Design Document

Group 44:
Chronicling America Project

Team Members

Ángel Chacón

Ricardo Pulido

Cole Silvernail

Sponsors

Amy Larner Giroux, PhD.

Computer Research Specialist

Center for Humanities and Digital Research

University of Central Florida

Marcy L. Galbreath, PhD.

Advisor & Lecturer, Department of Writing & Rhetoric

College of Arts & Humanities

University of Central Florida

Executive Summary

Goals & Objectives

The goal for this project is to deliver a functioning ALTO XML parser and a website with a back-end database. The parser needs to be able to separate newspaper pages into separate articles using the information in ALTO XML files. The database for this project must be able to hold all required information pertaining to user accounts, their queries, and the parsed newspaper data. Because of the quantity of newspapers and articles that will be in the database, the query tool needs to be robust, so that the server will not get overwhelmed by fetching too many articles. The query tool needs to be efficient and give the user many options to use in order to find their desired results without being flooded with irrelevant data. Proper indexing of relevant fields should help to decrease lookup time. Results should be chunked as to reduce load to both the server and the client, but quickly allow navigation to any chunk. The users need to be able to get specific about what they are querying. The website will allow users to save their queries, as well as download a heat map or csv file of returned query results.

Motivations

Ángel Chacón

As a person who enjoys programming, I have experience with web technologies and systems programming. Writing parsers and using XML are things I've recently wanted to try. My hopes for this project are to be able to write the tools needed to achieve an efficient product which can be used in real-world situations, and be able to show them off as part of the work I am capable of producing.

Ricardo Pulido

As a computer science student in UCF, I have been exposed to many different types of technology. Database System was a very interesting class that I enjoyed. I want to expand upon the knowledge I gain in that class and work on a larger scale database. This project will use a database that will be massive in size; thus, I hope I will gain some experience with dealing with fast data retrieval when the size of the data is large.

Cole Silvernail

As a person who enjoys programming, I have experience using web technologies and with systems programming. Writing parsers and using XML applications are also things that currently peak my interests and would enjoy trying. My hopes for this project are to be able to write the tools needed to achieve an efficient product which can be used in real-world situations, and be able to show them off as part of the work I am capable of producing.

Broader Impacts

The results of this project will show our sponsors new areas that they can explore to improve the project, such as supporting other languages, handling newspapers from outside the Library of Congress, and/or providing analytics of how the project is used. Being able to obtain historic newspaper information can improve the teaching tools of education faculties across the country.

Journalists and researchers can benefit from a one stop shop location to aid in finding old or obscure articles. A vast amount of new information from decades prior will be available to anyone who uses these tools, which can be analysed to gain a deeper understanding of how it influenced the people and communities who read it, as well as opens the door into big-data research such as language processing of historical and geographical writing styles.

Also, we hope this will turn into a powerful research tool for academics. We hope the new features along with the user interface will make the web app easily accessible and convenient for researchers to gain new insight into the times and issues they are studying.

Requirements

For this project to work the following objectives were layout. The objectives labeled “shall” have to be implemented; whereas, the objectives labeled “may” are optional.

“Shall” objectives:

1. Create a MySQL database.
2. Be able to store user data in database.
3. Be able to store newspaper data in database.
4. Create an ALTO XML parser.
5. ALTO XML parser must be able to determine what an article is on a newspaper. Store parsed article data in the database.
6. Create a front-end web page.
7. Users must be able to register with email confirmation.
8. Have secure logins for users.
9. Users can retrieve lost passwords.
10. Be able to query the database for all Boolean operators.
11. Be able to query the database for quoted phrases.
12. Be able to query the database for words within X words of each other.
13. Obtain metadata from articles through an API call to Chronicling America.
14. Be able to query the database for all metadata.
15. Be able to query the database with Wildcards with a minimum number of characters before asterisk.
16. Allow users to store/save their query results.
17. Allow users to reuse a previous query and refine the content.
18. Query mechanism must be robust, so it does not crash the server by having every one of the 12 million+ pages of articles returned.
19. Allow users to download CSV files containing article query results.
20. Allow users to download PNG heat map showing the geolocated articles.
21. Have warnings and terms of use for cookies based on the EU requirements.
22. System must run on Linux environment.
23. Minimum 200 of the OCR tar balls need to be parsed and included in the project to ensure the parser and query tool is robust. At least 50 must be over a gigabit in size.

“May” objectives:

1. Be able to automatically clean the OCR or have a crowd source implementation.
2. Create a mobile app for iOS
3. Create a mobile app for Android
4. Create a mobile app for Windows
5. Be able to multi-thread the query process to take advantage of multiple cores.
6. Link articles that span multiple pages

Overview

This Senior Design project consists of several parts. There are four main parts, and three optional stretch goals. These parts are: the website, the database, the query tool, and the parser. These parts will be connected in different ways, mostly to the database.

Parser

The first task is to create a parser that can read and separate sections of a newspaper into articles that will be indexed and stored in a database. Articles will consist mostly of a title and content. Exactly how the title and content are determined will be part of the challenge. These articles must also be searchable from a website.

The parser will be connected to the database. It will take in newspaper pages and insert the relevant data to store with the article into the database. This will be a completely isolated task between the parser and the database server, and so there should not be any user-facing issues.

Query tool

The second task is augmenting and building the search engine. One of the requirements is to add more sophisticated search features. This includes AND, OR, and NOT operators along with proximity matching, search using metadata, and explicit string search, along with the ability to combine these in different ways. The user must also have the ability, if they choose, to save these queries to the their account so they can be conveniently used or referenced later.

The query tool will be connected to the database by an inverted index. It will get a user query from the website, fill in any information that is necessary, search for the keywords in the inverted index, and find which articles contain those keywords. Next, the results will be ranked using the user query specifications and then returned to the user and displayed on the website.

Database

The third task is to create a database for storing the data for users and articles. As mentioned in the first task, the articles table will store the title and content, along with some metadata. The user table will store the user's username and/or email, salted password and its salt, preferences, and their search queries, as mentioned in the previous task.

Website

The fourth and last task is to design a website that allows users to register an account, save their search queries, and view a heatmap for various articles based on the location of the newspaper they come from.

The website will be connected to the database and the query tool. When logging in or registering, the website will have to look up accounts, create new accounts, or access/edit an account's information from the database.

Stretch goals

We were also given a few stretch goals.

One is to provide a way to correct OCR errors, either automatically using software techniques, or through feedback collected from users. We present a clever idea for achieving this while deterring bots from automatically creating accounts. Due to the already large amount of software that needs to be configured and run correctly, we did not have the time to pursue this. Instead, we provide a possible plan later on for how it could be implemented in the future by another group if they were to attempt it.

The second stretch goal is to include articles not written in English. Looking at the software we have written, we believe our code is general enough that it will work with many languages, especially those that depend on the latin alphabet.

The last goal is to create a mobile application that works for various platforms, such as Android, iOS, and Windows Phone. We have decided not to pursue this goal since it would be easier to create a mobile-friendly website that has all the same functionality.

Ideas

It is important to chose to use technologies that are well supported and can enable the completion of all the requirements. Older libraries will have wider support across systems, but sometimes are discontinued or have slowed in development, whereas newer libraries are packed with recent standard ideas, but sometimes lack quick adoption. A delicate balance is necessary for the successful design of this project. Of the technologies that exist, there are a few that we have focused on using for the project.

Ricardo Pulido

One way to discern the difference between articles and non-articles of newspapers is to have a word count. Upon inspection of the newspapers and XML files, I have noticed the following pattern: most articles contain many words, whereas advertisers like to keep their message short and simple. Furthermore, all articles have a title, whose font size is bigger than the rest of the article. An XML file contains different fields. The data we want is inside many TextBlock elements, which have many sub-elements called TextLine. Each TextLine contains one or many strings. Each string has a height that is associated with a font size, and a width that is associated with the string length. Adding the height of each string of a textline and dividing the sum by the number of strings in that TextLine element, an average can be obtained. This average will start at a high number that corresponds to the article title. When the average goes down a significant portion, then that will mark the end of the article title and the beginning of the article data. Then we can keep calculating new averages until we come across a number that is significantly bigger than the previous one. That will indicate the end of the article, and the beginning of the next article title. This process can be repeated over and over, until all articles in the XML file have been parsed.

Ángel Chacón

The ALTO XML files are very large, due to a combination of factors: the inherent verbosity of XML files, and the huge number of words that can fit on a single page because of the small font sizes used at the time. Because of this, careful consideration should be taken to ensure that system resources, such as CPU time and memory consumption, are used efficiently when parsing the files. There are two modern tools which can help us to achieve this.

Rust

Rust is a systems programming language designed with modern language design concepts in mind to prioritize performance and safety. It guarantees that common errors and mistakes, such as segfaults, memory leaks, and data races, are all caught at compile time. This keeps the whole program running smoothly, and greatly reduces unexpected errors. Rust also generates machine code using LLVM, a collection of modular and reusable compiler and toolchain technologies that can generate very fast

code on almost any architecture. This makes Rust a perfect candidate for writing an XML parser.

SAX Parser

There's an efficient XML parsing method called a SAX (Simple API for XML) parser. It is a event-based push parser. This means that this type of parser goes through each entity (start and end tags, text, comments, etc.) of the XML document in order, and generates an event with all its related information. This allows us to store only the necessary data.

The difference between this kind of parser and other XML parsers is that it doesn't use as much memory. This is because typical XML parsers do the parsing and validation all in one go, resulting in a lot of memory being used (often unnecessarily).

In contrast, a SAX parser only reads parts of a document at a time, even if it is not completely valid. As a result, it uses much less memory. According to Wikipedia, "the minimum memory required for a SAX parser is proportional to the maximum depth of the XML [element tree] and the maximum data involved in a single XML event." This helps hinder the issue of memory consumption.

OCR Correction

For the stretch goals, there's the issue of how to correct errors made by the OCR software when reading a word or character. We've thought of several ways to solve this.

Automated checking of words against a dictionary file is an easy and simple solution, although not great. Some issues that could occur are encountering unusual names or words, and finding black boxes in text that can't be fixed because a page has been ripped out or contains an imperfection detected by the OCR. Unfortunately, there's no known way to deal with the latter, but there is a way to remedy the former.

First, after someone has created an account, we could have a section on the login landing page for helping decipher some scanned words that the OCR has low confidence in. We could also use these words as a form of CAPTCHA to guard against bots that use up system resources by automatically creating accounts. By doing this, we solve two problems at once: we deter bots, ensuring the account is made by a human, and we decipher words that the OCR couldn't.

As clever as this solution may be, it has the unintended consequence of making it more difficult for legally blind people to make an account. A possible solution that I've found for this is to give an option for a text-based question or statement that informs the person of what word they must enter. This requires the question or statement to have high variety in its phrasing and formatting, to make sure bots do not catch on to any patterns that would make it predictable.

Cole Silvernail

On our project's website people will be making user accounts for various reasons. When a user creates a user account, they will be trusting us with some of their personal information. It is imperative that we handle and store their information in a safe and secure manner. To partly accomplish this, we could use SSL. SSL stands for Secure Sockets Layer. It secures an internet connection in order for information to be sent across the connection. It does this by encrypting the plain text being sent between the user and web server. This results in having the "https" in your website's url, along with the green lock icon, telling users that the website is secure.

To get HTTPS on our website, we would need to get a certificate from a Certificate Authority and prove that we control the domain. From there, it's as easy as installing the SSL certificate on our web server. The presence of the SSL in the server triggers the protocol.

My teammates have already stated a couple ideas for finding article titles. One other idea uses the metadata in the ALTO XML files. Specifically the metadata of TextLine elements. It seems in some cases that when an article ends the following TextLine element has an identifying line number that is vastly different from the previous one. This sometimes points to the beginning of a new article. Therefore the first line of each new article, will usually have a line number that is non-sequential to the previous TextLine's line number. Article's first lines are usually the article's title. This solution won't work everytime by itself, but if it is paired with other checks, it can help improve the accuracy of our parsing algorithm that will be identifying article titles.

One of the stretch goals is to make the web app mobile friendly. In some cases this may involve making mobile apps for iOS, Android, and Windows phones and devices. For this project it may be enough to make sure that we build the website in a way that will make it easily viewable from a phone or tablet. We can accomplish this by using responsive web design. This means that the website will respond to the user's viewing environment, taking into account the screen size being used to view the site, the orientation of that screen, and the platform the website is being from. This requires flexible web layouts to be used. This gets rid of the necessity to build a seperate website for mobile viewers or a mobile application. We will not be using any of the phones native processing nor will we need any native information from the phone for the website, which is why a mobile friendly website makes sense.

Although the mobile friendly website may be a satisfactory solution. A native application may hold some advantages. The website users may require to use the app in a personalized setting. The charts and queries may also be easier to adjust on a native app, although those can all be done through a mobile friendly website as well.

Group Ideas

Back End

Laravel

Laravel is an open sourced cross platform web application framework that aids in the creation of elegant websites through expressive and creative code. Laravel attempts to handle the workload of common tasks to ease development of projects. Laravel comes with an extensive documentation set and video series that allows fast learning of the framework. Powered by PHP, Laravel is equipped to be driven by a multitude of databases, provide simple and fast routing, handle real time event broadcasting, and more.

We can utilize Laravel to quickly spin up a reliable and powerful site, giving us more time to focus on the project requirements. Laravel provides an easy syntax to design database queries, especially chaining queries together for increased control. Server side form validation is possible through Laravel's easy validation rule sets and is customizable to our needs.

ExpressJS

Express is a fast and minimalist cross platform web framework built on Node. Handy for single page application, Express can hook into a multitude of database and ORM drivers, and deliver robust routing for the application.

Front End

ReactJS

React is an increasingly popular JavaScript library for building wonderfully complex user interfaces. Originally developed by Facebook and open-sourced to the community, React utilizes simply designed views with efficient updates encapsulated into components that manage their own state. React is considered an extension with little requirements, giving us the freedom to develop code as we please.

Laravel Blade

If we select Laravel as our backend driver, then we will have access to Laravel's HTML Blade templating. Blade is a simple tool that builds dynamic server-side rendered views. Form logic becomes a breeze using the smart directives that Blade offers, connecting to the power of Laravel's validation rule sets. Blade supports a template driven design, pulling common logic into higher order views.

Other Technologies

Webpack

Webpack is bundler for JavaScript files and resources to be used in a browser. We can utilize Webpack to compile the front end of the project into efficiently sized runtime files

to aid in a seamless experience for the user interacting with the website, as well as apply plugins to transform the files such as: minification, validate code standards, and lazy load libraries.

Yarn

Built on top of the Node Package Manager (NPM), Yarn is a faster implementation for loading libraries into a project as well as a method to run debug the front end without the back end.

Project Milestones

No	Task	Start	End
Senior Design I			
1	Ideas	9/19/2018	9/26/2018
2	Role Assignment	9/24/2018	9/26/2018
3	Initial Document: Divide and Conquer	9/21/2018	9/30/2018
4	First TA Check-in	10/01/2018	10/01/2018
5	Technologies Approval	10/02/2018	10/02/2018
6	Project Status Review	10/15/2018	10/22/2018
7	Second TA Check-in	11/05/2018	11/05/2018
8	20 pages per member completed	11/05/2018	11/10/2018
9	25 pages per member completed	11/15/2018	11/20/2018
10	30 pages per member completed	11/20/2018	11/25/2018
11	Final Adjustments	11/25/2018	11/30/2018
12	Final Design Document	12/03/2018	12/03/2018
Senior Design II			
13	Complete the Database	12/03/2018	01/07/2019
14	Continue to work on Individual Tasks	12/03/2018	01/07/2019
15	Status Team Meeting	01/07/2019	01/07/2019
16	Complete the Parser	01/07/2019	04/07/2019
17	Complete most of the Front End	01/07/2019	03/07/2019
18	Finish the project	03/07/2019	04/07/2019

Research and Investigations

What is the Chronicling America Project?

Chronicling America is a project aimed at collecting and digitizing old newspapers that can be indexed and searched online. It provides access to data regarding historic newspapers and some of their digitized pages.

It is produced by the National Digital Newspaper Program (NDNP), which is a partnership between the National Endowment for the Humanities (NEH) and the Library of Congress, and is an effort to create a queryable database of US newspapers. The Library of Congress is dedicated towards developing and forever maintaining this valuable resource. [13]

Chronicling America API

The Library of Congress hosts an API for the Chronicling America project that contains many useful features. This includes search features, auto-suggestions, permanent links to different resources, JSON representations of data with links to different entities, open Linked Data standards, bulk data, and methods for external access by JavaScript applications. Some of these features will be useful in the development of our project, so we will describe the relevant ones here. [12]

Bulk Data

The API provides a way to bulk download the newspapers scans and the OCR data. Usually this is done if someone using such data does not wish to rely on the uptime of the Library of Congress. There are two types of downloads.

One of them is the batch files that the Chronicling America project receives from the National Digital Newspaper Program (NDNP). This contains the newspapers themselves, their LCCNs, date received, number of issues, number of pages, the issue date for each of the issues, and a validated batch file.

The other is the OCR data generated from the newspaper. As the website mentions, this can be done when finding ways to index the newspaper data, or if high reliance on data availability for full text search is necessary.

The OCR data offered by this part of the API will be used very often in our project for gathering data for testing purposes, and for fulfilling our requirement of converting the OCR data (the ALTO files) into articles that can be indexed and stored in a database.

Search

The searching functionality is exposed to third-parties via an open standard called the OpenSearch protocol. Part of it defines how search features can be discovered using a format called the OpenSearch Description document. A description document is

available for searching newspaper titles or for searching their pages. They are discoverable through a <link> element in the website's HTML template.

The title search description document has three features:

- terms – terms or keywords for the newspaper title to include
- format – (optional) what format to return; either "html" (default), "json", or "atom"
- page – (optional) which page of results to return; default is the first page

The title search description document has three features:

- andtext – terms or keywords for the newspaper content to contain
- format – (optional) what format to return; either "html" (default), "json", or "atom"
- page – (optional) which page of results to return; default is the first page

Part of our requirements for this project consist of augmenting the searching capabilities mentioned here to include more complex querying features, such as binary operators (eg. AND, OR) and explicit string search, which can be saved, modified, and reused by the users of the website. Therefore, an understanding of how the search feature of the project works is necessary.

AutoSuggest

This is also part of the OpenSearch standard. The AutoSuggest API can return search suggestions for a given search string. With this, users can receive search suggestions relevant to the search engine being used, even while they are still typing out their query.

This returns a JSON array with:

1. The original search string given
2. A list of suggestions
3. A list of Library of Congress Control Numbers (LCCN) relevant to the respective suggestion in the previous list
4. A list of links to those LCCN numbers

While our new search capabilities will affect what kind of suggestions show up, our changes will likely not involve modifying this functionality in any way.

Links

The links to various resources in the Chronicling America project are persistent and follow a specific namespaced format. They are actively supported with redirections if the format were to ever change.

The links follow a specific format. They start with a "lccn" segment, and segments are separated by forward slashes ("/"). The next segments are, in order:

1. The LCCN
2. A date in ISO 8601 format (optional)
3. "ed-" followed by an edition number (optional)
4. "seq-" followed by a sequence number (optional)

Rather than having a date, one can also put “issues” instead and see a calendar view of the different issues in a given LCCN. Adding the string “first_pages” to this allows one to browse the first pages of available issues.

These links have already become crucial for us to find the original newspaper material from which the OCR data is based on. They are necessary in order to compare the format of the newspaper to that of the ALTO files and gain an understanding of how the machine will see the data.

JSON

There are links to JSON representations of the data provided by the website. Links like those mentioned above can be appended with “.json” to receive the JSON version of the data rather than the HTML version.

The JSON data contains links to other relevant entities that can be fetched for more information. This makes information traversal and searching much easier to do automatically.

This JSON data will be very useful when we start creating tools for testing and verifying our stuff works, and when we need data to test our parser on, so it is very useful.

Linked Data

Linked data refers to the structuring of information using standards with well-defined terms in order to describe both an entity that can hold certain data, and its relationships with other entities, which could also include data about that relationship.

This is part of the idea of the Semantic Web, where information is semantically encoded and defined in ways that can be read by computers. Multiple standards – and therefore multiple terms – are often used in order to support a variety of ways for software to describe certain things. Depending on the use case, a different standard may be more or less appropriate to use.

Such terms are used in an XML file format called RDF to describe the data. Any third-party that understands the terminology being used has a more thorough understanding of the semantic information being presented and how to use it than one who only has, say, a general understanding of XML.

The Chronicling America project uses different terms in their data from various organizations, particularly from:

- DBpedia
- OAI-ORE
- OWL
- RDA
- WorldCat
- Dublin Core and DCMI Terms
- FRBR concepts in RDF
- LCCN Permalink
- GeoNames
- lingvoj.org

As exciting as sharing semantic information using open standards may be, unfortunately this part of the API will likely not be used in our project. This is because our requirements do not specify anything about using any standards to encode the articles we will be parsing and storing into the database. Perhaps that is something that can be considered later on.

CORS and JSONP

Cross-Origin Resource Sharing (known as CORS) is a mechanism that allows a website script to make requests to resources that are located on another website or “origin”. As a safety measure, this can only be enabled by the server that would be receiving those requests.

JSON with Padding (known as JSONP) seems to be a similar mechanism, but is reserved for legacy browsers that do not support CORS.

Inside the JSON files

The JSON data offered by the Chronicling America API does not seem to be documented anywhere. This section will attempt to document the information returned by each API endpoint. [12]

Some shorthand types:

- URL: String with a URL as its value
- Date: String with ISO 8601 date format
- Year: String with “YYYY” year format, where the rightmost 1-2 digits can be ‘?’

All other types are the usual JSON types. If a type for a field is “Object”, then the fields for said object (along with their appropriate types) will be documented below in a sublist. If a type for a field is “Array”, it is followed with “of X”, where X is the type of the values in the array.

/lccn/snNNNNNNNN/YYYY-MM-DD/ed-E/seq-S.json

jp2: *URL*
Link to a picture of the newspaper page scan in JPEG-2000 format.

sequence: *Number*
The sequence number that this page corresponds to for this edition.

text: *URL*
Link to a text file of the content scanned by the OCR.

title: *Object*

> **url**: *URL*
> Link to the JSON file describing the newspaper publication.
>
> **name**: *String*
> Title of the newspaper publication.

pdf: *URL*
Link to a PDF file of the scanned page.

ocr: *URL*
Link to the ALTO file generated by the OCR.

issue: *Object*

> **url**: *URL*
> Link to the JSON file describing the edition/issue this page is a sequence of.
>
> **date_issued**: *Date*
> Date of when the edition/issue of this page was published.

/lccn/snNNNNNNNN.json

place_of_publication: *String*
String of the location of this newspaper and publisher, in “City, State” format, where State is an abbreviation of the state name.

lccn: *String*
The LCCN corresponding to this newspaper and publisher.

start_year: *Year*
Year in which this newspaper started.

place: *Array of String*
Array of places associated with this newspaper. It is unknown how these values are determined.

name: *String*
Name of the newspaper.

publisher: *String*
Name of the publishing company.

url: *URL*
Link to this JSON document.

end_year: *Year*
Approximate year to when this newspaper stopped publishing.

issues: *Array of Object*
Issues/editions aggregated in this LCCN.

> **url**: *URL*
> Link to the JSON document describing this particular edition.
>
> **date_issued**: *Date*
> Date that this editions was issued.

subject: *Array of String*
List of (supposed) subjects. How these values are obtained is unknown.

