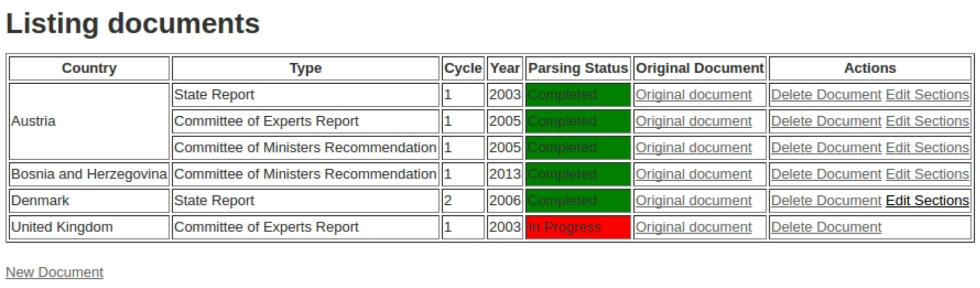
Document type: Committee of Experts Report

2003

1

Save

Figure 4.10: Inserting new document.



Listing documents

Country	Type	Cycle	Year	Parsing Status	Original Document	Actions
Austria	State Report	1	2003	Completed	Original document	Delete Document Edit Sections
	Committee of Experts Report	1	2005	Completed	Original document	Delete Document Edit Sections
	Committee of Ministers Recommendation	1	2005	Completed	Original document	Delete Document Edit Sections
Bosnia and Herzegovina	Committee of Ministers Recommendation	1	2013	Completed	Original document	Delete Document Edit Sections
Denmark	State Report	2	2006	Completed	Original document	Delete Document Edit Sections
United Kingdom	Committee of Experts Report	1	2003	In Progress	Original document	Delete Document

New Document

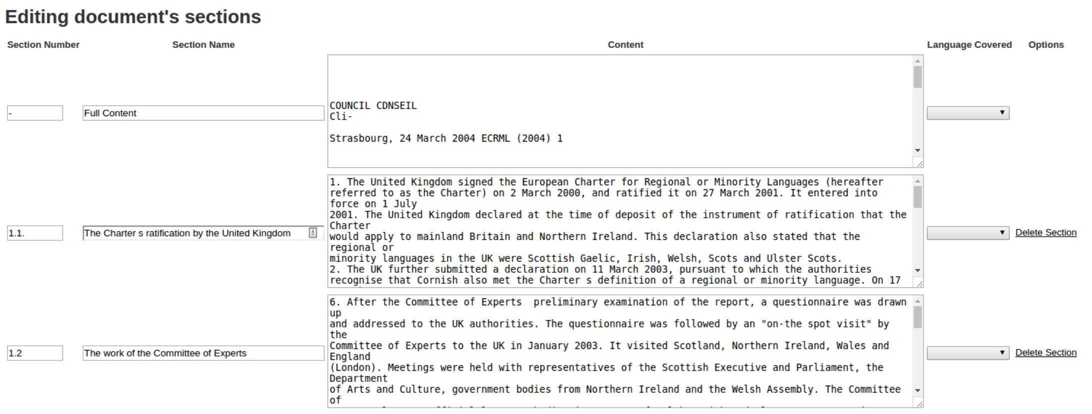
Figure 4.11: Manage documents page after inserting new document, showing "in progress" parsing status for the new document.

4.5.2 Manual Section Separation

In order to enable the users to narrow their searches by sections, the sections of the documents need to be stored and indexed separately. As discussed in Section 4.2.2, Committee of Experts' evaluation reports may be separated automatically by the system, but the other two types of reports are not. An administrator can separate the sections manually using this interface. They can also edit the existing section separation of the documents, including the Committee of Experts' evaluation reports, as the automatic separation by the system may not be perfect.

The documents are indexed by sections, which means that when a user searches for a query, the system returns one result per section. Therefore, it is important for the administrators to separate the sections accordingly to improve the search results. Furthermore, the system is designed to prioritise search results from sections not stored as "Full Content".

As can be seen in Figure 4.11, for each document that has been parsed successfully, there is an "Edit Sections" button. An administrator can manage how the sections of a document are separated to each other by clicking that button.