/lccn/snNNNNNNNN/YYYY-MM-DD/ed-E.json

title: *Object*
    url: *URL*
    Link to a JSON document describing the LCCN this edition belongs to.

    name: *String*
    Title of this newspaper edition.

url: *URL*
Link to this JSON document.

date_issued: *Date*
Date of when this edition was issued.

number: *String*
It is unknown what this field is for.

batch: *Object*
    url: *URL*
    Link to the JSON document describing the batch that this edition comes from.

    name: *String*

volume: *String*
Volume this edition corresponds to. How it gets this values is unknown.

edition: *Number*
The number of this edition.

pages: *Array of Object*
A list of the pages that this edition consists of.

    url: *URL*
    Link to the JSON document describing a page in this edition.

    sequence: *Number*
    The sequence number that this page corresponds to in the edition.

Structure of ALTO

An ALTO file consist of three major sections as children of the root element.

The first section is the description section. This section contains metadata about the ALTO file itself and the processing information of how the file was created. Below is an example of the Description element that we will be working with.

```
<Description>
    <MeasurementUnit>inch1200</MeasurementUnit>
    <sourceImageInformation>
        <fileName>/mnt/ra.iarchives.com/data01/jobq/root/projects/production/
            newspaper/Connecticut/NDNP_2014/batch_kent/BTEF_19200101-19200214/ocr/0039.tif</fileName>
    </sourceImageInformation>
    <OCRProcessing  ID="OCR.0">
        <ocrProcessingStep>
            <processingStepSettings>abbyy9.version:9.0.0.7394-3Abbyy9.cache-check.image-CRC:
                be7943ec3f086e483def717706f37d4dLang:engabbyy9.option.ocr-auto-pictures:trueConf.
                SPead:1Dictionary Words:3715Node Count:3993Option Count:3993abbyy9.
                option.analyze-manual-zones:falsePredicted Word Accuracy:100%Abbyy9.
                cache-check.base-page-CRC:e28d6eb57f809eaed91a3daa4ddb59eesource-image:
                /mnt/ra.iarchives.com/data01/jobq/root/projects/production/newspaper/
                Connecticut/NDNP_2014/batch_kent/BTEF_19200101-19200214/ocr/0039.
                tifAbbyy9.cache-check.wrapper-version:1.3Inverted:falseEngine:
                Abbyy9Conf:1abbyy9.option.analyze-zones:trueCharacter Count:19571abbyy9.
                option.hyphenation:trueSuspicious Character Count:771Word Count:3993
                width:5171height:6624xdpi:300ydpi:300</processingStepSettings>
            <processingSoftware>
                <softwareCreator>iArchives</softwareCreator>
                <softwareName>iArchives OCR Framework</softwareName>
                <softwareVersion>Multiple</softwareVersion>
            </processingSoftware>
        </ocrProcessingStep>
    </OCRProcessing>
</Description>
```

Figure: Alto Description

In this example the Description element contains the following children elements, MeasurementUnit, sourceImageInformation, and OCRProcessing. All measurement values inside the ALTO file (except font size) are related to the MeasurementUnit. The child element fileName, of the parent element sourceImageInformation, contains some metadata of the newspaper. In this case, the State that the newspaper originated from is easily obtained. Amongst other information, the width and height of the image file that was processed is available in the processingStepSetting element.

The second section is the style section. This section contains the text and paragraph styles with their own individual descriptions. TextStyle contains the font descriptions. Below is an example ALTO style section that we will be working with.

```
<Styles>
    <TextStyle  ID="TS_10.0_I" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="italics"/>
    <TextStyle  ID="TS_10.0_M" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="smallcaps"/>
    <TextStyle  ID="TS_10.0" FONTSIZE="10.0"/>
    <TextStyle  ID="TS_10.0_IU" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="italics underline"/>
    <TextStyle  ID="TS_10.0_U" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="underline"/>
    <TextStyle  ID="TS_10.0_S" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="subscript"/>
    <TextStyle  ID="TS_10.0_IP" FONTTYPE="serif" FONTWIDTH="proportional" FONTSIZE="10.0" FONTSTYLE="italics superscript"/>
</Styles>
```

Figure: Alto Style

The font family, size, and style of the entire text is available in this section. For our ALTO files the ParagraphStyle element, which contains information about the alignment of paragraphs, will be excluded.

The final section is the Layout section. This section contains Page elements, which consists of margins and printspace. Example below:

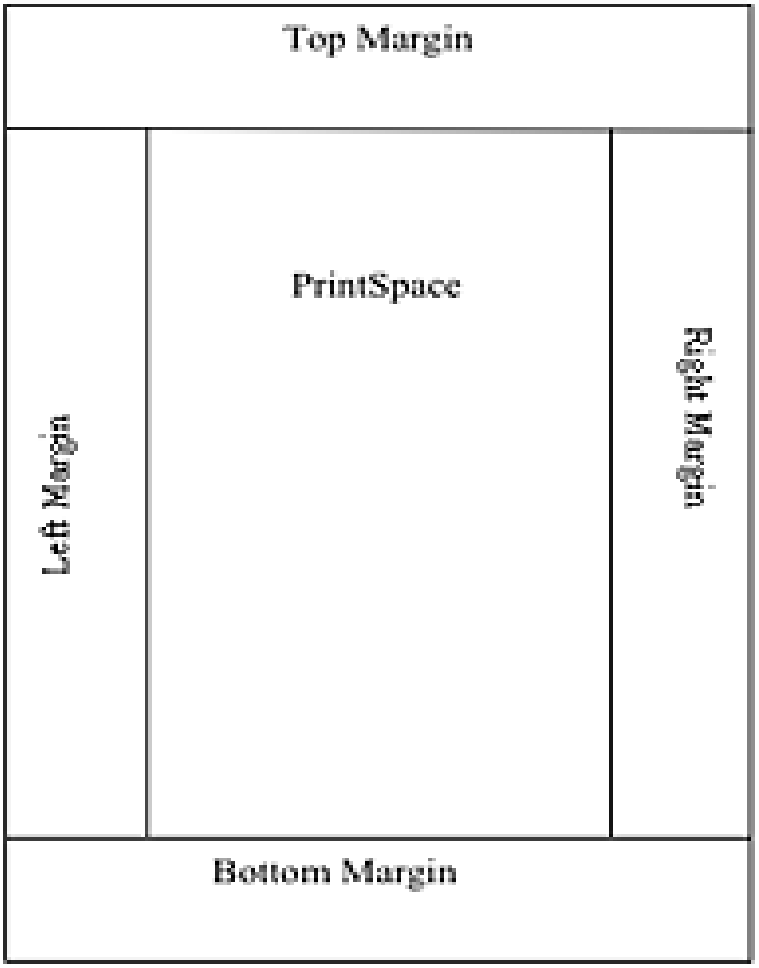


Figure: Alto Layout

Each of Page element can contain any number of objects like lines, images, or textblock and more. TextBlocks are divided into TextLines, and those are divided further into strings. Below is an example ALTO Layout section that we will be working with.

Figure: Alto XML layout

Each Page element contains an ID, HEIGHT, and WIDTH. This information is useful to know to keep track of the page number and the dimension of the page. Likewise the PrintSpace has an ID, HEIGHT, and WIDTH. Also, PrintSpace has HPOS and VPOS, which is the horizontal and vertical position respectively. All of the articles in the

newspaper will be within this element. The TextBlock element contains a set of TextLine elements, which contain any number of String elements that hold the article data. Furthermore, HEIGHT and WIDTH is available for all of these elements.

Forming an Article

One of our sponsor's requests was to group together the various words extracted from the OCR into distinct articles to be stored in the database. Before being able to do that, we need to figure out exactly how to group these words together in a coherent way as to form an article.

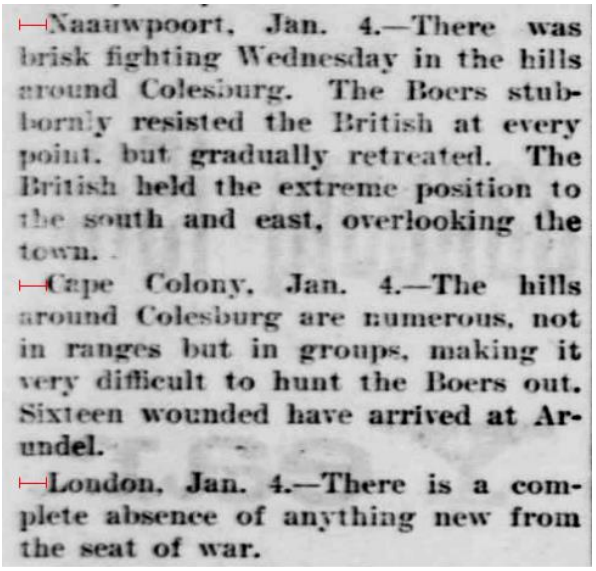
A big issue here is that, without any restrictions or guidelines to help us decide what is considered an article or not, we are left to observe common patterns in the corpus and decide for ourselves. This has led us to consider the very philosophical questions, "What *is* an article, really?", and "What does it consist of?". We tried not to impose too many arbitrary restrictions in order to maximize the amount of material that can be searched, however we still want to exclude advertising, headlines, and uselessly small articles.

As such, we have delved into what some of the articles in several different newspapers look like in order to get an idea of what we are working with. We have found that despite some wide variety in style between them, there is still some common patterns among them.

Common structures and patterns

To develop a parser that will work on the vast majority of articles, we need to know what common structures and patterns we can find in the data. To do this, we looked at some newspapers images, along with their generated ALTO files.

After studying a small sample, we have found that the most consistent structure in a newspaper article is the paragraph. A paragraph consists of normal text with some indentation on the first line, and is justify-aligned (ie. words align to the left and right margins).



Naauwpoort, Jan. 4.—There was brisk fighting Wednesday in the hills around Colesburg. The Boers stubbornly resisted the British at every point, but gradually retreated. The British held the extreme position to the south and east, overlooking the town.

Cape Colony, Jan. 4.—The hills around Colesburg are numerous, not in ranges but in groups, making it very difficult to hunt the Boers out. Sixteen wounded have arrived at Arundel.

London, Jan. 4.—There is a complete absence of anything new from the seat of war.

Sample of paragraphs showing their indentation (with red marks) and their justify-alignment

INDUSTRIAL FINANCE COMPANY HAVE NEW BUILDING NEARLY READY

Building Nearly Ready Expect to Finish Next Week Machinery be Here Monday or Tuesday.

Yesterady practically all the iron was put on the big building of the Industrial Finance company, inclosing it

Just one way in which titles (and possible subtitles) can be formatted in an article.

All other features are too inconsistent to form any reliable rules about them. Sometimes articles have the author under the title at the beginning, and sometimes at the end in a right-aligned format, usually in small caps. Sometimes there is no author specified at all.

Usually, horizontal lines separate articles, with smaller horizontal lines separating the article title from their content. Often, horizontal lines vary in their style and length, even between columns of the same newspaper. Sometimes there is no horizontal lines separating articles at all. Sometimes, advertisements are what separate articles.

Often, titles are bold and center-aligned. Sometimes there is more than one title, each having a different size or font weight. Sometimes, there is no title at all.

Despite these obstacles, some overall structure can still be derived. In the next few sections, we explore ways to gain as much information as we can and take as much advantage as possible of the little consistency there is.

Rejoining split words

In order to get a good understanding of what transformations we will need to perform when importing articles into a database, we must first know what the OCR software is doing for us and what the ALTO files are already telling us.

Initially we thought we would need to rejoin words that have been hyphenated and split across multiple lines, but luckily for us, the OCR already encoded that into the ALTO file. It gives us the whole rejoined word as an attribute.

So the rest of this section is dedicated to documenting how the OCR software seems to be operating, some edge cases it might have missed, and possible methods we can implement on our end to deal with them, should they arise.

OCR Behavior

To be able to rejoin words that have been split up across lines, the software must first detect that a word has been split. This is easy to do when a word fragment at the end of a line ends in a hyphen (“-”). The following line must then match the style of the previous one and start with another fragment that completes an actual word. There exists the possibility for a hyphen to be used as a stylistic choice to separate lines rather than to represent a long word that has been split up, although that’s likely to be *very*

rare. Dictionary verification is likely used to decide whether it is a word that has been split up or not.

Due to the way the ALTO files are organized, it is relatively easy to traverse consecutive lines of text and search for split words. We suspect that the general behavior of the encoder goes something like this:

For each TextLine element *line* in the document:

1. Let *first_segment* be the last string in *line*.
2. If *first_segment* does not end in a hyphen (ie. "-"), return successfully.
3. If there are still TextLine elements in the same TextBlock element:
    a. Let *last_segment* be the first string in the next TextLine element.
    b. If *first_segment* (without the hyphen) concatenated with *last_segment* form a valid word:
        i. Replace the word segments with the word itself in the stored plaintext of the article
        ii. Return successfully
4. Let *top* be the size of the top margin, and *bottom* be the size of the bottom margin.
5. If *top* is greater than the *bottom* margin, or if *line* is on the last column:
    a. Let *page* be the next page in the sequence.
6. Else:
    a. Let page be the previous page in the sequence.
7. Let *nextline* be the first text line on the next column of *page*.
8. Let *last_segment* be the first string in *nextline*.
9. Try concatenating *first_segment* with *last_segment*, and if it is a valid word, return successfully.
10. Otherwise, return with an error

There is one edge-case to take care of, however: What if the last line of a page has a split word? Depending on whether the newspaper's page has been segmented into multiple files, this means the software would either need to check the top of the next column, or look at the ALTO document for one of the other segments of the page, which could either be the previous sequence or the next one. Indeed, this seems like a case of diminishing returns.

Including hyphenations in queries

Since there are likely to be cases where the OCR could not find a matching segment for a hyphenated word, we could give the user the ability to search for a word with possible hyphenations in-between. A simple approach would involve choosing points at which a long word could likely be split, inserting a hyphen, and looking for all such possible pairs of segments in a document. As an example, for the word "imagination", an equivalent query might look something like this:

"imagination" OR ("ima-" AND "gination") OR
("imagi-" AND "nation") OR ("imagina-" AND "tion")

The rules for where to hyphen a word in English are slightly arbitrary and sometimes depend on which dictionary one is referring to, but there are some common patterns that can be followed. Here is a general guideline [23]:

- Break words at morpheme boundaries (*inter-face*).
- Break words between double consonants (*bat-tle*).
- Never separate an English digraph (e.g., th, ch, sh, ph, gh, ng, qu) when pronounced as a single unit (*au-thor* but *out-house*).
- Never break a word before a string of consonants that cannot begin a word in English (*jinx-ing* and not *jin-xing*).
- Never break a word after a short vowel in an accented syllable (*rap-id* but *stu-pid*).
- **Last resort, Maximal Onset Principle**: If there is a string of consonants between syllables, break this string as far to the left as you can (*mon-strous*).

There are certain issues with searching words using queries like this, however. A long word that generates a query to include possible splits can give results that do not actually have that hyphenation of the word.

Take the term (“imagi-” AND “nation”) in the previous example. This term would include results that have the word “nation” and some form of “imagi-”, which can include “imagination”, “imaging”, “imaginary”, or some other word that the user is not looking for. That being said, this edge-case is likely to be rare.

In the worst case, we could let the user search for certain fragments themselves if they feel it is necessary. Choosing where a search word should be split will be slightly difficult, but definitely not as difficult as searching around files, looking for where a possible second segment of a word could be located. As such, we’ve decided to replace the first fragment of a word with the word chosen by the ALTO file, and the second fragment will be removed.

Deriving structure from font style

Articles have a predictable pattern where titles, headings, and such at the beginning can have a very large font size (or just a bold font weight), and as one goes down the article, gets smaller until it reaches the actual content of the article. There are exception (sometimes the first few paragraphs are bigger than the later paragraphs, for example), but this is a fairly consistent pattern across different newspapers.

We can also compare the properties of the font style with the rest of the text to determine whether the text we are currently processing is “normal text”, and should therefore be excluded from the title field of an article. We define “normal text” to mean the font style used for text content that does not hold any special semantic information, and it is determined by whichever TextStyle element is referred to the most throughout the ALTO file.

Combining these two ideas, we have a reliable and generalized way to determine when the title of an article starts and ends. We can start with a title with a big font size; a reduction in size signals the software to treat everything before that point as the title and everything after as the content. The ending is easily detected when the article eventually goes from using the “normal text” style to a different font style, usually by an increase in size and/or weight, or suddenly encountering centered text.

Unfortunately, during testing we found that this was very finicky and often unnecessary. The OCR software already does a surprisingly good job at separating chunks of text into blocks, which often coincide with articles. In the case where an article would be split into pieces, the user could still find parts of the article with a direct link to the PDF.

Article titles are usually short, so we will place an estimated limit of 256 characters for the title field. If we find that a significant number of titles surpass this limit, we can increase the length to accommodate.

Dealing with advertising

A sore sight often seen when glancing at a newspaper page is that of an advertisement. These colossal wastes of space and hoarders of humans’ eyes can be a nuisance when parsing articles, especially so when they interrupt an article in between paragraphs.

Advertising section often have very big font sizes, and large sections without any text, which are often left for pictures or drawings. They also very often have borders surrounding them. This makes it very easy to identify them visually, but because ALTO does not encode borders into its files, we can not use that as a way of filtering them out.

Due to the way we plan to design our software, there is a high possibility for ads to show up as “articles” if they are formatted in just the right way.

One can almost tell apart the ads from the articles

One easy method of resolving this would be to check if an “article” uses the normal text of a newspaper. Since the normal text for newspaper articles are usually very small (in order to fit as much information as

possible) or very boring (ads tend to prefer using fancy or stylish fonts that stand out and grab the user's attention), this could serve as a very simple and reliable filter for our purposes.

Another method is to set a minimum size for articles. Because newspaper articles are very long in textual length compared to ads, we could simply set a minimum number of lines; if too few, the "article" is discarded. In our testing we've found that any legitimate articles are almost never less than 4 lines in size, so *very* little information would be lost.

XML parser

For this project we need to decide if the parser should be created using Java or Python. There are many different types of XML parsers available to use. In order to choose the best parser for this project, let us examine three different types of parsers: DOM, SAX, and StAX. To demonstrate the different way these are implemented in Java and Python, an example will be provided for each.

DOM Parser

DOM parsers let programs access the style, structure, and contents of XML documents. The parser is often used when the user needs to know the structure of the document, when the user needs to sort elements of the XML document, or when information from the XML document needs to be reused multiple times (Java DOM Parser Overview).

When the parser is used on an XML document, it returns a tree structure with all the elements of the XML document. The parser then supports different types of functions that allows the user to examine the contents of the tree structures. One of the key advantages of this parser is that the code written in Java for a DOM compliant parser can be reused in another DOM compliant parser. [1]

According to TutorialPoint, the following steps need to be taken when parsing a document with a Java DOM Parser:

- Import XML-related packages.
- Create a SAXBuilder.
- Create a Document from a file or stream.
- Extract the root element.
- Examine attributes.
- Examine sub-elements.

SAX parser

SAX, Simplified API for XML, is an event-based parser for XML documents. Unlike the DOM parser, this parser does not create a parse tree. Instead it reads the XML file from the top down, element by element, in sequential order. "The SAX is a streaming interface for XML that receives event notifications about the XML document being processed an element, and attribute".

One of the main reasons to use a SAX parser is if the files beings parsed are extremely large. Due to the lack of a parse tree, the SAX parser uses less memory. The disadvantage of this implementation is that it does not allow going back and analyzing previous elements once they are passed. This is because the parser is like a stream; it only goes forward. [2]

TutorialPoints outlines the features of SAX API:

- Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document.
- Tokens are processed in the same order that they appear in the document.
- Reports the application program the nature of tokens that the parser has encountered as they occur.
- The application program provides an event handler that must be registered with the parser.
- As the tokens are identified, callback methods in the handler are invoked with the relevant information.

Unlike the DOM parser, the SAX parser requires more than one class. The main class, and the handler class. The handler class will extend the DefaultHandler, which is the default base class for the SAX event handlers.

In Python, to parse an XML, SAX requires that the developer creates his own ContentHandler by subclassing xml.sax.ContentHandler. This ContentHandler will be responsible for all the different types of tags and attributes of the XML file. In addition, the ContentHandler object will handle all the various parsing events. The parser calls the ContentHandler methods as it parses the XML file. The methods startDocument and endDocument are called at the beginning and at the end of the XML file respectively.

The character(text) method allows character data of the XML file via the parameter text. Furthermore, at the beginning and end of each element the ContentHandler is called. The methods startElement(tag, attributes) and endElement(tag) are called if the parser is not in namespace mode. Otherwise, the methods startElementNS and endElementNS are called. [4]

According to TutorialPoint, the following methods are very important to know to understand how the Python SAX parser works.

- xml.sax.make_parser([parser_list]) : This method creates a new parser object and returns it. The object will be the first parser type the system finds. The parser_list parameter is an optional argument that consist of a list of parsers to use which must implement the make_parse method.

- xml.sax.parse(xmlfile, contenthandler[, errorhander]) : This method creates a SAX parser and uses it to parse a document. The parameter xmlfile is the name of the XML file to read from. The parameter contenthandler is a ContentHandler object. The parameter errorhandler is an optional SAX ErrorHandler object.

- xml.sax.parseString(xmlstring, contenthandler[, errorhandler]) : This method parses the specified XML string. The parameter xmlstring is the name of the XML string to read from. The parameter contenthandler is a ContentHandler object. The parameter errorhandler is an optional SAX ErrorHandler object.

StAX Parser

StAX XML parser is an API that parses XML documents similar to the SAX parser. However there are two major differences between them. First, StAX is a *pull* API and SAX is a *push* API. This means that for StAX, the client application needs to inform the StAX parser to get additional information from the XML document when it needs more. However, the SAX parser requires the client application to get more information when SAX parser notifies the client application that information is available. Second, unlike the SAX parser that can only read an XML documents, the StAX parser can read and write XML documents. Like the SAX parser, StAX should be used if the XML document is really big. [3]

TutorialPoints outlines the features of StAX API:

- Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document.

- Tokens are processed in the same order that they appear in the document.

- Reports the application program the nature of tokens that the parser has encountered as they occur.

- The application program provides an "event" reader which acts as an iterator and iterates over the event to get the required information. Another reader available is "cursor" which acts as a pointer to XML nodes.

- As the events are identified, XML elements can be retrieved from the event object and can be processed further.

Choosing a Parser

The main difference between the DOM parser and both the SAX and StAX parser is the element tree. The DOM parser creates this tree when it reads the complete XML file; whereas, the SAX and StAX parsers skip this step and instead they read the XML file like an input stream. Thus, the DOM parser ends up consuming more memory than both the SAX and StAX parsers. However, the DOM parser has the ability to look at any element in the tree; whereas, the SAX parser can only read the elements as they come up, and is unable to go back to previous elements. This means that the DOM parser is more flexible XML parser than the SAX parser.

The ALTO XML files that will be parsed will be really big. With the DOM parser will be able to create an algorithm more easily thanks to the flexibility of the parser. However, the system will be using a lot of resources because the amount of memory that the DOM parser consumes. In this situation, the only choice will be either the SAX or the StAX parser.

SAX and StAX are very similar parsers, they are both stream event driven oriented XML parsers. However, they have a subtle difference in that SAX uses a *push* model and and StAX uses a *pull* model. For SAX, the parser calls the handler; whereas, for StAX the handler calls the parser. This means that with the *push* model you have no control over how the parser iterates over a file, and that with the *pull* model you have control over the parser and can stop parsing at any time. However, due to the nature of this project, we will need to parse the entire XML file to make sure that all articles are parsed.

The best parser for our project will be the SAX parser. This is because the size of the XML files are really big and the SAX parser is very memory friendly.

Choosing Python or Java

Both programming languages support the SAX library. Either one of the languages will work and in terms of difficulty they are both about the same. However, the server that we will be using already supports Python, making it convenient to choose Python. Python is also a beginner-friendly programming language with little verbosity, making it easier to immediately understand what is happening for anyone contributing to this project, irregardless of programming skill. Due to this, the programming language that will be used is Python.