Editing document's sections

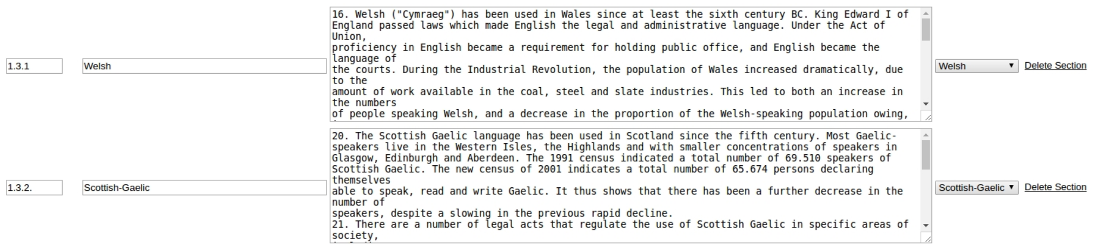
Section Number	Section Name	Content	Language Covered	Options
-	Full Content	COUNCIL CDNSEIL Cli- Strasbourg, 24 March 2004 ECRML (2004) 1		
1.1.	The Charter s ratification by the United Kingdom	1. The United Kingdom signed the European Charter for Regional or Minority Languages (hereafter referred to as the Charter) on 2 March 2000, and ratified it on 27 March 2001. It entered into force on 1 July 2001. The United Kingdom declared at the time of deposit of the instrument of ratification that the Charter would apply to mainland Britain and Northern Ireland. This declaration also stated that the regional or minority languages in the UK were Scottish Gaelic, Irish, Welsh, Scots and Ulster Scots. 2. The UK further submitted a declaration on 11 March 2003, pursuant to which the authorities recognise that Cornish also met the Charter s definition of a regional or minority language. On 17		Delete Section
1.2	The work of the Committee of Experts	6. After the Committee of Experts preliminary examination of the report, a questionnaire was drawn up and addressed to the UK authorities. The questionnaire was followed by an "on-the spot visit" by the Committee of Experts to the UK in January 2003. It visited Scotland, Northern Ireland, Wales and England (London). Meetings were held with representatives of the Scottish Executive and Parliament, the Department of Arts and Culture, government bodies from Northern Ireland and the Welsh Assembly. The Committee of		Delete Section

Figure 4.12: Editing the section separation of the previously inserted document.

Figure 4.12 shows the section separation page. As mentioned before in Section 4.2.2, all documents are also stored as a special section containing all texts contained. We call the special section "Full Content". In the section separation form, this section cannot be modified and acts as the source for the editing administrator to copy and move the passage to the proper section. The administrator can add a new section with a button at the bottom of the form and a new empty section will be added. They can also delete the sections (except the "Full Content" section) by clicking the "Delete

Section" button next to each section. There is currently no duplicate-checking done on the sections submitted by the administrators, as it is not computationally efficient to compare very long strings to each other (a 29-pages report may contain around 100,000 characters). However, future development may modify the section separation interface in a way that duplicate-checking can be done efficiently (see Section 5.2).

Another filter available for the users to narrow their search is the minority language covered. After a document is inserted into the system, there is no language assignment to the sections yet, as we have not discovered a way to automatically associate a chapter with its corresponding minority language. An administrator can then manually assign a minority language to any section that covers it. For example, in the previously uploaded United Kingdom's document, there are several sections each covering different minority languages in the United Kingdom. Figure 4.13 shows how the language can be assigned to the section.


1.3.1 Welsh

```
16. Welsh ("Cymraeg") has been used in Wales since at least the sixth century BC. King Edward I of
England passed laws which made English the legal and administrative language. Under the Act of
Union,
proficiency in English became a requirement for holding public office, and English became the
language of
the courts. During the Industrial Revolution, the population of Wales increased dramatically, due
to the
amount of work available in the coal, steel and slate industries. This led to both an increase in
the numbers
of people speaking Welsh, and a decrease in the proportion of the Welsh-speaking population owing,
```