Populating the article database

In order to populate the database with parsed articles, we need to write a program that will receive the information of an article from an endpoint at https://chroniclingamerica.loc.gov/articles/store and will generate the SQL statements necessary to insert the information. We will call this endpoint the “articles store”.

The information received will be in the form of a JSON array containing objects describing the properties of those articles. This array will allow us to batch and import several articles at a time. The program will report any errors with a status code and a description of the error.

The code will be relatively straightforward:

1. Let *insertions* be the string that will hold the SQL insertions statements.
2. Append the table and necessary syntax to *insertions*.
3. For each *article* in the array:
    a. Let *lccn* be the LCCN number of the newspaper the article is from.
    b. Let *metadata* be the necessary metadata from the newspaper pointed to by *lccn* that needs to be stored with the article.
    c. Let *stmt* be the SQL statement string formatted to contain the title, content, all the necessary *metadata*, etc. that will be stored with the article.
    d. Append *stmt* to *insertions*.
4. Run the insertions string through SQL.

Error handling

Any problems encountered (e.g. reading the JSON, unexpected values, etc.) will have the program return an error message to the standard output about what happened. There are procedures in place for handling any of the errors that the parser can run into.

Any newspaper pages that result in an error – be it from online resource that aren’t reachable, or files that couldn’t be saved locally – will be skipped. No data from those pages will be inserted into the articles table. However, the parser can tell when duplicate data is found, so it can process the same data again without any issues and get any pages that it missed.

Successful parses will have all local files related to the individual page (like the ALTO and image files) removed from the filesystem in order to save space.

MySQL multithreading

Due to the size of the database and in anticipation of the number of queries that might run through it, we figured that multithreading SQL queries would improve the performance of the system.

Although MySQL uses threading internally to manage its resources, it still uses only one thread per user when fetching information. MySQL itself is not designed to split queries or merges and run them simultaneously to save on computing resources. Unfortunately, no known options exist within MySQL to enable such functionality because it's not how MySQL was designed in mind.

There is, however, a method to improve its parallelism without having to depend on how MySQL works. By using a shell script to run multiple MySQL commands, the operating system will automatically parallelize the tasks by using the native thread manager to run the tasks at the same time when possible. This results in relatively faster execution times and less reliance on a single CPU core. [14]

```
#!/bin/bash
date
for y in {1988..2013}
do
  sql="select yeard, count(*) from ontime where yeard=$y"
  mysql -vvv ontime -e "$sql" &>par_sql1/$y.log &
done
wait
date
```

Here are the results:

```
Start: 11:41:21 EST 2014
  End: 11:41:26 EST 2014
```

So the total execution time is ~5 (10x faster) seconds.

Performance improvements running MySQL commands from a shell script

However, this is a very ad-hoc way of improving performance, and would require execution of a bash script whenever a query is performed. Because of this, we have decided that we will not be pursuing the idea, and will instead try to improve server performance elsewhere.

Salting passwords

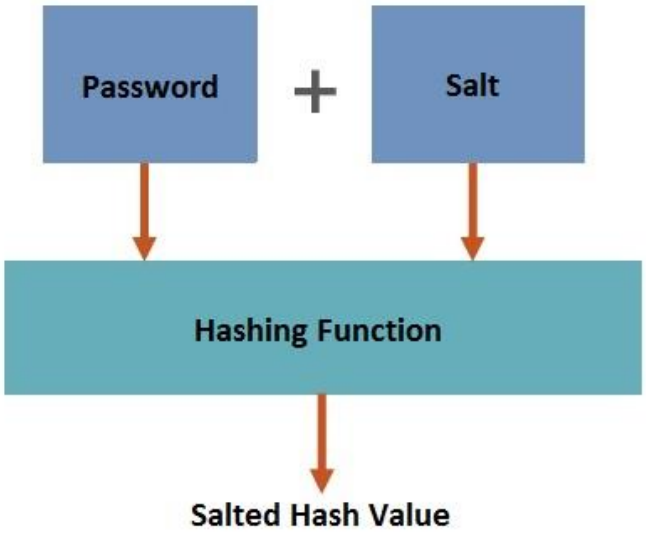
It seems like every week or so, there is a new security breach at a company or startup. There have been many instances where user information was not stored securely on private servers, and when they are inevitably compromised, it is the users who suffer the most.

We want to ensure that the service we build will be safe for those that will use it. As such, we will be salting user passwords in an effort to help protect users' personal information in the case a breach should occur.

The idea of salting passwords is very simple. Instead of storing passwords in a plaintext form (in other words, easily readable to anyone that has access to the database), we store a *salted* form of the password.

$$salted\_pw = \text{hash}(password \mid salt)$$

Above is the formula used to get the salted form of a password. A salted password is the result of attaching (signified by the vertical bar) a random string of characters unique to the user (called a *salt*) to the user's password, and then running this new string through a hashing function. The result of the hashing function is the password in salted form, and is stored in place of the plain password. Whether the salt is attached on the front or the back does not have much influence. The salt is stored in the database with the user, and as a general rule, the longer it is, the more secure it is. [15]



Visual of how the salting functionality is achieved

Stretch goals

As mentioned previously, part of this project consists of three different stretch goals. Unfortunately, we decided to not actively pursue them, due to the scope and size of the project itself and the time limit in order to implement it all.

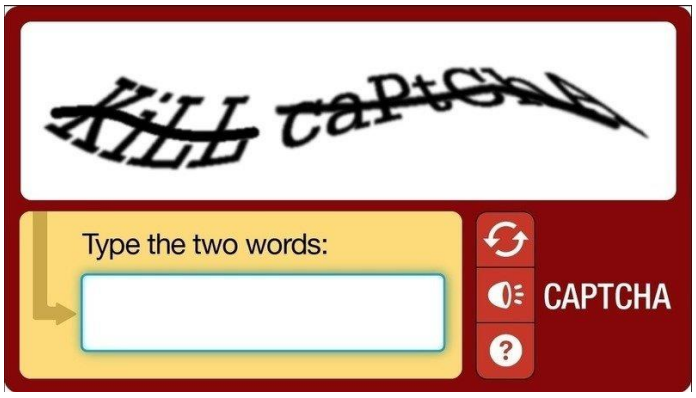
However, we will give a brief overview of ways that the project and some of its parts could be modified in order to implement the stretch goals in the future, should some other team after us decide to do so.

Error correction

One of our stretch goals is to come up with a way to clean up errors made by the OCR software. This can happen regularly when the newspaper is faded and the OCR confuses certain characters or parts of a word for other characters. Corrections like this are achievable using automated or crowd-sourced methods.

Automated Corrections

We suspect that a significant part of these errors can be corrected easily using automated methods. In order of increasing complexity, these include methods such as dictionary lookup, trying multiple types of OCR software, or some form of machine learning AI that notices patterns where errors repeat for certain characters.



CAPTCHAs rely on two words to be deciphered

Crowd-sourced corrections

In addition to trying automated methods, we have also come up with an idea for how to do this effectively using crowd-sourced methods.

Since one of our requirements states that there must be a way for users to create accounts on the website, there exists the possibility of automated account creation using bots that could use up processing time or storage resources. A known way of

filtering out the bots from the humans is to use a CAPTCHA, where a human would be able to read an image and enter what it says, but a bot would not.

Our idea is, since it just so happens that we need to correct OCR errors and have a way to prevent automated account creation, we could build a sort of custom CAPTCHA using the image and OCR data we already have. We can easily create an image for some words using the position and size information provided by the ALTO files and the image files for their respective newspaper scans. However, this would require that we store that data in the database, which will likely use a lot of space.

Word choice algorithm

The way this would work exactly would be randomly choosing two words from the ALTO files, one with a high confidence value and another with a low confidence value. Like traditional CAPTCHA, the word with the high confidence value is treated as one whose value we already know. We treat this word as a sort of verifier that the words on the image are being read. The input the user provides for the word with the low confidence value is recorded to help determine what the value of the word is.

The general algorithm for how to decide which word to gather input for based on its confidence is as follows:

1. Choose a random ALTO file to read
2. Let *uncertain_words* be an empty list
3. For each *word* in the ALTO file:
    a. Let *confidence* be the confidence value stored in the file for that word
    b. If *confidence* is between 0.05 and 0.95:
        i. Let *weight* = 1 - |0.5 - *confidence*|
        ii. Create a *Word* object storing the *word*'s position, size, and weight
        iii. Add *Word* to *uncertain_words*
4. Sort *uncertain_words* by decreasing weight values
5. Let *chosen_word* be a *Word* chosen from *uncertain_words* using 1 - |0.5 - *confidence*| as the equation for the probability weight

Accessibility issues

Although this is a very convenient idea, we must also keep in mind those who are not able to complete CAPTCHAs due to a disability. CAPTCHAs are somewhat known for being inaccessible for those who wish to sign up for an account online. Besides, if the answer were easily given those users, bots could just as easily cheat the CAPTCHA by getting the answer that way. Feedback shows that sound CAPTCHAs can be just as bad, if not worse than, visual CAPTCHAs. [26]

A possible solution for this is to give the user the option to fill in a text-based question or statement that informs the person of what word(s) they must enter. This requires the question or statement to have high variety and randomness in its phrasing and formatting to ensure that bots do not catch on to any patterns that would make it predictable. [27]

There are several other methods to perform a Turing test filter for account signup forms, as has been documented by the W3C. [28]

Database changes

To fetch the words to display to a user, we need to know its size and location (where on the page, which page, which newspaper, etc). To do this quickly, it must be stored on the database, and that requires a few changes.

Structure layout

There must be a new table to store all words parsed from the ALTO files. We will call this the Word table. It will consist of a primary key, the value of the word itself, the OCR's guess as to what the word is, the confidence value, width, height, coordinates on the page, a single string field for the path of the page in the usual format used in the API (LCCN, date, edition number, and sequence number, in that order), and a boolean indicating if the value of this word has been officially verified by an administrator.

There must be another table which will hold the guesses made by users for a particular word. We will call this the UserGuess table. UserGuess consists of a primary key, a foreign key that points to a word in the Word table that the user tried to correct, the string value of the guessed word, and how many people have guessed this value.

The reason for having a single field for the path to the page in the Word table is that it will allow us to get directly to the page without piecing the data back together. Just prepend the hostname, append the path to the image file, and it can be fetched immediately.

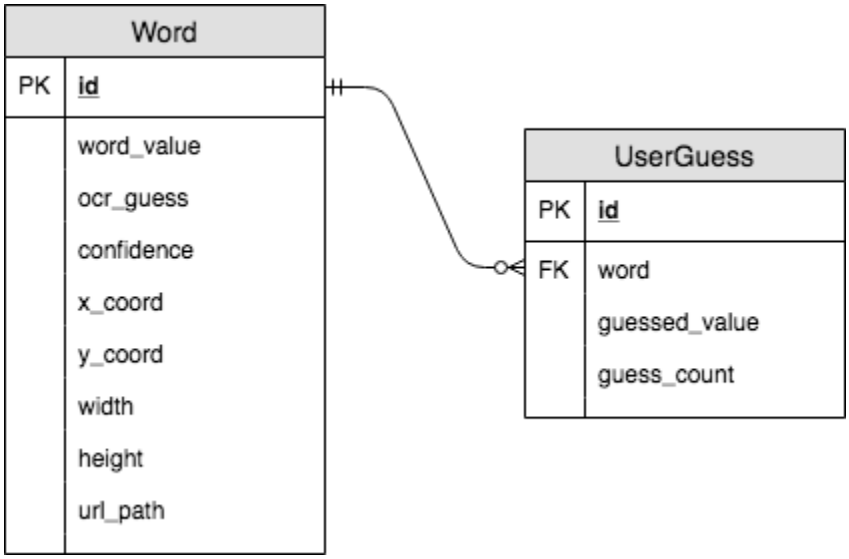


Table structures and their relationship

Rules for updating tables

When a user guesses what a word says, a request will be sent to either insert a new row with a count of 1 (in case the guess is entirely new and is the first time being used), or a currently existing row will be found and its count will be incremented by one (for when the guess has already been tried by at least one other person).

For a CAPTCHA word with a low confidence value (less than 95%), when the total count for all the guesses reaches more than 100, the most commonly submitted answer will be checked against the OCR answer.

If the submitted answer has a higher percentage than the confidence value given by the OCR, the value of the word in the Word table is changed to be that of the correction submitted by the users. Otherwise, the count continues until the confidence value is surpassed, or when an administrator has verified the value of the word.

Estimating size

Because of so much data being stored *per word*, we will need a large amount of space to store it all. But to get an idea of how much, we will perform an estimate.

First, let's find the size of each column in the Word table and add them up to calculate the total size for a table row. We did not include the verification column here because its size is negligible. [9] See the break down table below:

Properties for Word table		
Name	**Type**	**Size (bytes)**
id	UNSIGNED BIGINT	8
word_value	TINYTEXT	256
ocr_guess	TINYTEXT	256
confidence	FLOAT	4
x_coord	UNSIGNED SMALLINT	2
y_coord	UNSIGNED SMALLINT	2
width	UNSIGNED SMALLINT	2
height	UNSIGNED SMALLINT	2
url_path	[VAR]CHAR(34)	35
	Total:	**567**

This gives us a maximum total of 567 bytes per word. The About page for the Chronicling America API links to a newspaper page with 6528 String elements. For our purposes, we will treat them as individual words. This would give us a size of 3,701,376 bytes per page, or about 3.7 megabytes per page.

The edition this page comes from has 8 pages, and if we assume they have a similar number of words per page, would give us a total of about 29.6 megabytes per edition. This edition is also part of LCCN sn86069873, which contains 2431 issues. If we assume they are all similarly sized as the one we calculated, that means that storing sn86069873 would require almost 72 gigabytes.

Another way to interpret the data is to look at the total number of pages in the Chronicling America website. As of the time of this writing, the website says that there is 14,329,958 pages available. If we assume the pages have an average number of words similar to the first page we calculated, that would give us about 53 terabytes total.

Note that this would not include the size needed to store all the guesses created by users. Also note that this is still an estimate based quite a few assumptions, and that pages will not always consist of that many words; there may be whole pages that have pictures or drawings and have relatively few words, which would lower the actual size needed to store this table.

Mobile application

Our sponsors expressed interest in developing a mobile application version of the website. This would allow users an easily accessible and convenient way to access the same data, without having to go through a web interface on a mobile browser.

The goal consists of creating an application for Android, iOS, and/or Windows Phone. One of our sponsors prefers a Windows Phone application that she could personally use on her phone.

The biggest issue with this goal is trying to include the Windows Phone platform. Most software platforms that can deploy applications to various mobile operating systems do not support Windows Phone, since Microsoft has dropped support for it in favor of the Universal Windows Platform (UWP). [10] As a result, they tend to stick to just Android and iOS.

If truly desired, it is entirely possible to both port the functionality of the website to various Microsoft platforms using UWP, and to all other platforms using a different technology. For Android and iOS, we would use a programming language called Haxe.

When porting to several platforms, it’s a good idea to use a tool that is designed to save some of the work. There are several software packages that allow a developer to write in one language and compile it to several other languages, allowing one to run it in many different platforms.

Haxe is one such tool. It supports compiling to Android and iOS, and is used by a number of notable companies, such as BBC, Coca-Cola, Disney, Hasbro, Mattel, Nickelodeon, Prezi, TiVo, Toyota, and Zynga. [11]

From here, it would be a relatively straightforward process of just designing the different “activities” or screens that a user will be able to see, and hooking up all the relevant elements on the activities to the functionality and services that are provided by Chronicling America.

Supporting multiple languages

Due to the way the currently-planned project would work, it is possible to allow for multiple languages to be scanned and imported into the database without any further configuration on anyone’s part. The language it is written in is straightforward to get from its ALTO file and store in the table.

However, if corrections through user contributions is to be implemented, then there are some issues to address first. For example, we cannot have users try to decipher a word for a language they do not know (at least, not usually). It is likely that letters from a language other than English, for example, will not be available on an English speaker’s keyboard layout.

Therefore, we will need to know the language the newspaper is in, and the languages that a user can understand. The table that holds the user’s information will need to be changed somehow to include a list of languages they understand. How this is done is left as an exercise to the implementer. Programs will refer to this information when searching the database for words that a user can try and decipher.

If desired, the user could also input any words that use the same script from the one their language uses. For example, except for the addition of one character (ñ), the Spanish alphabet is identical to that of English. Therefore, since the English alphabet is a subset of the Spanish alphabet, a Spanish-speaking person could fill in any words that show up from an English newspaper.

Project Design

It is important to chose to use technologies that are well supported and can enable the completion of all the requirements. Older libraries will have wider support across systems, but sometimes are discontinued or have slowed in development, whereas newer libraries are packed with recent standard ideas, but sometimes lack quick adoption. A delicate balance is necessary for the successful design of this project.

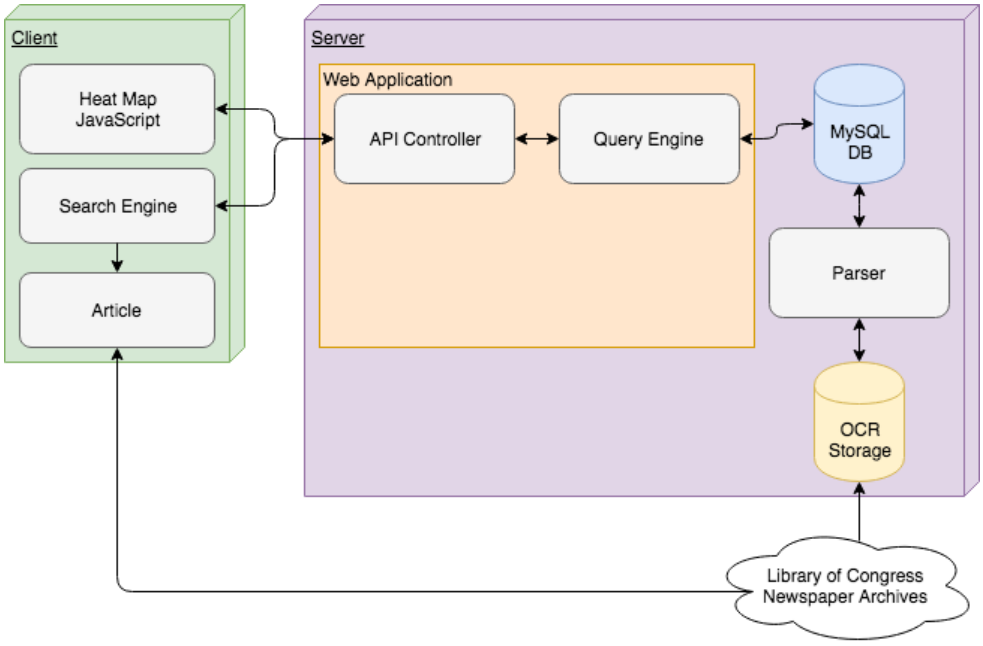
We aim to build a Model-View-Component (MVC) approach for the website.

Block Diagram

Below is the block diagram for this project. It shows the general architecture of the entire project. All blocks are currently being researched. It will operate similarly to many websites in a stack. The API connects the front end to the backend. The Parser will process articles from the OCR storage. The database will be storing user information as well as users' saved queries.

The database will also be storing parsed articles, their metadata, and the full text of the article. The Search Engine will be responsible for passing the user search terms to the server to be expanded into detailed queries by the query engine. The results passed back to the client will include ranked items of each article.

Selecting an article will display the raw text and provide a link to the Library of Congress to view the original archived newspaper page. The Heat Map component will ask the server to organize articles by geographical location, and display the data as a heat map over a geographical map.



Project Specifications

Back End

Laravel

Laravel is an open sourced cross platform web application framework that aids in the creation of elegant websites through expressive and creative code. Laravel attempts to handle the workload of common tasks to ease development of projects. Laravel comes with an extensive documentation set and video series that allows fast learning of the framework. Laravel's features include:

- Simple, fast routing engine
- Powerful dependency injection container
- Multiple back-ends for session and cache storage
- Expressive, intuitive database ORM
- Database agnostic schema migrations
- Robust background job processing
- Real-time event broadcasting

We will utilize Laravel only as a Restful API. The routing will be handled by React router.
Rather than serving HTML or PHP views, we plan to use Laravel to deliver powerful JSON APIs.

Front End

ReactJS

React is an increasingly popular JavaScript library for building wonderfully complex user interfaces. Originally developed by Facebook and open-sourced to the community, React employs the following ideals that will aid in our project:

- **Declarative:** React makes it painless to create interactive UIs. We can design simple views for each state in our application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable, simpler to understand, and easier to debug.
- **Component-Based:** By building encapsulated components that manage their own state, we can compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, we can easily pass rich data through the app and keep state out of the DOM.
- **Learn Once, Write Anywhere:** React does not make assumptions about the rest of the technology stack, so we can develop new features in React without having to rewrite existing code.

We will use ReactJS to render all html to the client.

Authentication

We will use JSON Web Token (JWT) for user authentication. Laravel Restful API will generate the token and send it back to the user. Laravel will verify the JWT when a user makes an API request.

Project Budget and Financing

There is no budget for this project. This will be adequate. The database and website will all be hosted by the UCF Computer Science Department initially and then by the UCF Arts and Humanities Department later. This means that while there will be costs for hosting the website, it will all be handled by UCF's IT and Humanities Departments. We will just be using server space allotted to us.

None of the software we are using will have any licensing fees. For the project we will need a database. The database software we will be using is MySQL. MySQL only requires licensing when you are modifying its source code. We do not plan on modifying MySQL, therefore we won't be needing any licensing to use it on our server.

There are a plethora of free frontend frameworks we could use for the web application. One that we are considering using is Bootstrap 4. Bootstrap is a popular frontend framework developed by Twitter. Another option for frontend use is ReactJS. It is a JavaScript library developed by employees of Facebook. This is also free to use, so it would be a cost effective way of development. Outside of these options there are many more frameworks we could use that would not cost anything.

On the backend we need to create programs to parse through the XML files. Our options include Python and Java. Python is a free and open source scripting language. Java is a compiled programming language whose development kit is free to use in commercial programming.

Version control is imperative in this type of project. We plan on using GitHub for our version control. The University of Central Florida Department of Computer Science is providing free private repositories for these senior design projects. This is also convenient because the sponsors of this project will have access to these repositories once we are done.

Communication is key in any project. To communicate with our sponsors we will mainly be using the email service provided to us by UCF. For communication between group members we will be using Slack. Luckily, Slack has a free tier that provide more than enough features for our needs.

If we want to use SSL certification on our site, we need to find a free version of it. Comodo offers free trial services. If you want to use theirs for free it seems you need to renew it every 90 days. This could be inconvenient, and if the sponsors get a budget for the long-term upkeep of the website this may be one of the services they consider paying for.

This project will require a comprehensive, efficient, fast, and powerful search tool. By using Solr as a platform for this project's query tool, we can accomplish all these things. Solr is open-sourced and also requires no licensing fees to use for any purpose.

Overall, all the technologies being used in this project are mostly open-sourced, but all require no licensing fees to be used in this project. The only cost going forward could be for server space considering how large the project could end up being. In terms for this project though, the server space will be provided for us by the Computer Science Department initially, then the project will be transferred to the UCF Arts and Humanities Department's servers. Thus, this project requires a budget of $0.

Database Structure

For this project we will build a MySQL database. It is a relational database, therefore the planning of the database needs to reflect that. The entity relationship diagram is shown below in figure DB-1.

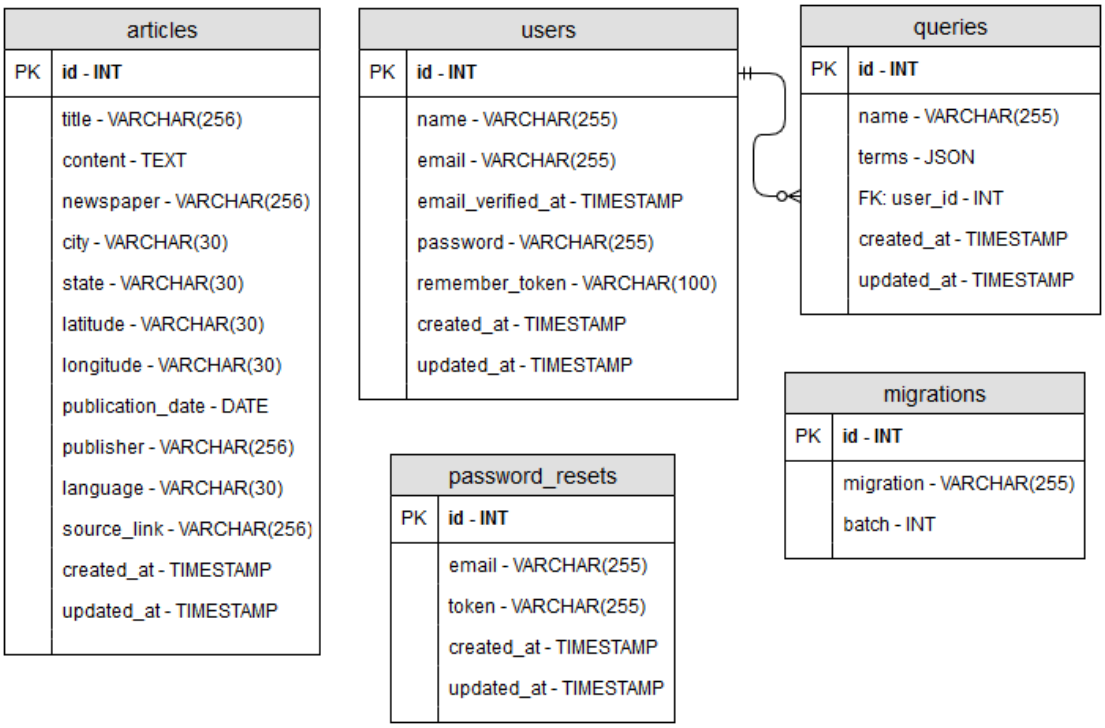


Figure DB-1: Entity Relationship Diagram
This diagram represents the database planned for this project.

Queries Table

- **id** - is the primary key for this table, meaning it must be unique and cannot be null. It is an integer and will auto-increment as each query is added.
- **name -** is the name that the user can choose to name the query, so they can identify it later. It is stored as a VARCHAR(255).
- **terms** - will be stored in a JSON format including search parameters in their respective fields. This includes the search terms, the operators, and any other search options.
    - I.E. “title: Rocks content: sandstone AND aquifer” could be a possible entry into this field.
- **user_id** - is a foreign key from the users table. It is the primary key in that table, where it refers to the user ID as an integer.

- **created_at** - will indicate the time and date a query was created. It will be recorded as a TIMESTAMP data type in 'YYYY-MM-DD HH:MM:SS' form.
- **updated_at** - will indicate the last time this record was updated. It will be recorded as a TIMESTAMP data type.

Users Table

- **id** - is the primary key for this table and will be in an integer data type. Will auto-increment.
- **name -** is the user's name. It is stored as a string of 255 or less characters.
- **email** - each users' email will be collected when they sign up to use the site, and it will then be entered in this field as a string of at 255 or less characters.
- **email_verified_at -** is stored as a TIMESTAMP and reflects the time and date when the user's email was verified at.
- **password** - when a user signs up for the site they will enter a password, it is recorded as a string of characters/text. It is stored as a string of 255 characters or less.
- **remember_token -** is stored as a string of 100 characters or less.
- **created_at** - it will indicate when the user's account was made and will be stored as a TIMESTAMP data type.
- **updated_at** - will indicate when the user information was last changed and will be stored as a TIMESTAMP data type.

Articles Table

- **id** - is the primary key for this table and will be an integer data type. It will also auto-increment.
- **title** - is the title of the article. It is recorded as a string of 256 or less characters.
- **content** - is the actual body of the article containing a majority of the text within the article. Because of its potential size it is being stored as a text data type, meaning it can hold up to 65,535 characters.
- **newspaper** - is the name of the newspaper the article was printed in. It is stored as a string of 256 or less characters.
- **state** - the geographical state the newspaper containing the article was printed within. It is stored as a string of 30 or less characters.
- **city** - is the geographical city that the article was originally published in. It is stored as a string of 30 or less characters.

- **latitude -** is the latitude that corresponds to the geographical location of the city this article was published in. It is stored as a string of 30 or less characters.
- **longitude -** is the longitude that corresponds to the geographical location of the city this article was published in. It is stored as a string of 30 or less characters.
- **publication_date** - the date the article was published. It is stored as a DATE data type in a 'YYYY-MM-DD' format.
- **publisher** - is the name of the publisher that published the newspaper containing the article. It is stored as a string of 256 or less characters.
- **language** - is the language the article is written in. It is stored as a string of 30 or less characters.
- **source_link** - is the link of the source of the article within the Chronicling America API. It is stored as a string of 256 or less characters.
- **created_at** - indicates when the article was added to the database. It is stored as a TIMESTAMP data type.
- **updated_at** - indicates when the article data in the database was last changed and will be stored as a TIMESTAMP data type.

Password_resets

- **id** - is the primary key for this table and will be an integer data type. It will also auto-increment.
- **email -** is the email of the user. It is stored as a string of 255 characters or less.
- **token -** is stored as a string of 255 characters or less.
- **created_at -** indicates when the article was added to the database. It is stored as a TIMESTAMP data type.
- **updated_at** - indicates when the article data in the database was last changed and will be stored as a TIMESTAMP data type.

Migrations

- **id** - is the primary key for this table and will be an integer data type. It will also auto-increment.
- **migration -** is stored as a string of 255 or less characters.
- **batch -** is stored as an int.

In this project the user needs to be able to save their query. That is why the user_id field is required as a foreign key in the queries table. This ensures that the query is

connected to the user that created and ran the search. The diagram illustrates that each user can have zero to many queries. Each query can only have a single user however.

The articles table does not have any relationships to any of the other tables in the database. Articles are not connected to users. It may seem like it could make sense to create a table connecting the articles and queries tables. However, the written text form of the search is the part that needs to be saved, not the articles that were returned. This allows the same query the ability to find new and possibly more relative articles that have been added to the database after the query was originally made.

Database Data-Types

The database that will be used in this project is a MySQL database. MySQL has many different data types to hold different types of values. This project requires that the entire article is saved in the database, so choosing the appropriate data type is important, so the database is as efficient as possible. We will look into the TEXT data type objects as a solution to hold vast amount of text strings.

NAME	**CHARACTERS**	**SIZE**	**OVERHEAD**
TINYTEXT	255	255 BYTES	1 BYTE
TEXT	65,535	64 KILOBYTES	2 BYTES
MEDIUMTEXT	16,777,215	16 MEGABYTES	3 BYTES
LONGTEXT	4,294,967,295	4 GIGABYTES	4 BYTES

The TEXT data type objects allowed to store long-form text strings in a MySQL database. There are four TEXT data type objects called TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT. The capacity to hold characters range from 255 to 4,294,967,295. The size can range from 255 Bytes to 4 Gigabytes. They have an overhead of 1 to 4 Bytes [6].

It is important to choose the optimal data type to make sure that we have enough space to store the article data, but not too much to make the database unnecessary big and slow. The best way to go about this is to analyze how many words an article takes and the average character length of english words. This is hard to estimate on older newspapers, but we can guess based on how modern stories behave [7].

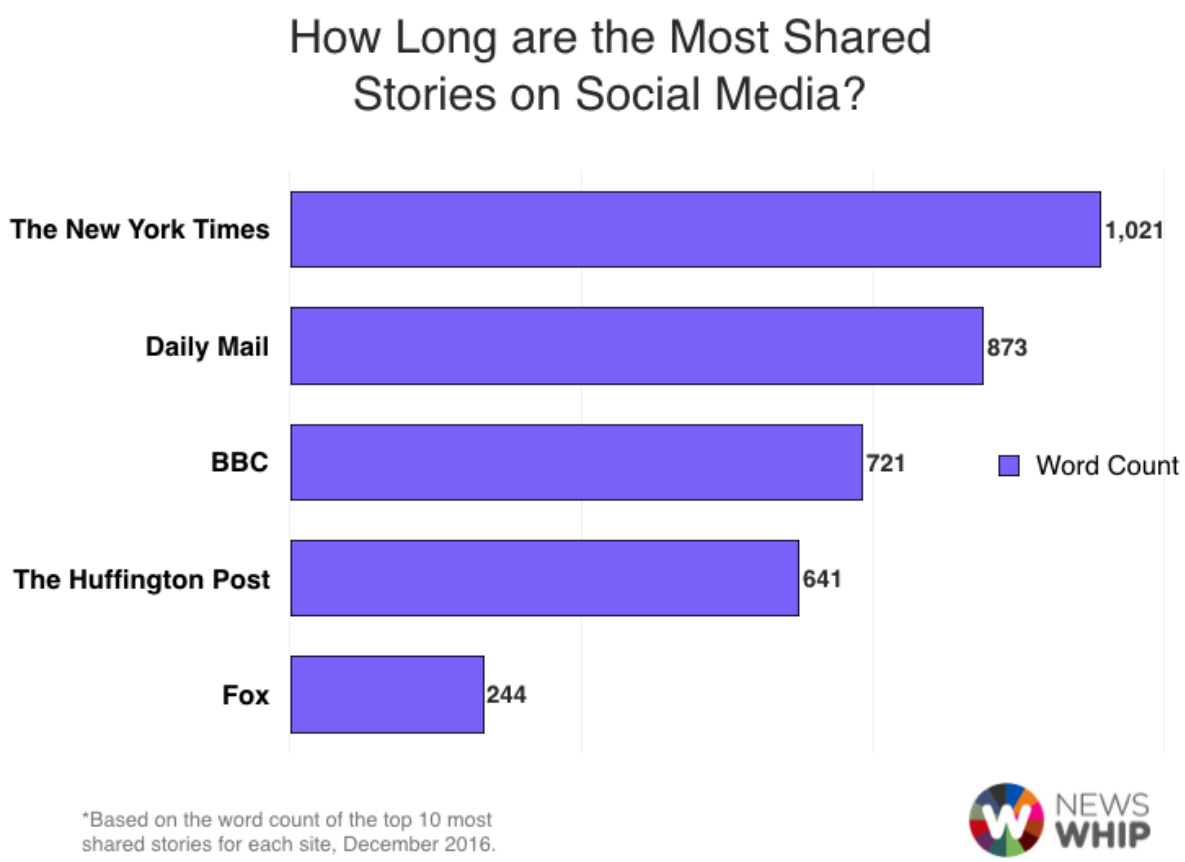


Figure: Newswhip Stories Length

This chart represents the word count of different news company and how popular the articles were. Taking the highest number being 1,021 words. We can double it to be safe, and we get a final word count 2,042. Then we have to find out the average character count in a word.

Using Google books data there was a total on 97,565 distinct word. Those words were mentioned 743,842,922,321 times.

The following graph represents the top 50 distinct words and how many times they were mentioned in billions [8].

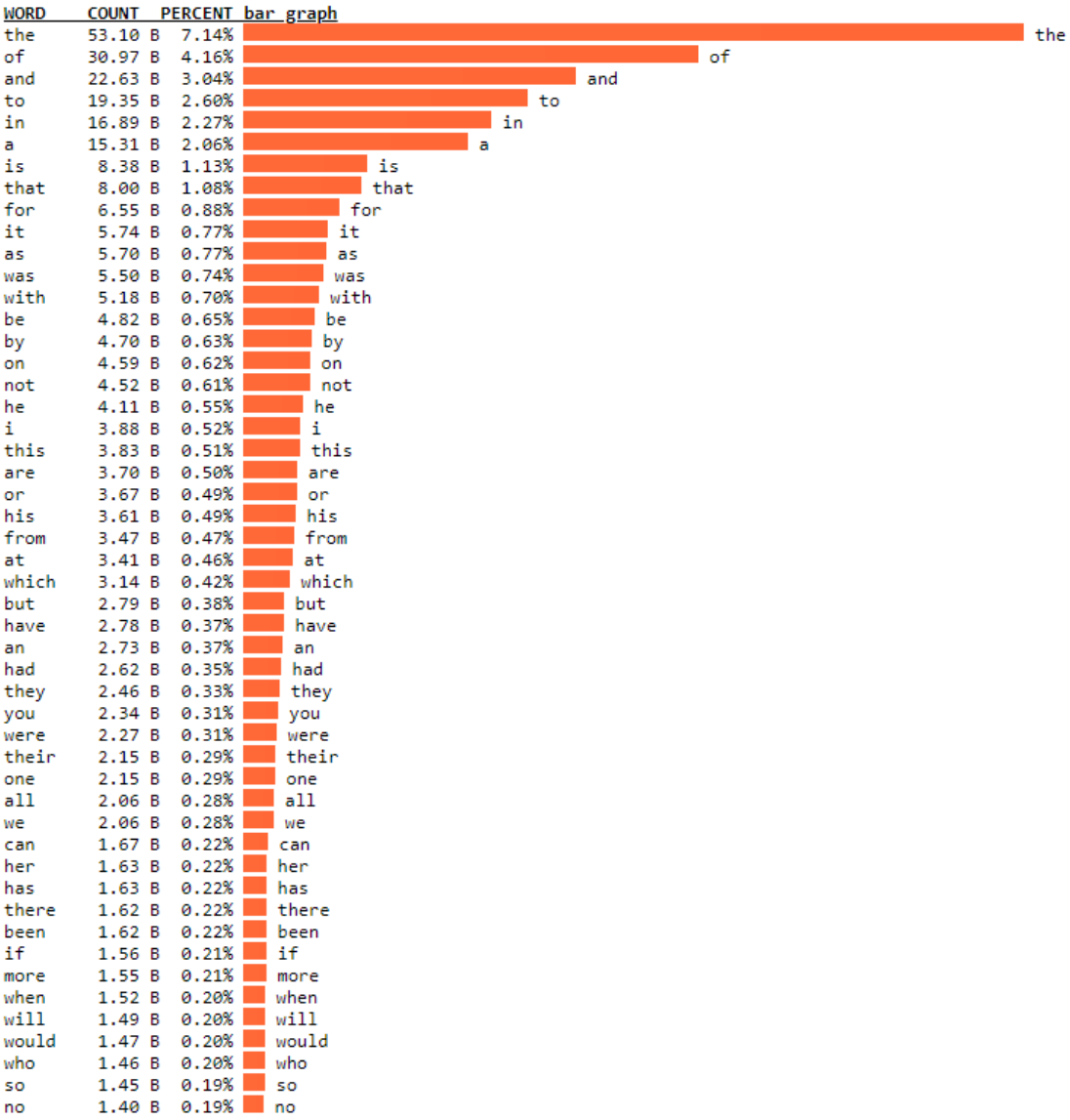


Figure: Norvig top 50 distinct words

As you can see the most common used words are not very long. This will affect the average word length. Based on these words, we will now examine the average word length.

The following graph shows the breakdown of mentions (in millions) by word length:

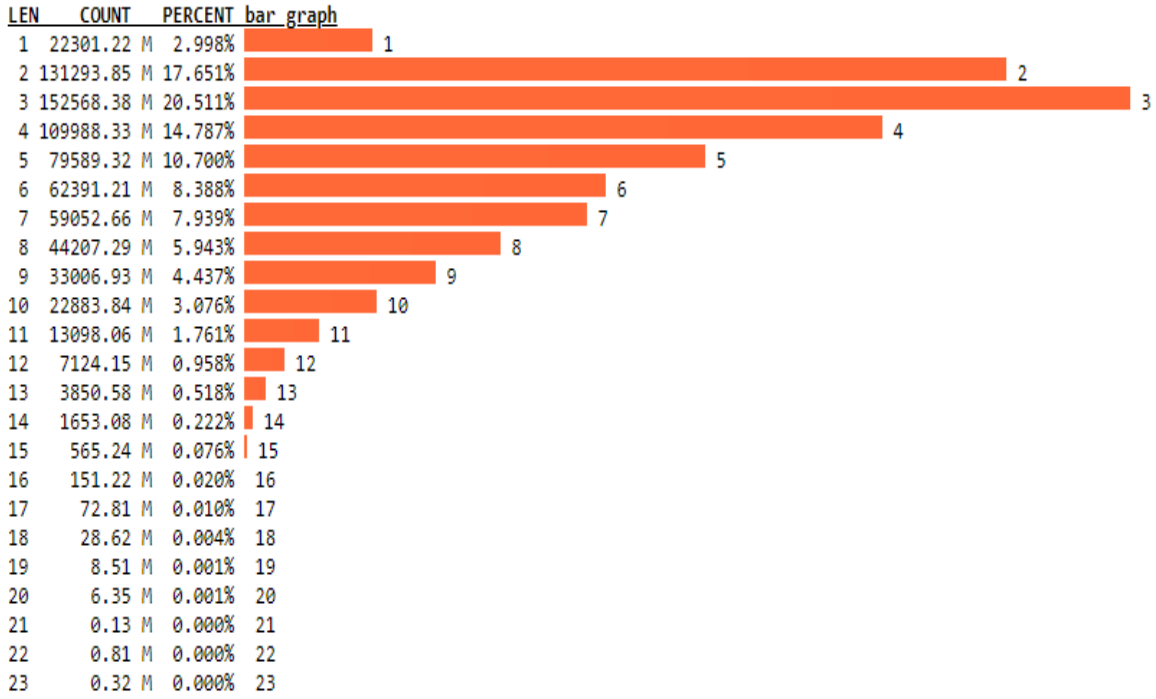


Figure: Norvig word length frequency

The average is 4.79 letters per word, and 80% are between 2 and 7 letters long. Meaning that for our data type we will need 2042 words in articles X 4.79 average characters per word = 9781.18 characters per articles. Thus, the best data based on this data is the TEXT data type with 65,535 character capacity, 64 KB in size, and 2 Bytes overhead [6,8].

Table Indexes

A database index is a data structure that improves the speed of select queries. Indexes improve the speed of queries by creating a pointer to each field in a table. The indexes can be created using one or multiple columns. The benefit for having indexes is an improvement in select queries; however, the disadvantage is that insert statements become slow because for every new row, the table must update related indexes.

There are two different types of indexes, simple and unique. Unique indexes can not have the same index value. For example, if a unique index is created for the name column of a table, then there can not be two people with the same name in that table.

This also applies to unique indexes that have more than one column. For example, if an index using a first name and last name of a table is created. Then there can not be two people with the same first and last name; however, there can be two people with the same first or last name, if one or the other is different. Unlike unique indexes, simple indexes do allow duplicate values in a table [5].

For this project, the main table that users will be querying is the articles table. Example below:

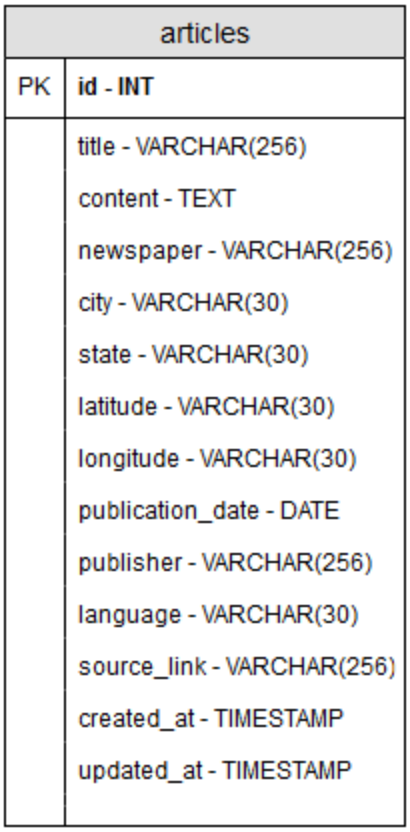


Figure: Database Article Table

This articles table contains a primary key called id. This id is also a unique index in this table, which makes sense because we want unique id number for each article in the table.

To speed up article queries, we must anticipate what a user will search for. Usually is a good idea to think in terms of sorting. When a user wants to search for something that is sorted by an attribute, then if that attribute is indexed it will speed up the search result.

For this table we can create the following simple indexes:

- newspaper
- article_tittle
- location
- page
- date

The following simple indexes can also be created because sometimes users will search by most recent addition to the database. This is different from the date of the article.

- created_at
- updated_at

Simple indexes that have more than one attribute and that will likely be useful are the following:

- location, date
- location, date, page
- article_tittle, newspaper
- article_tittle, location

Server Specifications

Requirements

The server requirements as determined by the sponsors and corresponding to the used technologies are as such:

- Platform: Linux
- Proxy Software: Apache
- Database: MySQL 5.7.Latest
- Python 3.Latest
- PHP 7.2.Latest
    - OpenSSL PHP Extension
    - PDO PHP Extension
    - Mbstring PHP Extension
    - Tokenizer PHP Extension
    - XML PHP Extension
    - Ctype PHP Extension
    - JSON PHP Extension
- Composer 1.7.Latest — https://getcomposer.org
- Node 8.12 Latest — https://nodejs.org/en/download/

Hosting

During the implementation phase of our project and with the coordination of the UCF CS department, we will use school resources to host the server on a virtual machine. The virtual machine will be based on the Linux platform and include the necessary libraries and software as specified in the requirements [see Server Requirements]. We will be given a large enough amount of storage space to contain the input OCR files from the Library of Congress archives and the correspondingly parsed data.

When the project is handed off to the project sponsors, the server will be transferred to a similarly compatible setup provided by the UCF CAH department.

API Specifications

User Operations

Register

Description:
Register the user'

Endpoint:
/api/register

HTTP Verb:
POST

Parameters:
name: the name of the person registering
email: the email that will be used to login
password: the password that will be used to login

Sample Request:
{
name: “Ricardo Pulido”,
email: “Ricardopulido21@gmail.com”,
password: “secret”
}

Sample Response:

Object:

```
{
    "user": {
        "name": "Ricardo Pulido",
        "email": "ricardopulido21@gmail.com",
        "updated_at": "2019-04-11 22:14:15",
        "created_at": "2019-04-11 22:14:15",
        "id": 4
    },
    "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9sb2NhbGhvc3Q6ODAwMFwvYXBpXC9yZWdpc3RlciIsImlhdCI6MTU1NTAyMDg1NiwiZXhwIjoxNTU1MDI0NDU2LCJuYmYiOjE1NTUwMjA4NTYsImp0aSI6InRQcnNycjZ6RkpiNnNsM
```

EUiLCJzdWIiOjQsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.qhpXc6_pnuQGcn4UlDMMMqWxjOzV5NVGwVaNk-5RYdA"
}

Sample Failed Response:
{
"{\"email\":[\"The email has already been taken.\"]}"
}

Login

Description:
Logins the user'

Endpoint:
/api/login

HTTP Verb:
POST

Parameters:
email: the email that will be used to login
password: the password that will be used to login

Sample Request:
{"email":"ricardopulido21@gmail.com",
"Password":"secret"
}

Sample Response:

Object:
{
"token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9sb2NhbGhvc3Q6ODAwMFwvYXBpXC9yZWdpc3RlciIsImlhdCI6MTU1NTAyMDg1NiwiZXhwIjoxNTU1MDI0NDU2LCJuYmYiOjE1NTUwMjA4NTYsImp0aSI6InRQcnNycjZ6RkpiNnNsMEUiLCJzdWIiOjQsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.qhpXc6_pnuQGcn4UlDMMMqWxjOzV5NVGwVaNk-5RYdA"
}

Sample Failed Response:

{
"error":"Invalid Credentials"
}

Profile

Description:
Get the user profile information

Endpoint:
/api/profile

HTTP Verb:
GET

Parameters:
token: the token that was given upon register/login

Sample Request:

{
token:eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjE6ODAwMFwvYXBpXC9sb2dpbiIsImlhdCI6MTU1NTAyMTI5NiwiZXhwIjoxNTU1MDI0ODk2LCJuYmYiOjE1NTUwMjEyOTYsImp0aSI6IkZ0Y2dSdGNaQ1pGZ2s5eTciLCJzdWIiOjMsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.A0C8XiUZhWyzkHH6SY1KBQjKV9gBgSV5fMtJJNJdHXo
}

Sample Response:

Object:

{"user":{"id":3,"name":"Ricardo Pulido","email":"ricardopulido21@gmail.com","email_verified_at":null,"created_at":"2019-04-10 21:40:23","updated_at":"2019-04-10 21:41:32"}}

Sample Failed Response:

{
"error": "token_invalid"
}

Save Queries

Description:
Saved search query to user profile.