Welsh ▾ Delete Section

1.3.2. Scottish-Gaelic

```
20. The Scottish Gaelic language has been used in Scotland since the fifth century. Most Gaelic-
speakers live in the Western Isles, the Highlands and with smaller concentrations of speakers in
Glasgow, Edinburgh and Aberdeen. The 1991 census indicated a total number of 69.510 speakers of
Scottish Gaelic. The new census of 2001 indicates a total number of 65.674 persons declaring
themselves
able to speak, read and write Gaelic. It thus shows that there has been a further decrease in the
number of
speakers, despite a slowing in the previous rapid decline.
21. There are a number of legal acts that regulate the use of Scottish Gaelic in specific areas of
society,
```

Scottish-Gaelic ▾ Delete Section

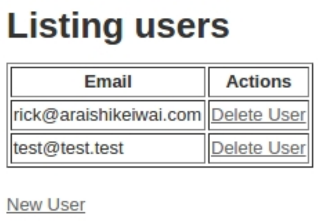
Figure 4.13: Assigning covered language to corresponding sections.

When the editing administrator finishes separating the sections, they click the "Save sections" button and the system will update the database and index to the new sections.

4.5.3 User Administration

In the current state of the system, there is no user registration system yet. However, an administrator can add new administrators for the system through the user administration interface. The administrators management page can be accessed through the link in the front page.

An administrator adds new administrator by clicking the "New User" button as can be seen in Figure 4.14. The system then shows a page with the new administrator form. The administrator then fills in the information regarding the new administrator, which contains their email address, password, and password confirmation.



Listing users

Email	Actions
rick@araishikeiwai.com	Delete User
test@test.test	Delete User

New User

Figure 4.14: Administrators management page.

After being created by hitting the "Save" button, the new administrator can log into the system using the submitted email and password and perform the administrators' tasks.

4.6 Integration

Figure 4.1 pictures the overall diagram where we can see how the system is centralised on the web module. Ruby on Rails and its gems make integrating all the modules into one seamless system easy. This section covers an overview on the relationships among modules, an example to show how the modules are connected to each other, and the deployment configurations. This serves as an illustration of the technical work undertaken during the development process. The example is taken from some parts of the code which give the general overview on the workflow of the system.

4.6.1 Modules Relationships

The web module, as the central module, is built with the MVC paradigm (see Section 3.4). Table 4.1 and Table 4.2 lists the models and controllers of the system and what they function as (each view corresponds to a controller's method, it renders the method's output).

Each model class represents a table in the database. The model classes integrate the database module to the web module. Some tables are not represented by any model class, for example the `COUNTRIES_LANGUAGES` table. This unrepresented table is integrated through the Ruby on Rails association method `has_and_belongs_to_many`. This method represents a many-to-many relationship between two model classes.

The controllers collectively serve as the backbone of the system, where they connect every module with each other. The documents controller acts as the main controller

Model name	Function
Country	It corresponds to the `COUNTRIES` table in the database.
Document	It corresponds to the `DOCUMENTS` table in the database.
Language	It corresponds to the `LANGUAGES` table in the database.
Section	It corresponds to the `SECTIONS` table in the database.
User	It corresponds to the `USERS` table in the database.

Table 4.1: Model classes of the system

Controller name	Function
Application controller	The parent controller, default from Ruby on Rails. As super class, codes that apply to all controllers go here.
Documents controller	The main controller of the system. It serves core functions: displaying the search interface, processing search queries and displaying the search results, displaying documents management pages, processing manual section separation inputs, receiving new documents, and invoking the parsing module. It is connected to the Document and Section models.
Users controller	It handles the users management interface. It is connected to the User model.

Table 4.2: Controller classes of the system

of the system, carrying out the core functions. It receives inputs from the users (including the administrators) and modifies the relevant database tables through their representative model classes. Because the indexing module is connected to the database module in a model class, all the modules are then integrated within the controllers. Section 4.6.2 further describes the interconnection among the modules.

4.6.2 Technical Details

The parsing process of a document is executed after an administrator submits it with the form in Section 4.5.1. Because the parsing process takes time, it is done asynchronously in the background. The background job is performed using the Ruby gem

SuckerPunch [27]. This job actually performs the tasks of the parsing module and partially the database module, where it downloads the document to the local server, performs OCR on it, separates the sections automatically given it is a Committee of Experts' evaluation report, and saves the sections to the database. The source code for this job can be found in the directory `app/jobs/document_parser_job.rb`.