Endpoint:
/api/saveQuery

HTTP Verb:
POST

Parameters:
token: the token that was given upon register/login
data: the search parameters

Sample Request:
{"data":{"from":"1789-01-01","to":"2019-04-11","search_terms":"berry"},"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjE6ODAwMFwvYXBpXC9sb2dpbiIsImlhdCI6MTU1NTAyMTI5NiwiZXhwIjoxNTU1MDI0ODk2LCJuYmYiOjE1NTUwMjEyOTYsImp0aSI6IkZ0Y2dSdGNaQ1pGZ2s5eTciLCJzdWIiOjMsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.A0C8XiUZhWyzkHH6SY1KBQjKV9gBgSV5fMtJJNJdHXo"}

Sample Response:

String: “query saved”

Sample Failed Response:
{
"error": "token_invalid"
}

Get User Queries

Description:
Get the user saved queries.

Endpoint:
/api/getUserQueries

HTTP Verb:

GET

Parameters:
token: the token that was given upon register/login
page: the pagination page

Sample Request:
{
token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjE6ODAwMFwvYXBpXC9sb2dpbiIsImlhdCI6MTU1NTAyMTI5NiwiZXhwIjoxNTU1MDI0ODk2LCJuYmYiOjE1NTUwMjEyOTYsImp0aSI6IkZ0Y2dSdGNaQ1pGZ2s5eTciLCJzdWIiOjMsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.A0C8XiUZhWyzkHH6SY1KBQjKV9gBgSV5fMtJJNJdHXo,
page : 1
}

Sample Response:

Object:
{"current_page":1,"data":[],"first_page_url":"http:\/\/127.0.0.1:8000\/api\/getUserQueries?page=1","from":null,"last_page":1,"last_page_url":"http:\/\/127.0.0.1:8000\/api\/getUserQueries?page=1","next_page_url":null,"path":"http:\/\/127.0.0.1:8000\/api\/getUserQueries","per_page":5,"prev_page_url":null,"to":null,"total":0}

Sample Failed Response:
{
"error": "token_invalid"
}

Delete User Queries

Description:
Delete the user saved query.

Endpoint:
/api/deleteQuery

HTTP Verb:
POST

Parameters:
token: the token that was given upon register/login

id: query id

Sample Request:
{
token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC8xMjcuMC4wLjE6ODAwMFwvYXBpXC9sb2dpbiIsImlhdCI6MTU1NTAyMTI5NiwiZXhwIjoxNTU1MDI0ODk2LCJuYmYiOjE1NTUwMjEyOTYsImp0aSI6IkZ0Y2dSdGNaQ1pGZ2s5eTciLCJzdWIiOjMsInBydiI6Ijg3ZTBhZjFlZjlmZDE1ODEyZmRlYzk3MTUzYTE0ZTBiMDQ3NTQ2YWEifQ.A0C8XiUZhWyzkHH6SY1KBQjKV9gBgSV5fMtJJNJdHXo,
id: 1
}

Sample Response:

String: “Success”

Sample Failed Response:
{
"error": "token_invalid"
}

Article Operations

Search

Description:
Provides a list of filtered and sorted articles'

Endpoint:
/api/search

HTTP Verb:
Get

Parameters:
from: the beginning date
to: the ending date
search_terms: what the user search for,
page: the start page for pagination,
per_page: how many article to return per page,

sortBy: the attribute used to sort,
sortOrder: the order of sorting

Sample Request:

{
from:1789-01-01,
to:2019-04-11,
search_terms:berry,
page:0,
per_page:10,
sortBy:publication_date,
sortOrder:asc
}

Sample Response:

response: {numFound: 2000, start: 0, maxScore: 0.00025441963,…}
docs: [{newspaper: "Heller-Howell", city: "Caribou", latitude: "21.3293", language: "te",…},…]
0: {newspaper: "Heller-Howell", city: "Caribou", latitude: "21.3293", language: "te",…}
1: {newspaper: "Auer Inc", city: "Dothan", latitude: "40.5834", language: "az",…}
2: {newspaper: "Williamson-Haag", city: "Cincinnati", latitude: "32.8137", language: "ko",…}
3: {newspaper: "O'Kon-Hahn", city: "Milwaukee", latitude: "45.5372", language: "nr",…}
4: {newspaper: "Kassulke Group", city: "Abilene", latitude: "37.6894", language: "ts",…}
5: {newspaper: "Gislason LLC", city: "New York City", latitude: "39.3051", language: "tn",…}
6: {newspaper: "Boyle LLC", city: "Buckeye", latitude: "39.7771", language: "ti",…}
7: {newspaper: "Glover, Kling and Hoeger", city: "Buckeye", latitude: "41.8229", language: "pl",…}
8: {newspaper: "Crist-Bruen", city: "Chicago", latitude: "41.7661", language: "lo",…}
9: {newspaper: "Fritsch-Fisher", city: "Savannah", latitude: "38.6358", language: "av",…}
maxScore: 0.00025441963
numFound: 2000
start: 0

Get Article Detail

Description:
Returns the specified article's data

Endpoint:
/api/details/

HTTP Verb:
Get

Parameters:
article_id: The specified article to retrieve

Sample Request:
article_id: 1
Sample Response:


```
{
   {id: 390, title: "Quod voluptas id praesentium sed dolores quia.",…}
              city: "Caribou"
              content: "Quod voluptas id praesentium sed dolores quia."
              created_at: "2019-04-09 07:30:23"
              id: 390
              language: "te"
              latitude: "21.3293"
              longitude: "-157.846"
              newspaper: "Heller-Howell"
              publication_date: "1970-01-07"
              publisher: "Marquardt-Rosenbaum"
              source_link: "https://chroniclingamerica.loc.gov/lccn/sn82014086/1917-
              08-18/ed-1/seq-13.pdf"
              state: "Maine"
              title: "Quod voluptas id praesentium sed dolores quia."
              updated_at: "2019-04-09 07:30:23"

}
```


Query Tool

Part of our project requirements is a query tool that allows boolean operators, quoted phrases, and other common query inputs. This query tool also needs to work without crashing our server or taking too long to return results.

The query tool will be searching through a database of thousands of articles. Because of this, we need to use a robust query tool that will weed out less relevant articles and will push the most relevant and useful articles to the top of the returned results. Without

a formidable query tool it is possible a query input could return every single article in the database. This would not only make the results useless, but could also crash the server.

We can learn a lot about how an efficient search tool works from Google. Without search engines like Google it would be near impossible to parse through all the information on the internet and find what you are looking for. Google is a very popular web search tool because it has a powerful query tool, and it does a good job at returning relevant websites to the user.

This project also needs a query tool that will return relevant articles quickly, therefore it may be useful to know how Google's search tool and algorithm works. Google does not share everything about how their search algorithm works, but there are certain parts of it that are known.

Google Web Crawler and Index

Google uses an automated program called a crawler [21]. Starting from a few diverse webpages the crawlers go through all the links to other webpages on each page then continues to do this from the new webpages. Once the crawler enters a new page, it is rendered then the program searches through the page looking for keywords and how those keywords are used [21]. It takes notes on what keywords are used in titles, links, URLs, and other parts of the website [21]. Using the keywords found in the page as well as other factors, the pages are put into a search index. To see an illustration of how a web crawler works look at figure C-1 below.

In our project the crawler program does not really apply, however we may be able to use indexing. The indexing could occur while each newspaper page is being parsed, and we may be able to make a note on what keywords are being used and how they are used, just like in Google's web crawling program.

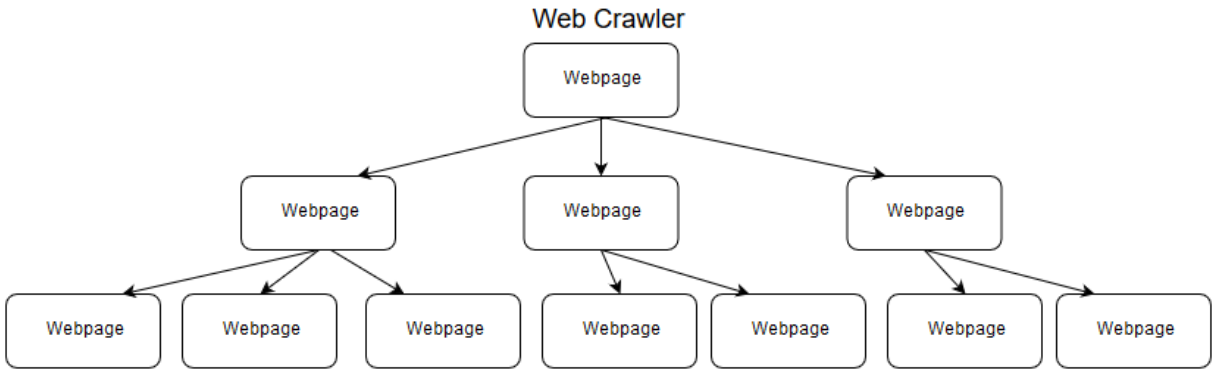


Figure C-1: Web Crawler - This illustration shows how a web crawler can spread to multiple web pages

Result Ranking

Google also ranks pages to make conclusions about the relevancy of the page to the user's query input. It determines a rank for the page using the factors:

- Counting the number of times a keyword is found in the page
- How old the page is
- How many links lead to the page

Google Search also prioritizes certain types of information, which is determined on what it perceives you are searching for. For example, if you search for "UCF Football" in Google Search it will first return information relevant to their schedule and the scores of played games as shown in the picture on the next page. You can see the result of this search below in figure C-2. Another common example is when you search the name of a restaurant, Google will try to send you the restaurant's website as the first website. It also prioritizes showing you the location of the restaurant, the hours it is open, its phone number, and the ratings of the restaurant. In our own search tool, it may be useful to prioritize certain types of data like Google does.

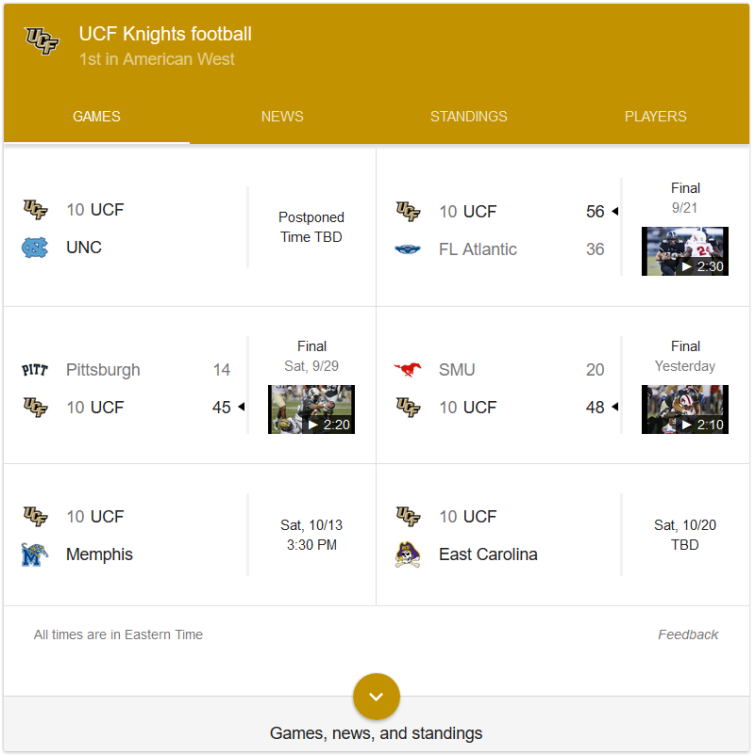


Figure C-2: UCF Knights Scores
This image shows how Google ranks this information above all else when a user searched "UCF Football"

For our own query tool we could create a ranking program in order to return the most relevant results. Our own query tool could use some inspiration from Google's page

rankings. First off, we should rank results with more uses of the keyword higher than articles that use the key word a lesser amount of times. We can also weigh uses of the keyword in the article's title more, since this means that the keyword is part of the main subject of the article. The pseudocode for this simple ranking algorithm is shown below in figure C-3.

```
for each article in database{
    rank = 0
    rank = rank + amount of times keyword used in article's main body
    rank = rank + (5 * amount of times keyword used in article's title)
    rankings[i] = rank
}
```

Figure C-3: Frequency Pseudocode
This pseudocode shows a simple ranking algorithm

The above algorithm assumes only one keyword. This is not realistic. In reality there will usually be more than one keyword entered into the query. The algorithm also assumes that every word is treated equally, and that the search tool has no syntax. This needs to be rectified.

In order to create a query tool powerful enough to return relevant results and never return all articles in the database, we need to create a syntax for inputs being searched for. Before this is done, it needs to be decided what query options are going to be included for this tool.

Boolean Operators

The first option to be included in the query tool is boolean operators. The boolean operators being used on this project's query tool will most likely be:

- AND
- OR
- NOT

OR Operator

The OR operator can be treated similarly to the simple algorithm above. The OR operator just means that there are more than one keywords listed and they are weighed the same. The more time all the keywords are listed, the more rank points the article will receive.

AND Operator

The AND operator is slightly more complicated than the OR operator in this situation. It could be treated in many different ways, but articles that have both keywords that are separated by the AND operator should be ranked higher than articles without both keywords. The pseudocode for the new ranking algorithm is shown below in figure C-4.

```
for( each article in database ) {
    rank = 0
    for( each keyword ){
        rank = rank + amount of times keyword used in article's main body
        rank = rank + (5 * amount of times keyword used in article's title)
        if( there is an AND operator between keyword and previous keyword || article contains both keywords )
            rank = rank * 2
    }
    rankings[i] = rank
}
```

Figure C-4: Advanced Ranking Pseudocode
This code shows how a more advanced ranking algorithm may work

In the example of the algorithm above, when an article has both keywords that are separated by the AND operator, the rank is multiplied by two. This gives a much higher ranking score to articles with both keywords.

Weighing the ranks higher of articles with both keywords is not the only way the AND operator could be implemented. Another way to use the AND operator is to only return results with both keywords. Therefore if the article does not have both keywords separated by the AND operator, that article will not be included in the returned articles. This would result in much fewer articles being returned from the search, and could be an effective way of eliminating irrelevant articles. The downside of this would be that it is possible it would completely eliminate articles that are still relevant to the user's query.

Not Operator

The Not operator can be easily implemented. There are two ways it can be implemented. The first way would be to deduct points from an article's search rank each time an unwanted keyword is found in the article. This would ensure articles that include the unwanted keywords would be marked as less relevant in the results, but it is still possible they could show up in the search results.

The other way to implement the Not operator is to eliminate any article that contains an unwanted keyword from being included in the returned articles of a query. This would certainly lead to less articles being returned, but it could become too extreme.

Direct Quotes

Users will want to be able to search for quotes in articles. It is a common query option and should certainly be included in this project. This option will be broken down into two parts. The first is direct quotes and the second is a quote with wildcard operators.

A direct quote is simply stating the exact quote this is to be looked for. This option is a great way to narrow down results if used correctly. If used incorrectly, it could crash the server by returning too many results.

For direct quotes a minimum amount of words should be specified. If there are not enough words in a quote, it will not be specific enough to be useful, and it will return too

many results. Our server may not be able to handle this. For example, if a user used the direct quote option to search only “and if” this could be disastrous. In this case, the direct quote is not long enough, nor is there enough unique keywords to narrow down search results.

In order to determine a reasonable minimum word count for direct quotes some testing may need to be utilized once a basic query tool is made and the website’s database is populated with articles.

Wildcard Searches

Wildcard searches are also an everyday option that people can use in many search tools around the internet. They help users search using parts of words and will use any search term matching that the pattern of the word. There are many reasons why a user might want to do this.

The implementation of this for the project needs to be carefully handled. Wildcard searches can create an excessively broad query that would return too many results, crashing the server. In order to resolve this dilemma there may need to be a minimum word or character limit enforced on the query. This would reduce the number of search results returned, while making the query more distinct.

Metadata Query Options

The metadata of the articles refers to information about the article itself. Metadata query options will help narrow down search results by allowing the user to specify information about the origins of the data they desire. This can be an efficient tool for researchers who wish to investigate only distinct areas of time and geography. The origin of data is important. It can help a researcher obtain insight into a specific perspective or simplify the collection of historical data.

Search by State and City

The first of the metadata fields that queries can be limited to is the State and city the article was written and published in. Using this option would potentially reduce the amount of returned results by a large amount. It would also allow the user to taper down the search results to the most useful if their research is in a specific geographical location.

Search by Newspaper

The second of the metadata fields that can be used to specify results is the name of the newspaper. This would limit the results returned to only articles from a single newspaper. This option would narrow down results even more than the state field, and could potentially allow a user to research a single town’s history if that is what they desire.

Search Using a Range of Dates

Another powerful query field that can be specified is the date range. This means that the user would specify a beginning date and ending date, and the query tool would search articles that were written between these dates. Again, this would reduce the amount of results returned, but possibly to a lesser degree than the state and newspaper name fields. In historical research the date that an article is written is extremely important, so it is imperative that users are allowed to specify this field.

Proximity Matching

Proximity matching is another useful and common query option. It helps to narrow the search to two words and how they relate to each other. If you simply search using two keywords it will return a bunch of results that have the two words in them. The two words could be in separate paragraphs or right next to each other in the article, it is not known.

While it cannot be used to determine the relationship between the two keywords, proximity matching can determine that a relationship exists. If the two words are relatively close to each other, it can be assumed that they are most likely used in a similar context.

Proximity matching is a fairly simple concept. The user enters two words and an integer. The search tool will search the documents for each word and if they are within the specified number from each other.

For example, a user wants to search for results that have the words “pelicans” and “eat” within 4 words of each other. If a document has the phrase “pelicans eat fish” within it, that document would be included in the returned search results. In this example the words “pelicans” and “eat” do have a relationship to one another. Using the proximity matching search option, the user may be searching for what food pelicans eat. In this example the user can see that pelicans eat fish.

One other option to complete this would be to run a direct quote, however it would not be as effective as a proximity search. A direct quote may not be useful for answering the user’s intended search in the example given above because the user is searching for what pelicans eat, and they will not be able to predict the sentence structure that the answer will be given in.

Using a search tool to find results with two words that have a relationship to one another is a crucial part for a search tool. It should be especially important in this project’s context since it is meant to be used as a tool for research. The researchers using the resultant web app from this project will be expecting to be able to use proximity matching, therefore it should be included.

Apache Solr

Solr is a popular search platform used by some of the most popular websites on the internet. Some of the websites and applications that use Solr are Netflix, Instagram,

DuckDuckGo, and eBay [17]. First created in 2004, It is a powerful tool that will be able to fulfill all the needs of the query tool for this project. It is also open sourced, so no license is needed to use this platform for this project [17]. It was created using Lucene, which is a java-based library that is used primarily for building search tools [19].

One of the reasons that Solr is so popular is that it has been proven to be able to handle high traffic volumes and has many useful features [17]. Some of these features would be ideal for this project including inverted indexing and a plethora of potential search options.

How Solr Works

Solr is one of the most popular search platforms. By examining how Solr works it can be understood how Solr works, and how it will be utilized in the context of this project. First, we will look at each piece of the Solr platform and identify what they do. Below each block of Solr is listed along with their function [20]:

- Request Handler - The request handlers handle query requests and index update requests sent to Solr. There are different handlers for different requests [20].
- Search Component - A search component is a type of request search handler. It is a type of search handled by Solr [20].
- Query Parser - A query parser parses through the text of the queries that are passed to Solr and translates them to Lucene, all while checking for syntactical errors [20].
- Response Writer - The response writer formats and generates the output for each query. There are many formats that can be used in Solr, including XML, JSON, and CSV [20].
- Analyzer/Tokenizer - The analyzer and tokenizer divides the text content and turns it into tokens that Lucene can understand. The analyzer generates a token stream from the text and passes that token stream to the tokenizer where it is turned into tokens [20].
- Update Request Processor - The update request processor is made up of signature, logging, and indexing plugins. It is triggered when an update request is sent to Solr and will add and delete fields from the index when needed [20].

Using all the above listed components, Solr can build and run an entire search engine. It can take in data sets and create an index in order to access that data faster. It will take in queries, parse through the text, interpret them, and then return the results ranked in a logical and consistent order. Over the next few sections, it will be explained how Solr will be used in this project, and why it is the best choice to implement this project's query tool.

Inverted Indexing

Solr uses inverted indexing in order to make searching large databases much faster [18]. An inverted index is a data structure that includes a mapping of a database's contents and where those contents are located within it [22].

In the context of this project an inverted index would have a list of each unique word within the article and then would note each article where that word was used. If an inverted index is used in this way a query would no longer need to search the text of every article in the database, and instead would be able to only search by the keywords inputted by the user. This results in a faster and more efficient way of searching a large database with a substantial amount of text in each record.

This is particularly beneficial to this project because of the potential for an extensive amount of text in each newspaper article that will be within the database. A small scale example of an inverted index can be seen directly below in figure C-5.

Word	Articles
Maize	5
Wheat	3,7,16,29
Boat	20,89
Fish	9
California	7,9,72
Congress	1,8,33,51,66,84

Figure C-5: Inverted Index - This shows a small example of the structure of an inverted index. The left column is the words, and the right column is a list of the articles each word appears in.

Results Ranking

Once a search is performed and all the relevant articles are gathered, Solr will then rank the results in a way similar to Google's and many other search tools' ranking systems. Solr uses an organized set of considerations to decide how to rank the relevancy of each document.

First, it uses the frequency of each inputted keyword to determine ranking, which is a simple and commonly used method [18]. It will also use inverse document frequency. This means that the less a unique word is used across the database, the more that word can potentially increase an article's relevancy ranking [18]. On top of these, a coordination factor is considered. This will take into account how many different search terms are found in an article, causing the articles with more keywords to rise in rankings [18]. Lastly, Solr penalizes articles that contain more words than average, meaning that the probability that the article will include a search term is more than average. This is called considering the field norm [18].

These are the basic variables Solr takes into account when ranking search results. It becomes more complicated when other search options are added to the query. For example, search term boosting can be used. This search option as well as others will be considered as well when Solr is ranking the returned results.

Basic Search Options

The Solr search options are plentiful and diverse. For this project there are some basic search options that are needed, and would be included by default on any well developed search tool. The Solr search platform provides all these search options including:

- Boolean Operators - AND, OR, NOT
- Search by Phrase/Direct Quote
- Proximity Matching
- Searching by Metadata
- Wildcard Searches

Solr searching is also customizable, making it possible to specify which fields to use so that we can search by every metadata option that is included in the database [17].

Fuzzy Search

Solr can also run fuzzy searches. A fuzzy search will include results containing words similar to the inputted keyword. For example, if a user inputted “load” as a keyword into a fuzzy search, it would include similar words such as “loads”, “road”, and “roads” along with “load” itself.

Fuzzy searches are useful for this project because there are some errors in the XML files from reading the newspapers and converting them to text by the Chronicling America program. Some words are read and may be wrong by one or two letters. This is where a fuzzy search could help.