When a section is saved to the database, it is split automatically into chunks of 30,000 characters (see Section 4.2.2). This is done via the `add_section` class method of the `Section` (`app/models/section.rb`) model class. In the same class, we define the indices used by Elasticsearch in the `search_data` instance method. The method is part of the Ruby gem Searchkick [31] which ties the Ruby on Rails model class with Elasticsearch. This is part of indexing module, and we can see how the indices mentioned in Section 4.3 are put into code in Listing 4.1.

Listing 4.1 Indexing module code in `Section` class

```
searchkick callbacks: :async, highlight: [:content, :section_number,
    :section_name, :country], word_start: [:section_number]

def search_data
  {
    content: content,
    section_number: section_number,
    section_name: section_name,
    country: document.country&.name,
    year: document.year,
    cycle: document.cycle,
    language: language&.name,
    full_content: full_content?
  }
end
```

The `searchkick` method connects the model to Elasticsearch, with the following options:

- `callbacks: :async`

  This means that we want Elasticsearch to run indexing process in the background.

- `highlight: [:content, :section_number, :section_name, :country]`

  This means that we want Elasticsearch to highlight parts of the contents of the `content`, `section_number`, `section_name`, and `country` fields that match the query when displaying the results to the users.

- `word_start: [:section_number]`

  This means that we want Elasticsearch to index the `section_number` field in a way that searching by section number gives all sections with section number started with the query. For example, a search on section number "1.3" gives sections 1.3, 1.3.1, 1.3.2, and so on.

We define the indices in the `search_data` method. The method returns a hash containing the indexed fields and their contents. The first three indices are clear, they contain the value of fields of the same name from each section. As mentioned in Section 4.3, the index fields `country`, `year`, and `cycle` contain the section's document's country name, year, and cycle respectively. The syntax `&.` is called "safe navigation operator" in Ruby, where it calls the method (`name` in the `country` and `language` cases) or returns `nil` if the caller is `nil`. And finally, we index the `full_content` field by calling the `full_content?` method of each section (see the source code for its implementation).

When a user performs a search, they send a parameter containing the query to the system. The parameter is received in `DocumentsController`'s `search` method. This method parses the parameter sent by the user and converts it into a search query for the `Section`'s model (which in turn sends the query to Elasticsearch using `searchkick`). For example, the basic search sends a parameter `q` containing searched text, the `search` method executes as shown in Listing 4.2 (the full source code can be found in the directory `app/controllers/documents_controller.rb`).

Listing 4.2 Part of `search` method, showing data flow from raw query into search results

```
search_query = params[:q]
fields = [:content]
highlight = { tag: '<strong>' }
```

```
boost_where = { full_content: false }
@search_results = Section.search(search_query, fields: fields,
    highlight: highlight, boost_where: boost_where)
@search_results = @search_results.paginate(page: params[:page],
    per_page: 10)
render :search_results
```

We can see how the query in the parameter `q` is passed as parameter for the `Section`'s `search` method, with the following options:

- `fields: [:content]`

  This means that we want to search the occurrence of the given query in the `content` field of the section.

- `highlight: { tag: '<strong>' }`

  This means that we want to add the HTML tag `<strong>` before the highlighted content matching the query and `</strong>` after it. The tag displays the matching content in bold text.

- `boost_where: { full_content: false }`

  This means that we want to prioritise search results from sections that are not part of "Full Content" section (see Section 4.3 and Section 4.5.2).

After that, the search results are split into pages of ten results each. This is so that the search results page is not crowded with all the results in one place. Finally the method renders the `search_results` page where the search results are displayed to the user.

4.6.3 Deployment

The system runs on the web server Passenger [21], which is integrated into the Apache HTTP server on the DigitalOcean virtual server. We use a Ruby gem called Capistrano [2] to automate the deployment process. Capistrano works through SSH to the server by pulling the code from the repository, preparing it for production environment, and restarting Passenger. The deployment configurations can be found in `config/deploy.rb` and `config/deploy/production.rb`.

Listing 4.3 Some deployment configurations found in `config/deploy.rb`

```
set :repo_url, 'git@github.com:araishikeiwai/seeker.git'
set :deploy_to, '/home/seeker'
set :linked_files, fetch(:linked_files,
    []).push('config/database.yml', 'config/secrets.yml')
set :linked_dirs, fetch(:linked_dirs, []).push('log',
    'public/storage')
```

Listing 4.3 lists some of the deployment configurations with the following behaviours.