The use of this search feature should be limited since it will make a query more vague, possibly leading to an excessive amount of returned results. It would make sense to only allow the fuzzy search option if the user has input enough keywords or used a number of other search options to narrow down the query. Again, if too many search results are returned it could cause problems with the server.

Search Term Boosting

When multiple search terms are used, each are usually weighted with the same importance. If a user wanted to prioritize a certain word over others, they are allowed to indicate which terms they would like to weigh more, and then how much more they would like to be prioritized over the other words. This is called search term boosting [18].

An example of this would be a user is inputs “yarn cat dog” and indicates to boost the word “cat” by 1.5. This means that the results containing the search term “cat” will be

boosted by a factor of 1.5. Additionally, some search tools have negative boosting, where search terms' importance can be lessened. However, Solr does not have negative boosting.

Possible Solr Drawback

Solr is a great platform that can help accelerate query times and development time. The only drawback to using Solr for this project would be how much server space the inverted index will take up. It is impossible to know how big the inverted index will be until it is made.

It is important to say definitively that the inverted index can not be as large as the database but may still be a considerable size. The more articles that are added to the database, the larger the inverted index will get.

The inverted index will not grow proportionally to the database however. While more articles are added, there will be less and less words added to the index for each new article because of the increasing probability that each word in new articles will have already been entered into the inverted index. The index could be large, but the more articles being added to the database, the more marginally beneficial the inverted index that Solr creates becomes to the project.

Will using Solr be worth the space it will take up? It is vital to remember that the articles within the database will often contain a hefty amount of words and would take a lengthy amount of time to search through without an inverted index. This could be mitigated by searching by using metadata first; however, the sheer quantity of articles that are expected to be in the database will also make searching the metadata without an inverted index time-costly as well.

In the end, the heightened search speed and performance will most likely be worth the sacrifice of server space that will be needed. If the search engine is not fast, users will not want to use the site or will have a low opinion on the quality of the site.

Chronicling America Search Tool

In order to build the query tool for this project, the current Chronicling America query tool needs to be analyzed. This is important because the search tool for this project needs to be at least as good as Chronicling America's tool, so when users migrate from Chronicling America's site to this project's site, they will not become frustrated with a worse query tool.

At the very least this project's query tool needs to match the power and efficiency of Chronicling America's query tool. Ideally it will be improved upon so that this project will impress users of both sites.

Also, we can learn from analyzation of Chronicling America's query tool. From looking at their tool, it can be learned what absolutely needs to be included in this project's

query tool. It can also be found out what tools and options may be less useful and can be left out.

Chronicling America: Basic Search vs. Advanced Search

When a user searches the articles of Chronicling America, they decide to use either a basic search or an advanced search [16]. When using a basic search the user forgoes to use some of the site's more powerful searching options, but a basic search can effectively search for information on places, people, concepts, events, and unique passages of text [16].

Chronicling America's basic search tool can be seen below in figure C-6. The picture shows that the basic search tool allows users to choose what state they would like to search by and offers a range of years to choose from. Both of these are optional, but after these two options, a search bar is present for users to enter keywords to search by. The Chronicling America site states that, "common words such as 'and', 'not', and 'the' are ignored by the search engine [16]."

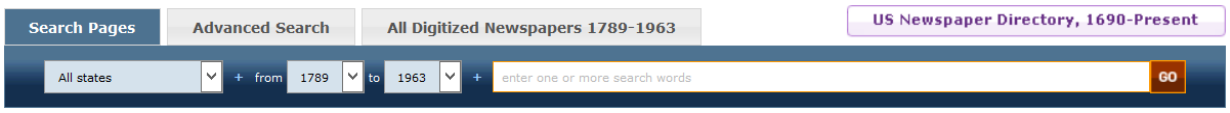


Figure C-6: Chronicling America's Basic Search Tool
This image shows Chronicling America's basic search tool. It is very simple, yet useful [16].

An advanced search has many more searching options, but these options could dilute the effectiveness of the search if a user is searching for information on the topics listed above for a basic search. Generally speaking, Chronicling America's advanced search tool allows a user to search using metadata.

The advanced search tool has separate fields to search by. The fields relating to metadata are state, newspaper name, year or year range, date range, and language [16]. The advanced search tool is better when a user needs to narrow down search results using metadata or other options. A picture of Chronicling America's advanced search tool can be seen below in figure C-7.

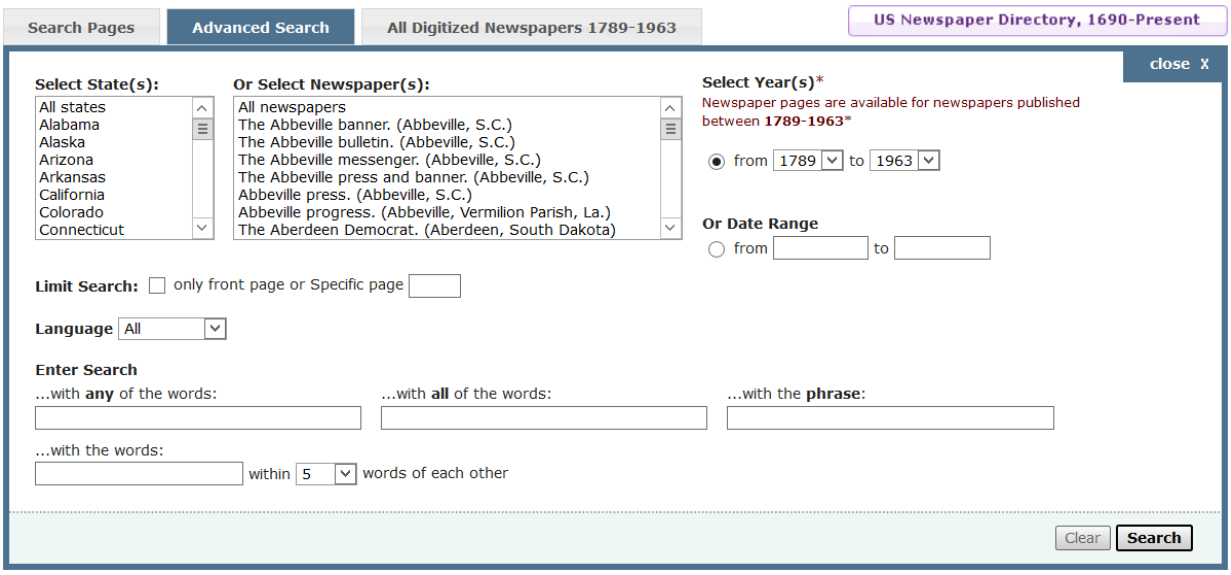


Figure C-7: Chronicling America's Advanced Search Tool
This image shows Chronicling America's advanced search tool. It is much more complicated than their basic search tool, but it is more powerful [16].

Chronicling America Search Options

Other fields included in the advanced search tool are "with any of the words," "with all of the words," and "with the phrase." Chronicling America's query tool has a few search tools and options as well. Most of these are used exclusively in the advanced search tool.

The first tool that can be noted is to search for articles "with the phrase" a user can specify [16]. This is similar to a direct quote search, however it is not the same thing. In this search option the user can enter the words in the phrase in the order they are most likely to appear. The order of the words will not disqualify any newspapers from being returned as a result, but it will change the order in which they are ranked when returned. In this way it is not exactly a direct quote search. It works similarly, but it is not the exact same. It is an important distinction to make.

A user can also search for articles "with all of the words" that a user enters. This would be similar to putting an AND boolean operator between each word.

One of the last search inputs is "with any of the words" [16]. This is similar to putting an OR boolean operator between each search word in this input.

There are a few other notable tools that are included in Chronicling America's query tool. Their advanced search tool has a field for naming words and then specifying how close they should be to each other. This is similar to Solr's proximity searching, which also allows a user to enter two words and specify how many words within each other they should be.

Chronicling America also uses a tool that allows a user to search only the front page or otherwise specify which page the user-entered search words should exist on. Being able to search by only front pages would allow the user to see only the articles matching their search terms and options that were most relevant at the time they were printed.

Also they have the previously mentioned search by date range and other forms of metadata like location and newspaper name [16]. The value of each of these tools can be discussed as it is decided on what should be included in this project's query tool.

How Chronicling America Ranks Search Results

How a query tool ranks their search results is almost as important as speed and power. Without a good ranking system, the results may be less useful or relevant to the user. Chronicling of America's advanced search tool will apply all of the search options the user chose to use, and rank accordingly.

The ranking system can become complicated when so many variables are in play. Be that as it may, Chronicling of America's basic search tool ranks the results in a much simpler and straightforward way. To rank search results, the basic search tool goes by:

- Quantity of search terms [16]
- Quantity each search term is used [16]
- Matching exactly the user's search terms [16]
- Proximity of search terms from each other [16]

Other Chronicling America Search Considerations

There are a few other details that need to be taken into consideration when discussing the query tool of Chronicling America. The first thing to consider is that Chronicling of America's query tool does not use any common words as keywords for searching [16].

This fact is comparable with this project's plan for using an inverted index to direct the search tool to articles containing key search words. The inverted index being used in this project will also not include any common words. For example the words 'the' and 'a' will not be in the inverted index, and they are also not considered in Chronicling America's search tool.

Some other facts about Chronicling America's search tool to consider is that the case of the letters for each key search word is ignored [16]. In other words, the searches are not case-sensitive.

Diacritic characters are also ignored in Chronicling America's query tool [16]. Diacritic characters are characters with accents over them. Some examples of diacritic are Ö, Ñ, and é.

Instead of using diacritic characters, their query tool will automatically substitute their respective unaccented version of each diacritic character [16]. It should be considered

for this project's query tool on whether or not to use diacritic characters in the tool. The merits of each option will be argued below.

When a user inputs a keyword into Chronicling America's query tool, their tool will also include each key search word's word variant [16]. Chronicling America explains this with an example using the word "house." When a user enters "house" as a search word, their query tool will also be searching using the words "houses" and "housing" [16]. They implement this by using a language specific dictionary to retrieve word variants [16].

What Can be Learned from Chronicling America's Query Tool

Much can be learned by examining Chronicling America's query tool, and it should help in the creation and improving of this project's own query tool. The Chronicling America search tool will be used by lots of people doing research on various subjects, and it was built like it. Overall, it is a functional and useful tool.

The first thing that can be discussed about Chronicling America's query tool is that it does not consider diacritic characters. Instead they are converted to their respective non-accented characters and then entered into the search.

On one hand allowing diacritic characters in a search tool would allow the weeding out of more articles when searches are performed. However, in some languages, especially English, accented and diacritic characters are used inconsistently. Therefore, it would probably be most useful for this project's query tool to not take into account diacritic characters. This may be more useful to use if this project decides to branch out into other languages though.

The next topic that needs to be discussed is if this project's search tool should intake words and consider their case. In other words, should this project's search tool not be case-sensitive like Chronicling America's search tool.

Similar to diacritic characters, case-sensitive searches would narrow down the amount of articles returned. Names are capitalized, so if a user enters a name and uses the correct capitalization, a case-sensitive search would not return articles that have a word similar to the name but are not capitalized. However, the articles that are not being returned due to case-sensitivity could still be relevant. For example, if words are at the beginning of sentences they are automatically capitalized. This has no bearing on the meaning of the word, only that it appears at the front of a sentence.

A case-sensitive search seems like it could be useful, however the fact that it could get rid of many relevant articles to the search due to syntax it should not be used. Maybe as a stretch goal this project could include a way for the user to choose which words they would like to search with case-sensitivity.

The third thing to contemplate about Chronicling America's query tool is that a user can search for keywords and get results in a specified language. There are many

newspapers in America and most will be English language newspapers, however there are still a good deal of newspapers written in languages other than English. It needs to be considered how this project will handle newspapers in languages other than English. It may be the case that no extra steps need to be taken to parse through non-English newspapers.

Since there are so many possible languages, and therefore possibilities for unknown variables, it would probably be best to test the whole project starting with English newspapers, and then see how it works with newspapers in other languages. If there are additional steps that need to be taken, then it should be considered an important stretch goal, or a main goal for the next group that continues this project because it should be important to include newspapers in other languages in this project.

The way that Chronicling America has divided their search tool into a basic search tool and an advanced search tool is another factor that needs to be considered for this project. Each tool has their strengths and purposes, and both are valuable tools.

For this project it needs to be considered whether or not to split this project's query tool into a basic search and an advanced search. Although it is important that this project improves upon the Chronicling America project and differentiates itself from their project, the way they split their tool like this can be admired and then proven upon by this project.

The argument against making both a basic query tool and an advanced query tool would be that it would increase development time of the query tool. The parser is the most important and difficult part of this project, so any time taken away from the development of that needs to be justified.

In this case, having an efficient and powerful query tool is imperative to the project. Furthermore, it is probably for the best that this project has both a basic and an advanced query tool, so the time spent on this would be more than justified.

Each tool has their strengths and uses, and both of the uses are needed in this project. The advanced tool can powerfully search for keywords while also using article metadata. The basic search will be able to efficiently search by just keywords.

Because the query tool is one of the main ways users will be interacting with this project, it will be time well spent to make this query tool powerful, efficient, and able to adapt to what the user wants to use. Therefore, it is concluded that this project will need to split its query tool into an advanced search tool and a basic search tool.

How does this Project's Query Tool Compare to Chronicling America's Query Tool

For this project it is important to have a better query tool than Chronicling America's query tool. For that reason it needs to be seen how this project's query tool plan stacks up to Chronicling America's existing query tool. If this project's query tool is found

lacking when compared to Chronicling America's tool, then a new plan needs to be made in order to ensure this project's quality and practicality.

Let us start by comparing the boolean operators that will be available for use in each query tool. As you can see in the table below, both Chronicling America's search tool and this project's search tool with both have ways of using the AND and OR boolean operators.

The difference between the two tools in this case is that this project's search tool will make the NOT operator available to the user, while Chronicling America's tool does not [16]. In this category, this project's search tool has the upperhand in a big way. The NOT operator is an essential tool for many search tools, including Google's search tool. It offers the opportunity in the ranking of results to penalize articles for containing specific search words.

The NOT operator is an effective tool to have because a certain combination of keywords will not be useful to the user, while a similar set of keywords would be useful. By allowing the user to exclude certain words, the chance of each article returned being relevant is greatly increased. These differences are also shown in figure C-8 below.

Boolean Operator Search Comparison

	Chronicling America Search Tool	Group 44 Project Search Tool
AND	YES	YES
OR	YES	YES
NOT	NO	YES

Figure C-8: Boolean Operator Search Comparison: This table shows the differences each search tool has in using boolean operators.

The next issue being used to compare the two query tools is searching by metadata. By viewing the chart below, it can be seen that the comparison is more equal than the boolean operator comparison.

Both query tools will be able to search by state, however only this project's query tool will be able to search by city. This may not always be possible, if the city is not included in the metadata of the article, but it usually is.

Both query tools will be able to search by the name of the newspaper that published the article, year range, and date range. The difference between the date range and year range is that the date range includes the month and day on top of the year.

The Chronicling America query tool is able to search by language, while it is yet to be seen if this project's query tool will be able to search by language, but it is a stretch goal. The major advantage that this project's query tool will have over Chronicling America's query tool is that it will be able to search by words in articles instead of

pages. Chronicling America does not have its newspapers separated into articles, while the main objective of this project is to parse through the newspapers and distinguish between articles. By separating pages into articles smaller portions of text can be analyzed for relevance by our query tool. Overall the query tools are pretty evenly matched in this category. This can be seen in figure C-9 below.

	Chronicling America Search Tool	Group 44 Project Search Tool
State	YES	YES
City	NO	SOMETIMES
Newspaper Name	YES	YES
Year Range	YES	YES
Date Range	YES	YES
Language	YES	MAYBE
Article Title	NO	YES

Figure C-9: Metadata Search Comparison - This table shows the metadata each search tool does and does not allow users to search by.

The next matter to compare these two tools on is search options. Chronicling America's search tool does not allow the user to use search term boosting, while this project's search tool will allow search term boosting. This option allows users to specify which search terms they want to prioritised over the other words.

Both of these query tools will allow users to search using proximity matching, which is when users specify how close certain search words should be next to each other. Also, both of these query tools can implement some form of direct quote search or phrase search. Lastly, only this project's query tool will let users run a wildcard search, while Chronicling America's query tool will not. Simply speaking, this allows the users to substitute a wildcard operator in place of letters for search words.

Overall, this section helps further solidify this project's search tool's superiority over Chronicling America's search tool. In this case, the project has clearly improved upon the implementation of Chronicling America's noble mission. This can all be seen in figure C-10 below.

	Chronicling America Search Tool	Group 44 Project Search Tool
Fuzzy Searching	NO	YES
Search Term Boosting	NO	YES
Proximity Matching	YES	YES

Direct Quote/Phrase	YES	YES
Wildcard Search	NO	YES

Figure C-10: Search Options Comparison - This table shows the search options each tool does and does not have.

Another difference between this project's query tool and Chronicling America's query tool is that this project's site will let you save queries. Meaning that it will let a user save the search inputs that led them to a certain set of returned articles. It will not save the articles themselves that are returned. Chronicling America's query tool does not have this feature.

To a select few of serious researchers, being able to save queries will be incredibly useful, especially if the user is making a substantial amount of queries. This pushes this project's query tool further ahead of Chronicling America's query tool, further legitimizing this project as a useful tool for researchers.

Next, we will compare how each query tool will rank returned results, and see if one system is better than the other. By looking at the table below, the way each query tool ranks results can be seen.

As a reminder coordination factor is the amount of different query terms found in a document [18]. Also, field norm penalizes documents for how long they are, since they have a higher likelihood of containing search terms. The last reminder is that inverse document frequency is that the more rare a search term is across all documents, the more it counts in the ranking score [18].

Taking all these into account, this project's search tool is more comprehensive and more likely to return relevant results. Both of the ranking systems have similarities, however this project's search tool takes into account the length of documents and how common each search term is throughout the inverted index, while Chronicling America's search tool does not.

The only upper hand that Chronicling America's search tool will have on this project's search tool is that it takes into account how close each search term is to one another. Overall deciding which ranking system is better is subjective, but this project's ranking system, provided by the Solr platform, is more sophisticated and more comprehensive since it will consider variables throughout all the documents and within each document itself. A table illustrating all of this can be seen in figure C-11.

Chronicling America Search Tool	**Group 44 Project Search Tool**
Search Term Frequency	Search Term Frequency
Exact Matches of Search Terms	Inverse Document Frequency
Coordination Factor	Coordination Factor

Search Terms Appearing Near Each Other	Field Norm
Does Not Count Common Words	Does Not Count Common Words

Figure C-11: Search Tool Ranking - This table shows how each search tool will rank their returned results.

Chronicling America's search tool is fairly fast despite the extensive amount of newspapers that it needs to be able to search through. It is important that this project's search tool is also just as fast.

To ensure that this project's search tool is quick, the inverted index that Solr will produce will be used. As a reminder, it will allow the search tool to search by words that point to which articles contain them. Whether or not this will allow this project's query tool to be as fast as Chronicling America's remains to be seen. Testing the query tool speed should begin as soon as it is set up. Until then, it can not be determined if this project's query tool will be able to match the speed of Chronicling America's query tool.

Overall, how do the two query tools compare to each other? From the above analysis, it is apparent that this project's query tool has not only matched Chronicling America's query tool, but it has improved on it in many ways. First, this project's search tool allows for NOT operators to be used, while Chronicling America's does not.

Also, this project's query tool will allow users to use much more search options that Chronicling America's does not, including fuzzy searching, search term boosting, and wildcard searches. On top of this, when a user uses this project to search for keywords, the search tool will return articles as opposed to whole newspapers. This allows the user to see a more discerning and relevant set of returned options to choose from.

The only way this project's query tool may fall short of Chronicling America's is by allowing users to search by language. This, however, may become possible in the future, but since it is not definitely planned to be implemented will continue to be a shortfall. However, since this query tool seems to offer a more comprehensive and powerful searching experience overall, the planned query tool is acceptable for this project.

Basic Search and Advanced Search

It has been determined that this project's query tool needs to be divided into two seperate tools. The first being the basic query tool, and the second being the advanced search tool. Next, it needs to be resolved what each tool will include.

The basic search will of course have the ability to search using user entered search terms. It should also include the ability to search by a range of dates. This is a simple addition, and it is one of the most useful metadata options to be able to search by so it should be included in the basic search tool. It should also be possible for users to enter boolean operators, however it may not explicitly be stated. Instructions on using

operators throughout all the search fields should be included in a “help” page on the website. A rough wireframe of the basic search tool can be seen below in figure C-12.

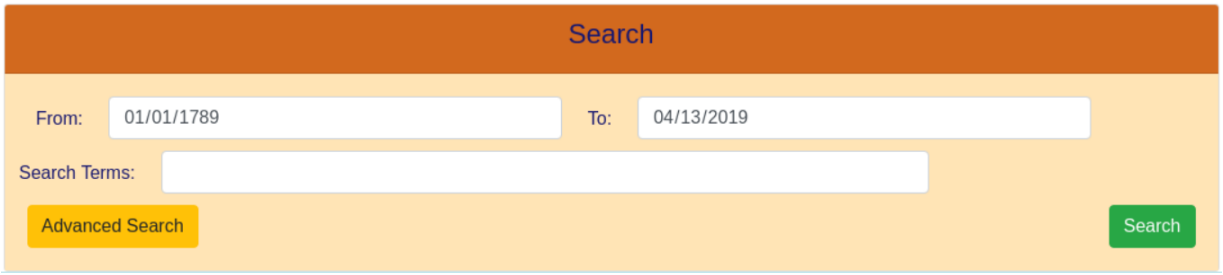


Figure C-12: Basic Search Tool
This image is a wireframe of what this project’s basic search tool may look like.

The advanced search will also include a date range to search by. On top of that it will also include the ability to search by publisher, newspaper, state, and city. This shows that users will be able to search by all the useful metadata available for each article. Additionally, there are fields for users to enter search terms. There are several fields for this function, and each field will have a different function.

First the “include any of these words” field is similar to using an OR operator to separate each search term. Next, the “include all of these words” field is similar to using an AND operator between these terms. The “do not include any of these words” field would be the equivalent of entering the NOT operator before each of these words. The “include the phrase” field would be the equivalent of using a direct quote. Last, the “include these words within X words of each other” field is the equivalent of a proximity search. In the future, it may be easier to include instructions on how to use the operators, direct quote function, and proximity search tools in a “help” page for users instead of providing explicit fields to enter each search function. An illustration of an advanced search tool meeting all these specifications can be seen below in figure C-13.