- `set :repo_url`

  This line tells Capistrano to pull the code from the assigned repository [32].

- `set :deploy_to`

  This line tells Capistrano to pull the code into the assigned directory on the virtual server. This directory is where Passenger points to and loads the application.

- `set :linked_files` and `set :linked_dirs`

  This line tells Capistrano to create symbolic links on the code directory to the assigned files and directories. There are several reasons for this to be done. Firstly, we do not want to publish the two files `config/database.yml` and `config/secrets.yml` to the repository (therefore they do not get pulled by Capistrano when deployed) because they contain private information such as database password and secret key base (required by Ruby on Rails), so we prepare the two files beforehand on the server and link to them after deployment. Also, when deploying the system, Capistrano basically removes the old deployment and pulls the new code into new deployment directory. Therefore, all the files and directories are replaced and this could result in missing files. Because the system downloads the documents into the server in the `public/storage` directory, we want to keep this directory intact everytime we deploy the system or a patch. We also want to use the same log files throughout the system lifetime. Therefore, we prepare the directories `log` and `public/storage` in another location on the server beforehand and link to them.

4.7 Testing and Evaluation

4.7.1 System Evaluation

The system makes use of Rspec [29] to perform unit tests on the software. The tests cover the documents management interface, checking whether the users can access each page correctly. However, the unit test suite is not able to evaluate whether texts resulting from parsing are correct or not. This needs to be done manually by the administrators. This can be done by the administrators altogether with editing the section separation of the documents. Manual evaluation is also needed on the search results. However, because we index the sections properly, Elasticsearch will perform the search correctly.

To compare the performance impact of the system to manual work, we can take the following example. There are around three hundred reports available on the ECRML's website [18]. Most of the reports are very long, containing more than twenty pages. A manual overall search means that the person needs to read over six thousands pages to get information. Using the system, it may initially take a long time to converts all the reports, but searching for an information takes less than one second.

There has been a plan to present the system to Professor Rob Dunbar, the Head of Celtic and Scottish Studies of the University of Edinburgh, as the representative of minority languages researchers, and have him evaluate it. However, it was not possible in the end because of tight schedules of both the development process and the representative. Nevertheless, the supervisor of the project has carried out some functionality tests and was happy with the results.

4.7.2 Project Evaluation

The system has been built with focus on its primary objective. It did not achieve the ideal solution stated in Section 2.2, but it achieved a proof of concept system with many available improvements. We discuss the difficulties found during the development process in Section 4.8, and possible improvements in Section 5.2.

4.8 Setback and Recovery

We faced many setbacks during the development process. The fact that most documents cannot be parsed without using OCR was not expected at all. The study on how to parse the malformed documents while integrating the parsing process into the system so that it can be done automatically (see Section 3.1) took most of the development time. Significant amount of time was also spent in the process of coding the automatic section separation (see Section 4.2.2). The developer found out that despite the pattern discovered, there was noise coming from the “Contents” page of the documents, the page numbers, and bad line breaks. This resulted in the fact that many proposed features did not get to be implemented, such as the user ticketing system and user personalisation. Long correspondence time with the University of Edinburgh School of Informatics’ computing support also delayed the preparation of the virtual server for deployment, as the developer did not have full access to the server to configure it. We decided to move the deployment temporarily to DigitalOcean server (see Section 3.5).

Chapter 5

Conclusion and Further Work

This chapter concludes the report and suggests possible future work on the project.

5.1 Conclusion

The developer of the project successfully built a system that covers core functions of the ideal system. The developed system can create a searchable database from the documents of ECRML reports. A minority language researcher can use the system to search for information they need just within one step of typing search query into the search form. Compared to manual work of opening the ECRML's website, opening relevant documents, and reading through the unsearchable malformed documents, the system greatly reduces time and effort needed to get information. The created database can also be modified by the administrators to refine the search experience.

5.2 Future Work

The built system is still far under-developed compared to the ideal system. There are many things that can be improved to achieve a system that completely removes manual work and is easy and intuitive to use. Here are some possible extensions to the project.

- Automatic document feed

  In the current state of the system, an administrator has to insert the ECRML documents manually, including typing in the document information such as the country of origin, they type of the document, and the year and cycle of publication. An automatic document feeding system can be developed to update the system's database when a new report is published by ECRML. It also fetches the document information automatically from the document itself.

- User experience

  In the current state of the system, a user may find restriction when using the search form. For example, a user may want to perform a more advanced query that is not possible to be executed through the current search form, such as finding a section from Germany's periodical state reports from 2005 to 2010 and covers Danish or Romani languages. A new search form can be introduced where the users can translate what they want to the form easily. Possible additions include logic operator option, filter additions, pre-filled or autocomplete (either using existing information or prediction) fields, and manual query for more advanced users.

- User interface

  As can be seen in the screenshots provided in Chapter 4, the current system uses only basic HTML with minimum stylings. A user may find it irritating to their eyes and prefers more beautification of it. An advanced styling can be introduced to give the website a modern look.

- Administrator interface: section separation

  As stated in Section 4.5.2, there is currently no duplicate-checking on manual section separation. This problem can be solved by modifying the section separation interface into a more advanced one, where there is only one field containing the whole content of a document. The administrator separates the section by selecting the lines from the content for each section. This way, duplication or overlap can be detected visually by the administrator or automatically by the system.

- User registration and personalisation

  As mentioned in this project's proposal, we expected to have a personalisation system for the users. However, there was not enough time to implement it. The personalisation implementation implies a user registration system, where any user can register as a user of the website. Personalisation includes search options, favorite documents/countries, and customised indexing, where a user may choose which fields they want to boost when indexing.

- Ticketing system

  Also proposed in the project specification is the ticketing system, where a user can communicate with the administrators. The main purpose of this feature was proposed to be a suggestion box, where a user can give recommendation to the administrators about documents they want to be included in the system. There was also not enough time to implement this feature.

- Adaptation for other languages

  The documents intended for the built system are in English. However, there are documents in other languages among ECRML reports. The system can be extended to adapt to other languages. There are a few things to take a look into when adapting the system into another languages, such as the OCR to recognise words from other languages and the indexing system to recognise other languages (so that it gives similar experience to the English one when searching for "language" will also give "languages").

- Adaptation for other organisations

  With the modular design of the built system, there are possibilities that the system can be modified for other organisations' needs. Another organisation who wants to convert their documents to a searchable database can modify this project by adjusting necessary modules to match their documents. For example, an organisation with different structure of document but with similar behaviour of search can modify only the parsing and the database modules.

  Gaelic Language Plans

  A related similar project that needs the capability of the system is the Gaelic Language Plans from Scotland. This project was founded on The Gaelic Language Act and, among others, the Council of Europe's European Charter for Regional or Minority Languages [23]. However, significant extra effort will be needed as the plans do not follow a common format and can be in various document formats (PDF, Microsoft Word, or HTML) [24]. Furthermore, they are not held in any central repository, and this future system would act as one. This project is of great interest to minority language researchers in Scotland and would also be of interest to institutions charged with developing Gaelic plans. It was always intended as a further development of the system reported here.

5.3 Reflections

The project arose from a problem faced by minority languages researchers. At first, we had a lot of things planned for the proposed system, however, it turned out that there were many unexpected problems occurring during the development process. One of the major problems was that most of the ECRML reports had been submitted in malformed documents. We took several days to find out the solution of the problem. However, after finding out the problem and resolving it, the system became more impactful, as without the OCR solution to parse the malformed documents, it is nearly impossible for the researchers to get desired information from the malformed documents quickly. We sacrificed many proposed features to make sure the main objective was reached properly.

We decided on the tools and frameworks mostly based on familiarity and popularity. This had to be done as there were not enough time to learn new ones. However, we also learnt some new technologies that were required by the project. Among others, we learnt how to integrate an OCR-based document parser into a web system, we learnt to setup a search system built on a website, a database, and a search server, and we learnt how to deploy a website into a production server.

The built system is not perfect, but it can be improved in various ways. Nevertheless, the current system is useful for minority languages researchers and may get refined in the future.

Bibliography

[1] Jeremy Ashkenas, Samuel Clay, and Ted Han. docsplit. `https://rubygems.org/gems/docsplit`. Accessed 28 July 2016.

[2] Capistrano. `http://capistranorb.com/documentation/overview/what-is-capistrano/`.