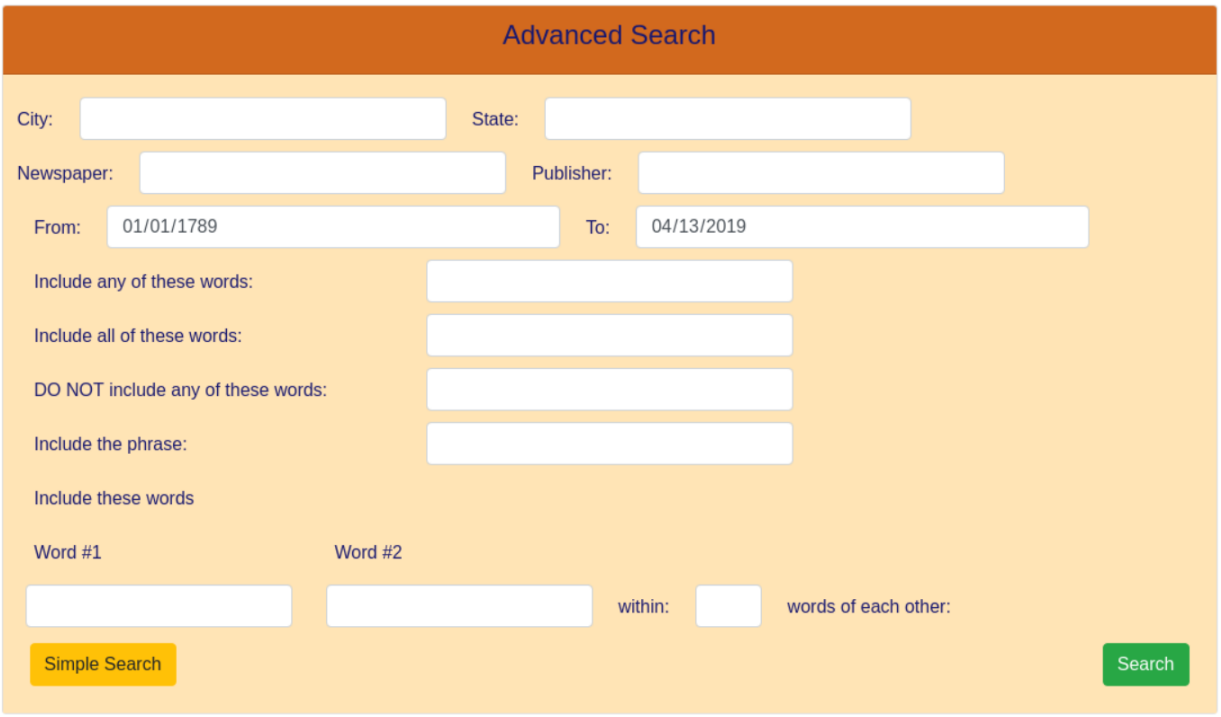


Figure C-13: Advanced Search Tool
This is a wireframe illustration of what an advanced search tool may look like in this project.

Summary of How the Query Tool will Work

Now that many critical parts of the query tool have been researched and decided on, the plan for how the query tool will actually work can be laid out. Even though Solr will be handling many functions within the query tool, the process of querying search words needs to be planned.

First, when a user begins using the query tool they will need to decide on whether they will want to use a basic search or an advanced search. Once they choose this, they will enter their search terms and criteria, including the metadata that will narrow down results.

Once the user clicks the “search” button their query will be lightly translated into a form Solr understands by adding fields and necessary operators. Once it is translated, it is sent to Solr to parse through the query text.

Next Solr will search for the search terms in the inverted index. The inverted index will tell Solr which articles in the database contain the search terms. Once Solr retrieves the full article content and its information, it will analyze them in order to rank the articles by relevance using Solr’s own ranking system and the criteria the user entered. Once the articles have been ranked, their information will be returned to the user allowing them to choose which article they wish to look through. An illustration of this process is shown below in figure C-14.

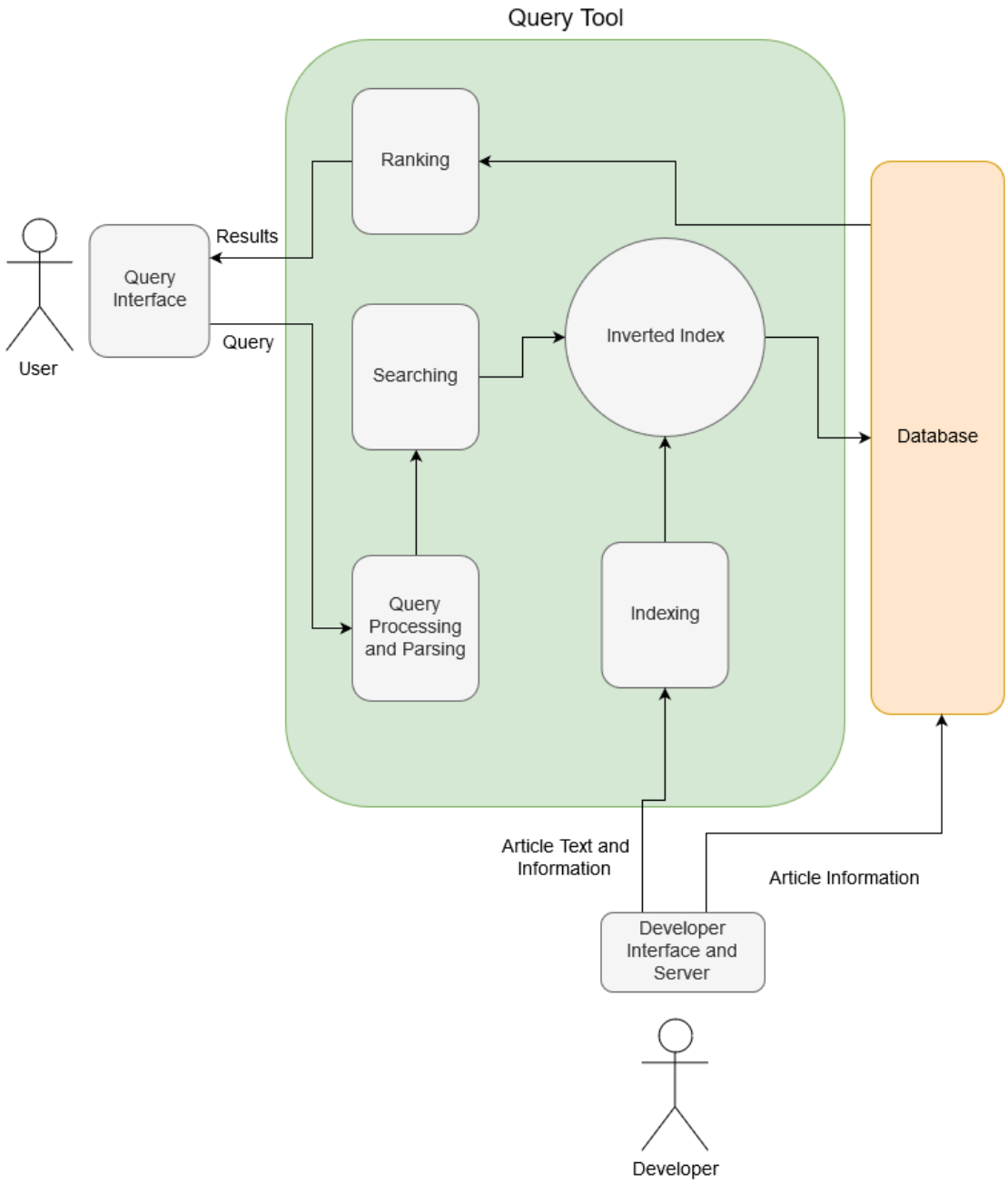


Figure C-14: Query Tool Process - This illustration shows the process of a user entering a query and the query tool processing that procedure. It also shows what happens when developers will be entering articles into the database and inverted index.

Final Thoughts on the Query Tool

For a project like this the query tool is essential. It will be one of the main points the users will be interacting with the site and is a big bottleneck to users trying to achieve

their goals while using the site. In some cases the usefulness and efficiency of the query tool will be how some users define their experience while using the project.

With that said, the query tool that has been planned out in this document will be an efficient and powerful tool that is designed to cater toward serious researchers. Furthermore, users who have experience using Chronicling America's search tool will not be frustrated by having less search options while using this project. On the contrary, they should be impressed by how this project's search tool adds to the search options they are accustomed to.

Technologies and Dependencies

For this project we will be using the following dependencies to power the website. These dependencies are stored in files on the project root and are installed via package managers, NPM for front-end/client libraries, and Composer for back-end/server libraries.

There are also development dependencies used to improve development of the website through developer tools such as testing, formatting, and code inspection.

Required

Package	Version	Description	Link
NPM			
react	16.6.Latest	React is a JavaScript library for creating user interfaces	https://www.npmjs.com/package/react
bootstrap	4.1.Latest	Sleek, intuitive, and powerful front-end framework for faster and easier web development	https://www.npmjs.com/package/bootstrap
moment	2.22.Latest	A Lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates	https://www.npmjs.com/package/moment
jquery	3.3.1	jQuery is a fast, small and feature-rich JavaScript library	https://www.npmjs.com/package/jquery
formik	1.3.Latest	Formik brings improved forms and validation logic in React	https://www.npmjs.com/package/formik
heatmap.js	2.0.5	Dynamic JavaScript Heatmaps for the web	https://www.npmjs.com/package/heatmap.js
Composer			
laravel/ framework	5.7.Latest	The core framework powering Laravel	https://www.npmjs.com/package/@babel/core

php	7.2.Latest	PHP is a popular general-purpose scripting language that is especially suited to web development	https://secure.php.net

Development Dependencies

Package	Version	Description	Link
NPM			
@babel/core	7.1.2	Babel is a tool that compiles and transforms JavaScript code to support browsers	https://www.npmjs.com/package/react
webpack	4.23.1	Webpack is a module bundler for JavaScript files and resources/assets	https://www.npmjs.com/package/webpack
Composer			
phpunit	7.latest	PHPUnit is a programmer-oriented testing framework for PHP	https://packagist.org/packages/phpunit/phpunit

Use Case Diagram

Below is our use case diagram. The two actors being displayed are a regular user and a developer. The developer is mainly only going to be adding newspaper files from the Library of Congress' *Chronicling America* program. Once added, they'll be parsed and added to the database along with their metadata. The use case diagram can be seen below in figure U-1.

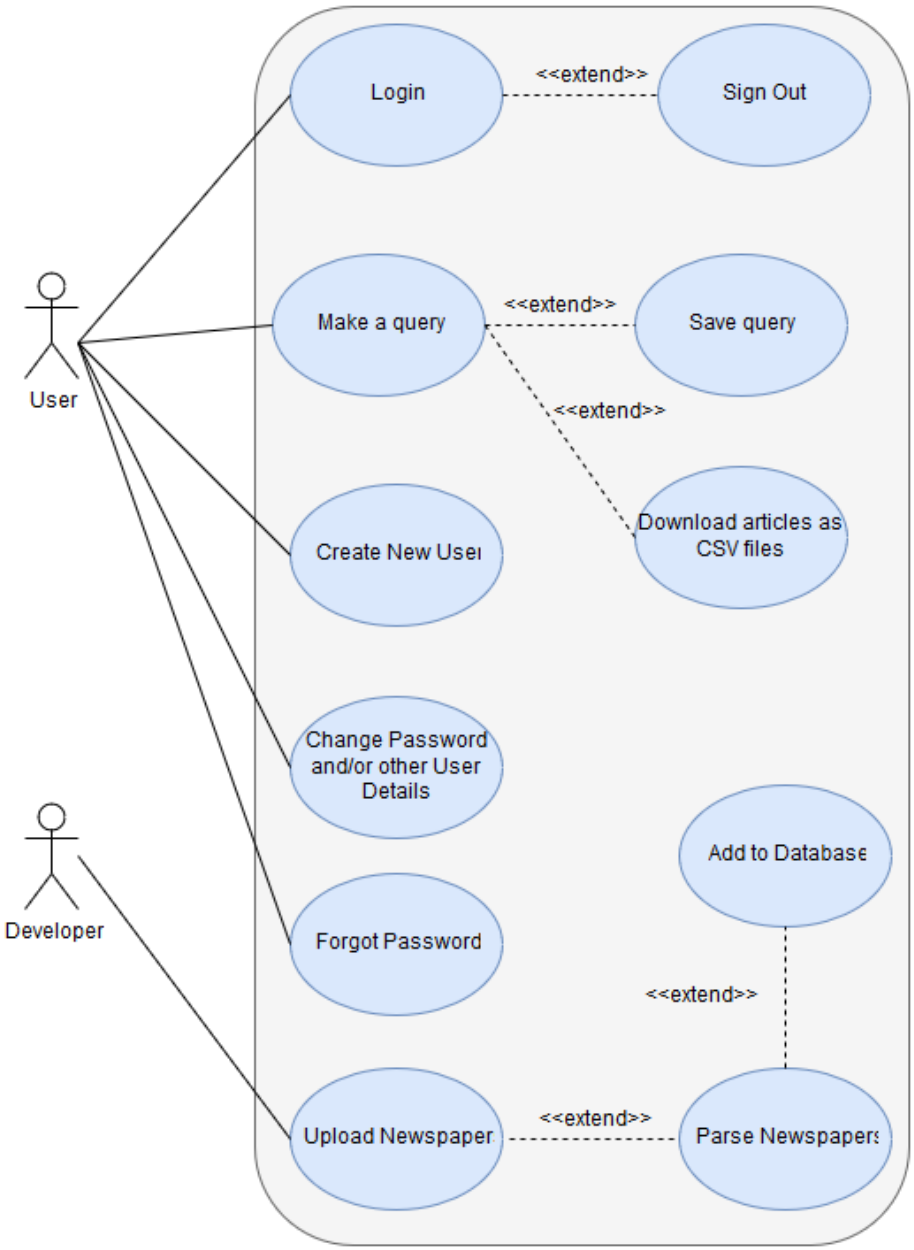
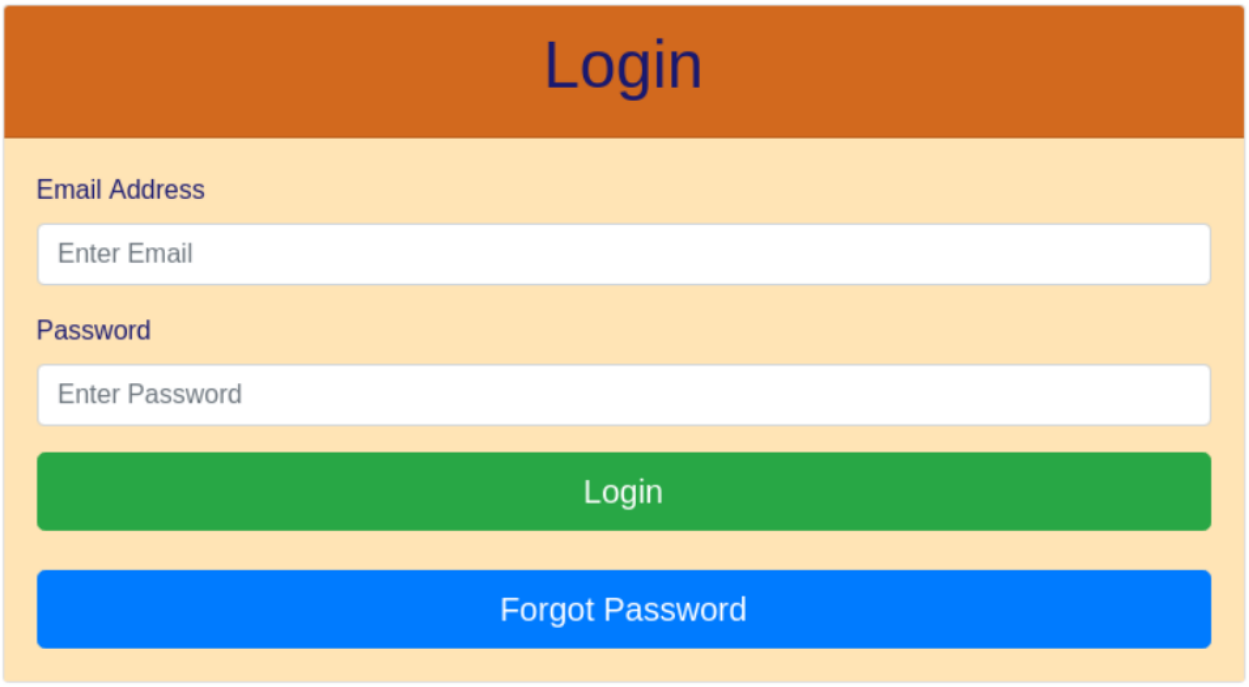


Figure U-1: Use Case Diagram
This is the illustration for this project's use case diagram

Wireframes

Sign up

Login



Search Results

Title: GARLAND'S EXONERATION'
Publication Date: 07/01/1886
Details

Article:PIB. BOYLE SUBMITS THS BE- POItT TO THE HOUSE. An Exlianattre Explanation of the Utter hack ot Foundation for the Eartiaan Insinuation of Malfeasance—Indlspu- tahle Evidence That There Wai No Intention to Use Official Power for Private Gain. Washington, .June 30.—1n the House to-day Mr. Hoyle, of Pe...

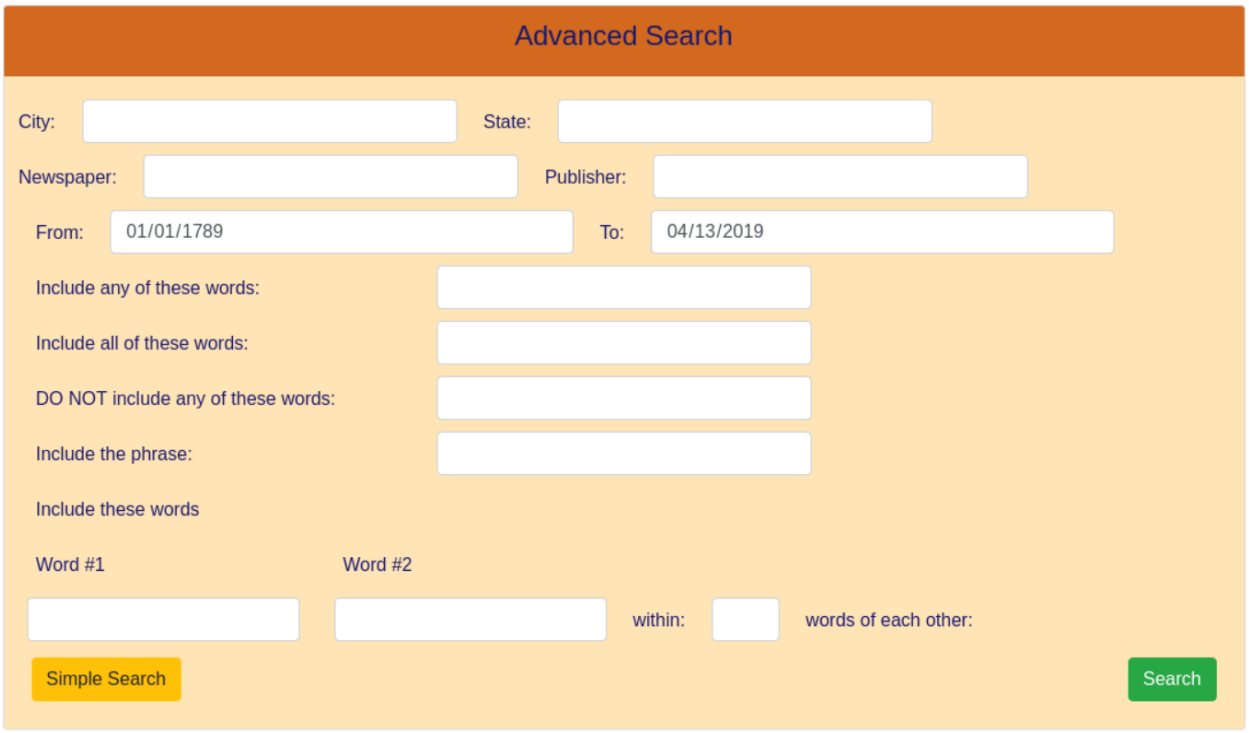
Title: THE memthis suit.
Publication Date: 07/01/1886
Details

Article: Touching the lirst government suit at Memphis, the report devotes some space to reviews of the testimony on that point m) *t-d 'ht the Washington meeting of the National Improved and Pan Electric people was not proposed with a view to the instiiution oi the government suit. The history of the procee...

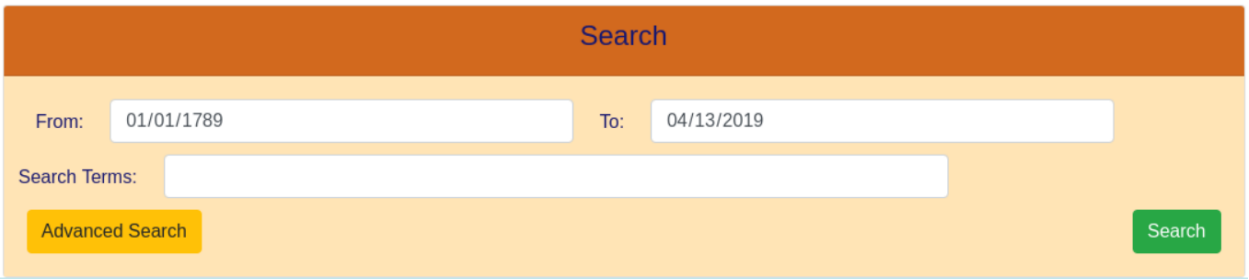
Title: LEGISLATIVE EXPENSES.
Publication Date: 07/01/1886
Details

Article: The Senate Takes tlie House to Task for Its A lk-<red Tardiness. Washington, June 30.—The Senate to-day proceeded to the consideration ot the legislative appropriation bill. In the course of a discussion on amendments in- creasing the clerical force ot some of the departments, Mr. Beck declared that...

Advanced Search Tool



Basic Search Tool



Parser Algorithm

Metadata

The first thing that our algorithm needs to do is build the URL that belongs to the file that we are parsing. This is done by using the xml file directory to build the URL. Below is an example of an xml file location.

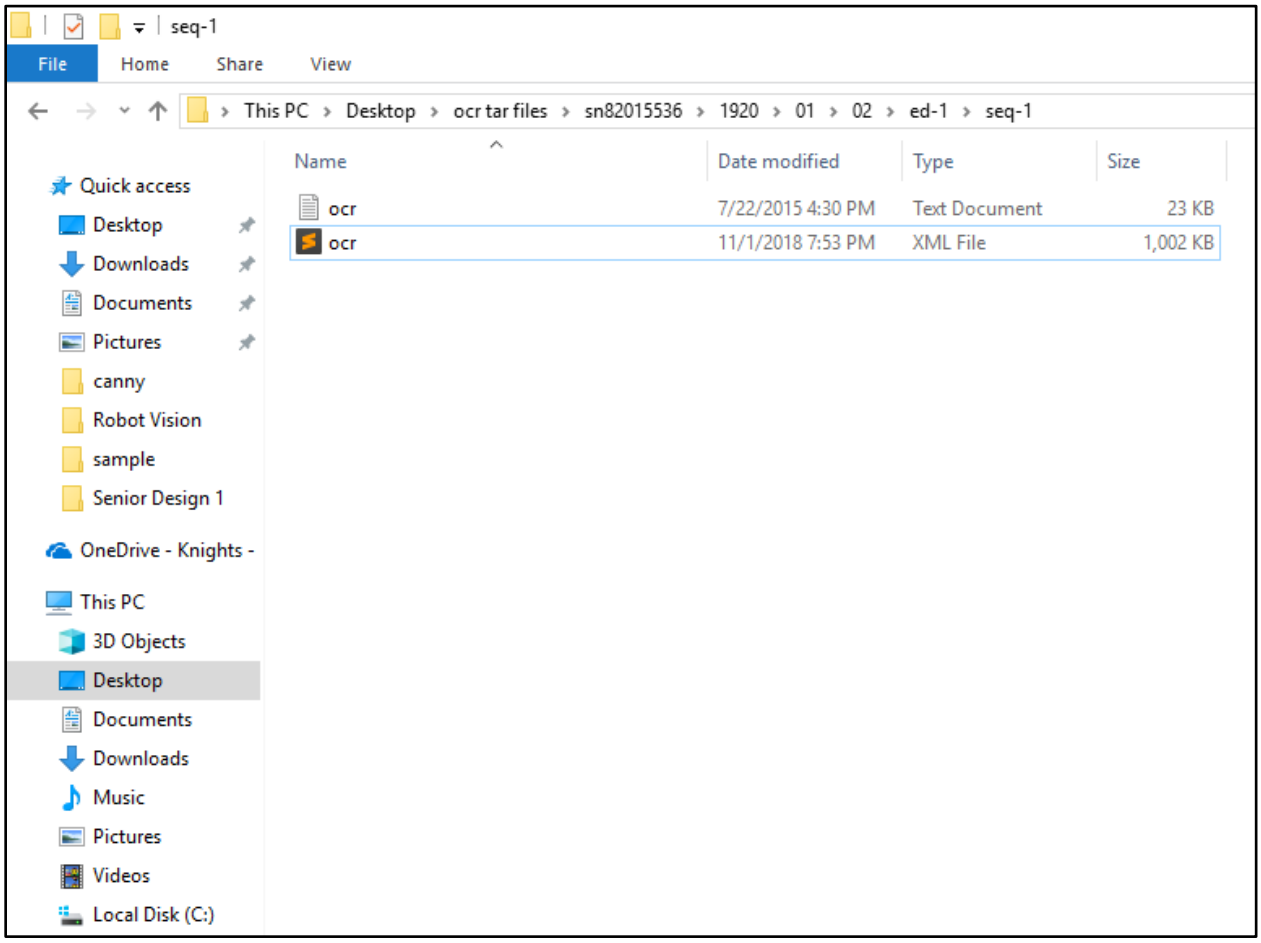


Figure: XML file directory example

Using the above file directory, the parser needs to construct the following newspaper URL:

https://chroniclingamerica.loc.gov/lccn/sn82015536/1920-01-02/ed-1/seq-1

This URL contains 5 parts:

1. https://chroniclingamerica.loc.gov/lccn/	This is how all newspaper URL will start
2. sn82015536	This is the root folder of all xml files
3. 1920-01-02	3 folders after root
4. ed-1	5th folder
5. seq-1	6th and final folder

After the URL for the corresponding XML file has been created the algorithm needs to get all the necessary metadata for that xml file by passing in parts of the URL as an argument to the API. For example, using part of the previous URL and adding a .json extension, https://chroniclingamerica.loc.gov/lccn/sn82015536.json, we get access to the following data:

The place of publication, the name of the newspaper, and the publisher name. As for the date of publication we already have that from when we created the URL, in this example it was 1920-01-02. All of this information can now be saved for each article that is parsed.