[3] DB-Engines. Db-engines ranking. `http://db-engines.com/en/ranking`. Accesssed 7 April 2016.

[4] DigitalOcean. `https://www.digitalocean.com/`.

[5] Docsplit. `https://documentcloud.github.io/docsplit/`.

[6] Ubuntu Documentation. Ocr. `https://help.ubuntu.com/community/OCR`. Accesssed 31 July 2016.

[7] Elasticsearch. `https://www.elastic.co/products/elasticsearch`. Accessed 31 July 2016.

[8] European Charter for Regional or Minority Languages. Language covered by the european charter for regional or minority languages. `http://www.coe.int/t/dg4/education/minlang/AboutCharter/LanguagesCovered.pdf`. Accesssed 25 July 2016.

[9] European Charter for Regional or Minority Languages. Outline for periodical reports to be submitted by contracting parties. `http://www.coe.int/t/dg4/education/minlang/StatesParties/OutlineInitial_en.pdf`. Accesssed 31 July 2016.

[10] European Charter for Regional or Minority Languages. Revised outline for three-yearly periodical reports to be submitted by contracting par-

ties. http://www.coe.int/t/dg4/education/minlang/StatesParties/Outline3yearly_en.pdf. Accesssed 31 July 2016.

[11] Snowtide Informatics. Pdfxstream. https://www.snowtide.com/. Accesssed 28 July 2016.

[12] Snowtide Informatics. A performance comparison of pdf text extraction libraries. https://www.snowtide.com/performance. Accesssed 28 July 2016.

[13] Snowtide Informatics. Pricing and online purchasing. https://www.snowtide.com/purchasing. Accesssed 28 July 2016.

[14] Moshe Kaplan. When to use mongodb rather than mysql (or other rdbms): The billing example. https://dzone.com/articles/when-use-mongodb-rather-mysql. Accessed 31 July 2016.

[15] MongoDB. Nosql databases explained. https://www.mongodb.com/nosql-explained. Accessed 13 April 2016.

[16] The University of Edinburgh School of Informatics. What is dice? http://computing.help.inf.ed.ac.uk/what-is-dice.

[17] Council of Europe. European charter for regional or minority languages. http://www.coe.int/t/dg4/education/minlang/aboutcharter/default_en.asp. Accesssed 25 July 2016.

[18] Council of Europe. European charter for regional or minority languages. http://www.coe.int/t/dg4/education/minlang/Report/.

[19] Council of Europe. European charter for regional or minority languages. http://www.coe.int/t/dg4/education/minlang/aboutmonitoring/default_en.asp. Accesssed 25 July 2016.

[20] Council of Europe. Statute of the council of europe. http://www.coe.int/en/web/conventions/full-list/-/conventions/treaty/001. Accesssed 25 July 2016.

[21] Phusion Passenger. https://www.phusionpassenger.com/.

[22] Pdf-reader. https://github.com/yob/pdf-reader.

[23] Gaelic Language Plans. The gaelic language (scotland) act

2005. http://gaeliclanguageplansscotland.org.uk/en/why/the-gaelic-language-act.

[24] Gaelic Language Plans. Portfolio of gaelic plans. http://gaeliclanguageplansscotland.org.uk/en/tools-resources/development/plan-portfolio.

[25] Abraham Polishchuk. Sql vs nosql ko. postgres vs mongo. https://www.airpair.com/postgresql/posts/sql-vs-nosql-ko-postgres-vs-mongo. Accessed 31 July 2016.

[26] PostgreSQL. Frequently asked questions. http://www.postgresql.org/about/press/faq/. Accessed 13 April 2016.

[27] Sucker punch. https://github.com/brandonhilkert/sucker_punch.

[28] Rails. http://rubyonrails.org. Accessed 13 April 2016.

[29] Rspec. http://rspec.info/.

[30] RubyGems. Stats. https://rubygems.org/stats. Accessed 7 April 2016.

[31] Searchkick. https://github.com/ankane/searchkick.

[32] Seeker. https://github.com/araishikeiwai/seeker.

[33] Seeker. http://46.101.91.181/.

[34] Tesseract. https://github.com/tesseract-ocr/tesseract.

[35] Yomu. https://github.com/Erol/yomu.