Example of the json data below:

Figure: Url json file example

Parsing Articles

The next step is to parse the xml file. The following algorithm will be used:

1) As a String element is scanned, store the content as a potential article title, while keeping track of each String element height

2) If the height change decreases and the magnitude of that change is greater than some threshold, then store the previous string as the title and go to step 3. Otherwise, if the height increases keep scanning and storing the String elements content as part of the title.

3) Keep scanning all String element, whatever comes next is the article data. If the height decreases, keep scanning and saving the String element content as part of the article data. Otherwise, If the height increases and the change is greater than some threshold, then store the previous saved strings as the article data.

4) If case 1 pass, then store article title, data, metadata in database. Otherwise, discard article title and data and go to next step.

5) Repeat step 1 through 4 until reach the end of the file.

The above algorithm is very basic and will function incorrectly unless we account for special cases. The following are some of the special cases that must be accounted for:

Case 1: Not enough words in article data. This is due to rapidly alternating height because of front page headers or junk data like advertisements. Assuming we choose 30 as our threshold, then our algorithm will choose the words in the red box as the title, and the words in the blue box as the article data for that title. Thus, in order to avoid storing junk data. We can check the length of the article data, and if that length is long enough, then we can accept the article. Note that the optimal threshold will be decided through trial and error. Example below:

Case 1: XML Representation

Color red is article title. Color blue is article data.

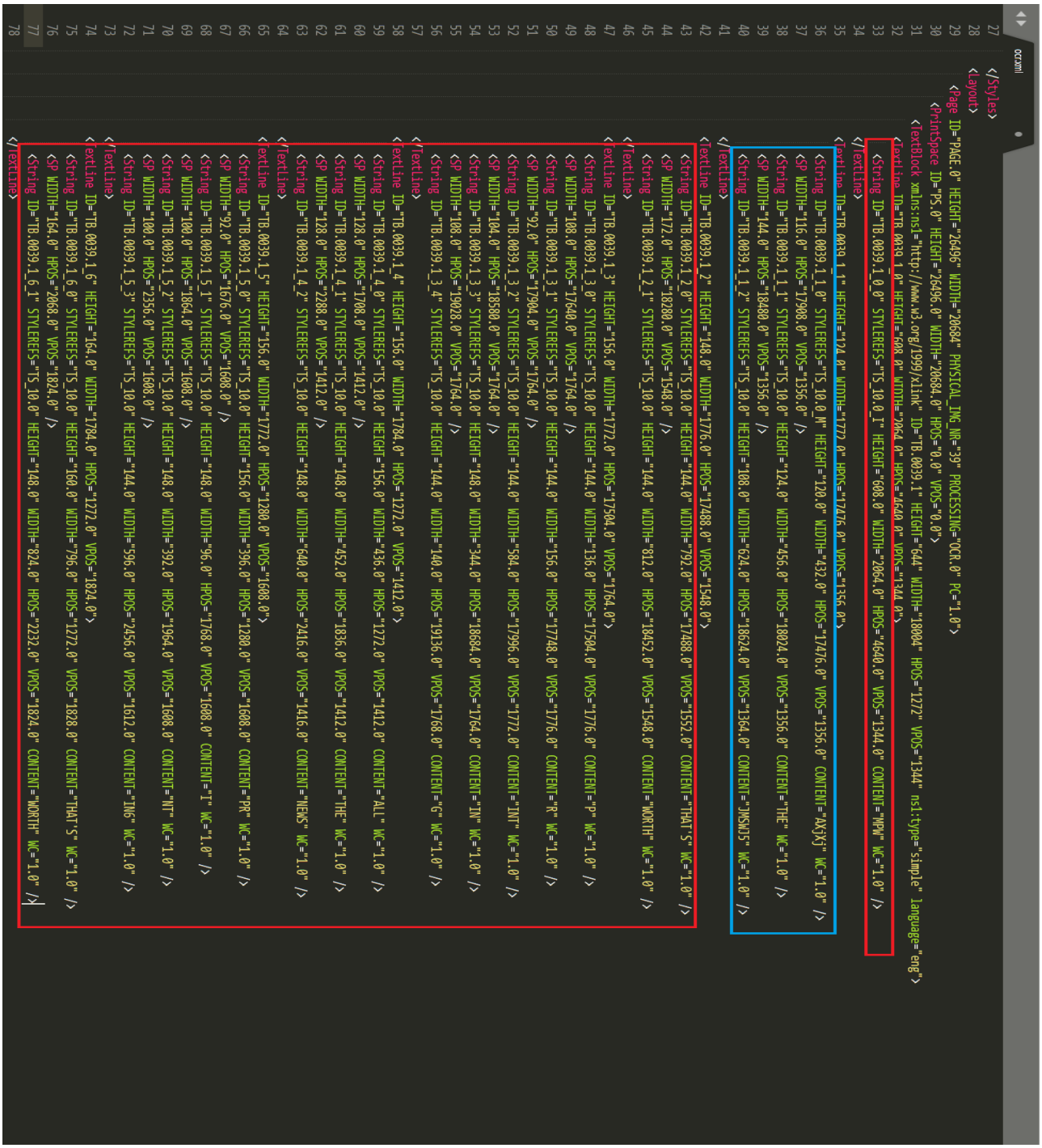


Figure: Case 1 XML Representation

Color red is article title. Color blue is article data.

Figure: Case 1 XML Representation 2

Case 1: Newspaper Representation

Color red is article title. Color blue is article data.


ALL THE NEWS THAT'S WORTH PRINTING

REPUBLICAN FARMER.

ALL THE NEWS THAT'S WORTH PRINTING

Established A.D. 1790—Vol. CXXVII

BRIDGEPORT, CONN. FRIDAY, JANUARY 2, 1920

OFFICIALS AWAIT NEW WAVE OF DEATH TOLLS

Famous New Haven Restaurant Totally Destroyed By Flames

1920 "OPEN SEASON" FOR ALL BOLSHEVISTS

THIRTY-FIVE UNDER ARREST WHO SOLD POISONED WHISKEY

Death Toll in State Was 17—Increased By But One Since Saturday—Authorities Awaiting Result of New Year's Celebrations in Country.

ASKS HUSBAND BE ADJUDGED IN CONTEMPT

Mrs. Frawley Asserts He Has Failed to Pay Alimony

TEST FOR WOOD ALCOHOL WITH A HOT COPPER WIRE

GET SUMMONS FOR RETURN OF MISS SUMMIT

Girl Whose Story Caused Implication of Roommate.

BAD FIRE WRECKS CAFE

Manager Held By Police in Wood Alcohol Cases.

ALCOHOL FUMES WERE NOTICEABLE

SOVIETS TO BE IN AUTHORITY ALL OVER WORLD

NEW YEAR'S DAY QUIETLY OBSERVED AT WASHINGTON

No Formal Celebration At White House—Reception Held.

POLICY TOWARD REDS WILL BE UNFLINCHING, AGGRESSIVE WARFARE

"GENE" M'CANN DEFENDANT IN DIVORCE SUIT

Desertion Is Ground—He Will Not Contest Action

Figure: Case 1 Newspaper Representation

Case 2: In this case the first letter of a sentence is larger than the rest of the words that sentence is comprised of.

The algorithm will check if the string content is the first element of a TextLine, and if it is true, then the height condition will be waived for that one string content. Otherwise, the height condition will apply. This will change the algorithm to the following:

1) As a String element is scanned, store the content as a potential article title, while keeping track of each String element height

2) If the height change decreases and the magnitude of that change is greater than some threshold, then store the previous string as the title and go to step 3. Otherwise, if the height increases keep scanning and storing the String elements content as part of the title.

3) Keep scanning all String element, whatever comes next is the article data. If the height decreases, keep scanning and saving the String element content as part of the article data. Otherwise, If the height increases and the change is greater than some threshold and case 2 passes, then store the previous saved strings as the article data.

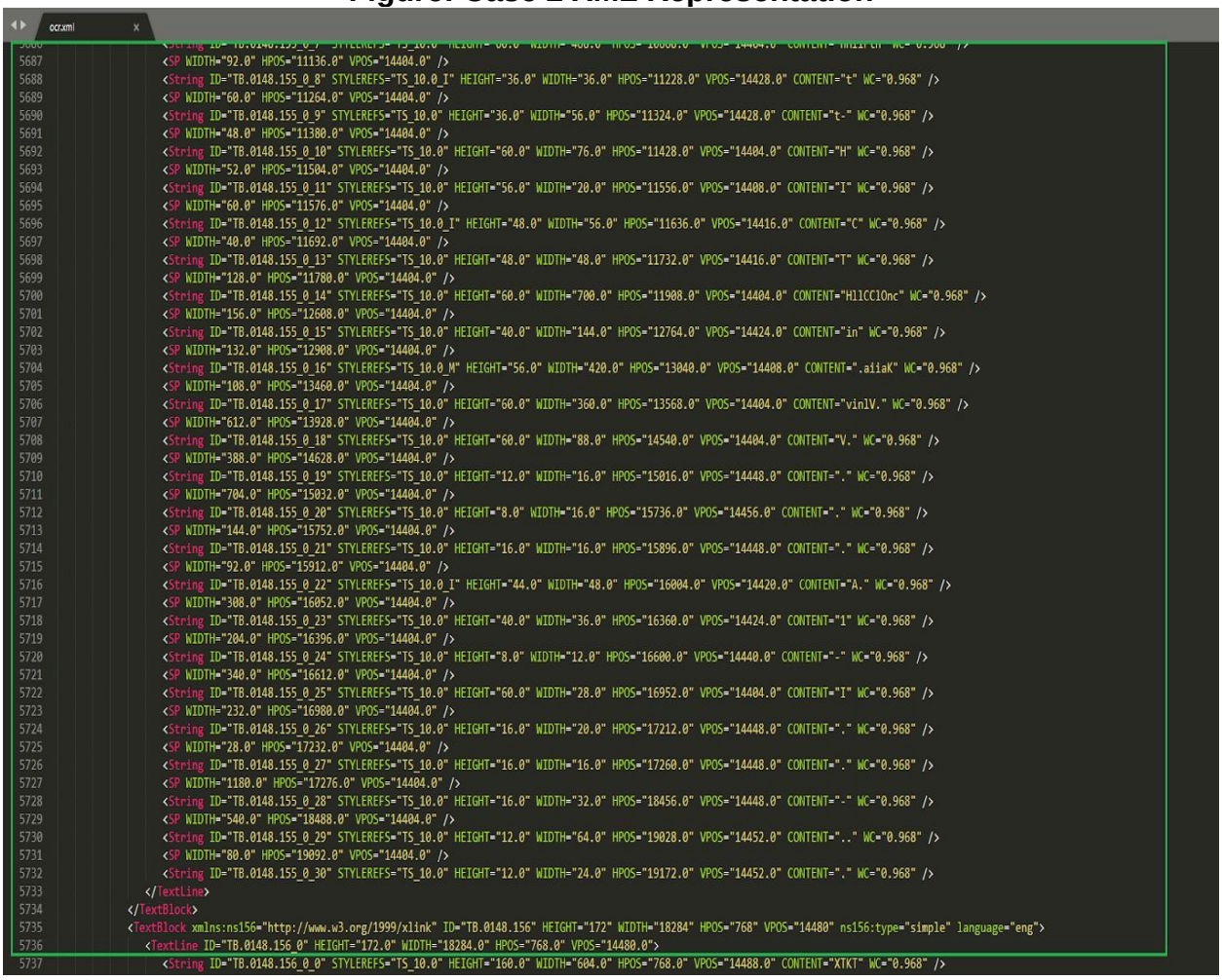
4) If case 1 pass, then store article title, data, metadata in database. Otherwise, discard article title and data and go to next step.

5) Repeat step 1 through 4 until reach the end of the file.

Next page is an example from an xml file and newspaper image. The green box is data from another article. The blue box contains the article data from a different article than the green one, and the first string content of the TextLine is much larger than the rest of the string contents of that TextLine.

Case 2: XML Representation

Figure: Case 2 XML Representation



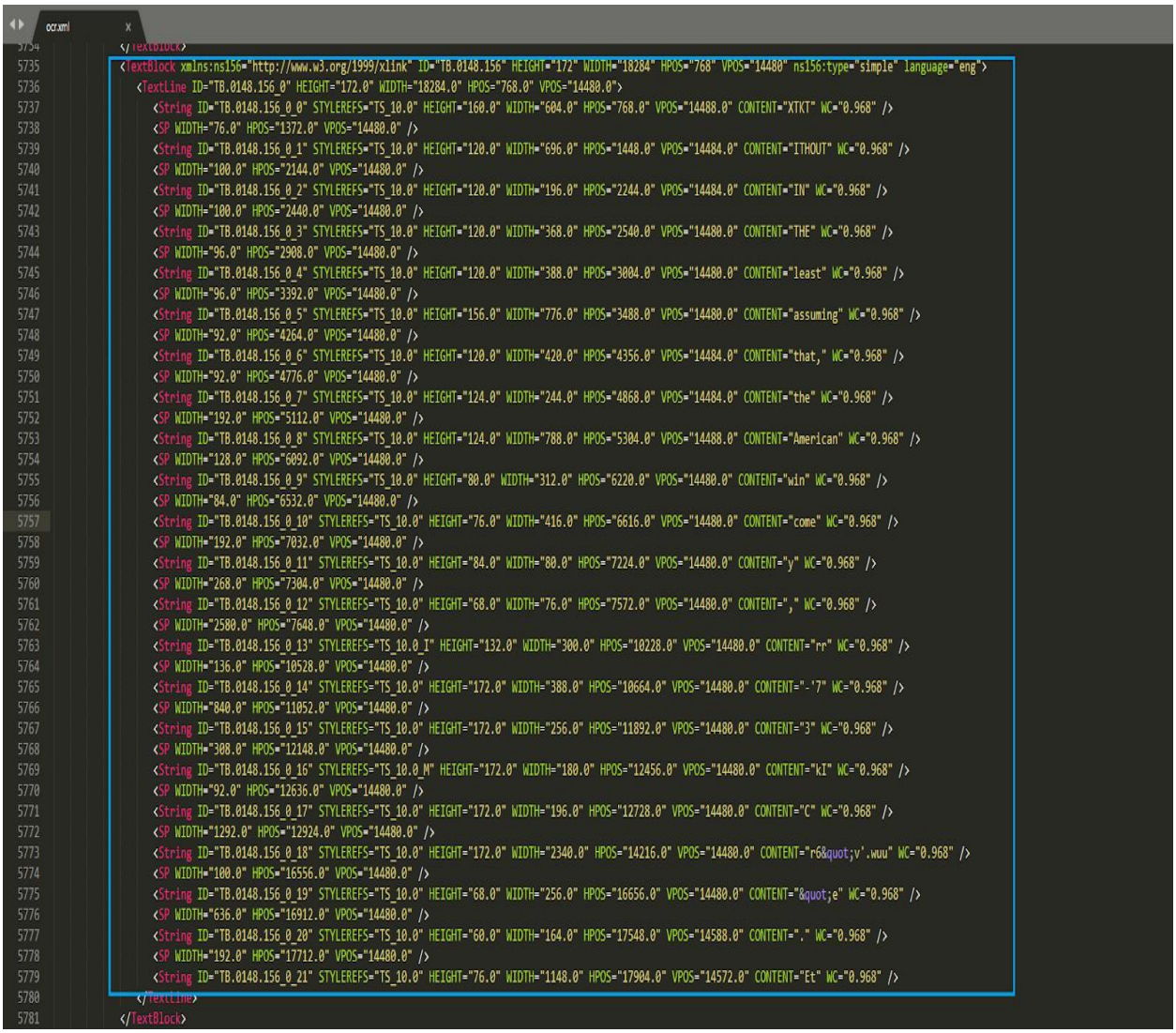
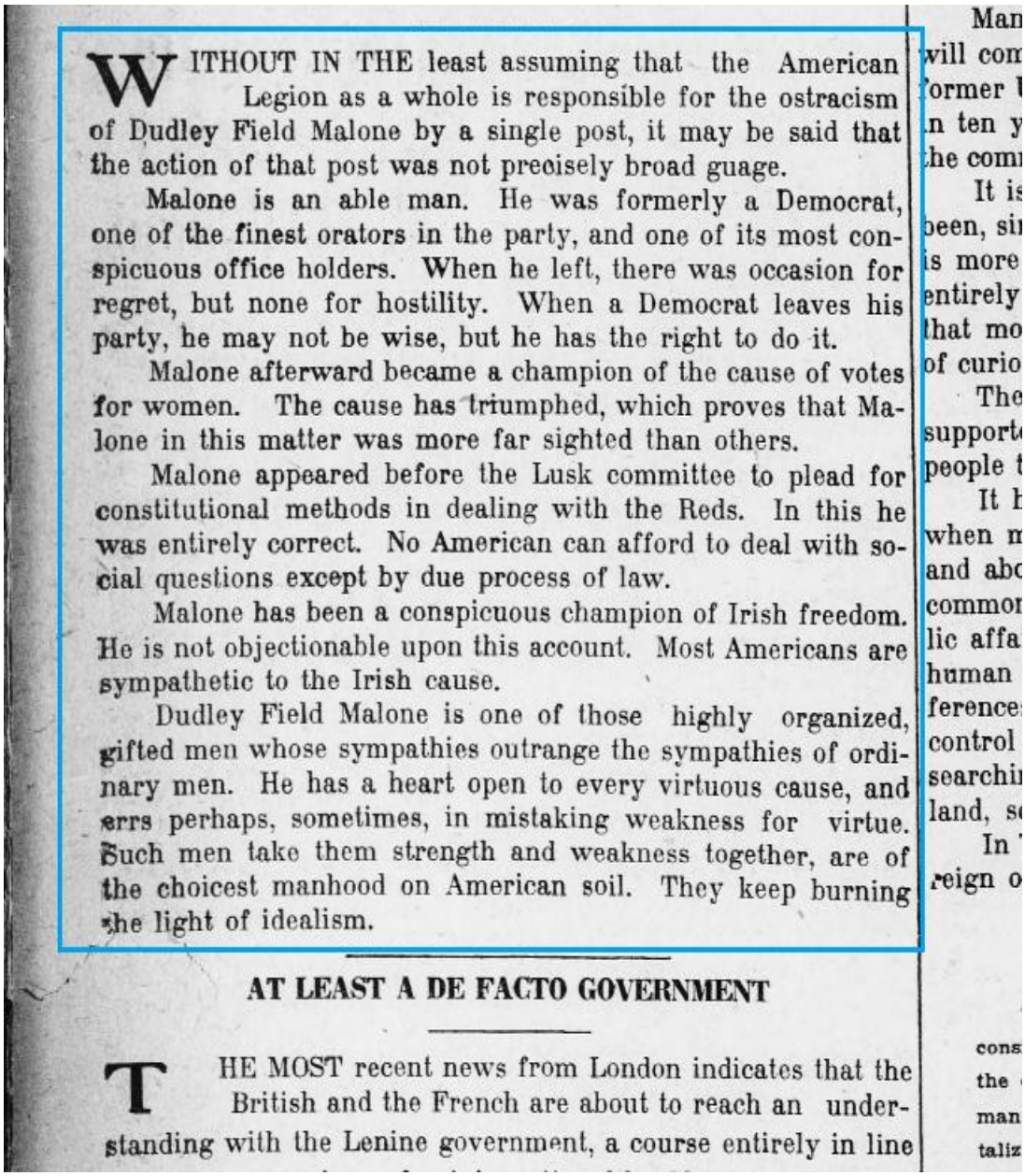


Figure: Case 2 XML Representation 2

Case 2: Newspaper Representation



WITHOUT IN THE least assuming that the American Legion as a whole is responsible for the ostracism of Dudley Field Malone by a single post, it may be said that the action of that post was not precisely broad guage.

Malone is an able man. He was formerly a Democrat, one of the finest orators in the party, and one of its most conspicuous office holders. When he left, there was occasion for regret, but none for hostility. When a Democrat leaves his party, he may not be wise, but he has the right to do it.

Malone afterward became a champion of the cause of votes for women. The cause has triumphed, which proves that Malone in this matter was more far sighted than others.

Malone appeared before the Lusk committee to plead for constitutional methods in dealing with the Reds. In this he was entirely correct. No American can afford to deal with social questions except by due process of law.

Malone has been a conspicuous champion of Irish freedom. He is not objectionable upon this account. Most Americans are sympathetic to the Irish cause.

Dudley Field Malone is one of those highly organized, gifted men whose sympathies outrange the sympathies of ordinary men. He has a heart open to every virtuous cause, and errs perhaps, sometimes, in mistaking weakness for virtue. Such men take them strength and weakness together, are of the choicest manhood on American soil. They keep burning the light of idealism.

AT LEAST A DE FACTO GOVERNMENT

THE MOST recent news from London indicates that the British and the French are about to reach an understanding with the Lenine government, a course entirely in line

Figure: Case 2 Newspaper Representation

Case 3: When scanning the String element content, we should check to see if the String element has a SUBS_CONTENT attribute. If it does, then use that as the String content and skip the next String content. This will make sure we are inputting the correct words into the article.

Implementing case 3 to our parser algorithm will change it as follows:

1. As a String element is scanned check case 3, store the appropriate content as a potential article title, while keeping track of each String element height
2. If the height change decreases and the magnitude of that change is greater than some threshold, then store the previous string as the title and go to step 3.

Otherwise, if the height increases keep scanning and storing the String elements content as part of the title.

3. Keep scanning all String element, whatever comes next is the article data. If the height decreases, keep scanning and saving the String element content as part of the article data. Otherwise, If the height increases and the change is greater than some threshold and case 2 passes, then store the previous saved strings as the article data.

4. If case 1 pass, then store article title, data, metadata in database. Otherwise discard article title and data and go to next step.

5. Repeat step 1 through 4 until reach the end of the file.

Next page is an example from an xml file and newspaper image. The red box indicates where a split word appears from a newspaper article.
Case 3: XML Representation

Red box indicates split word

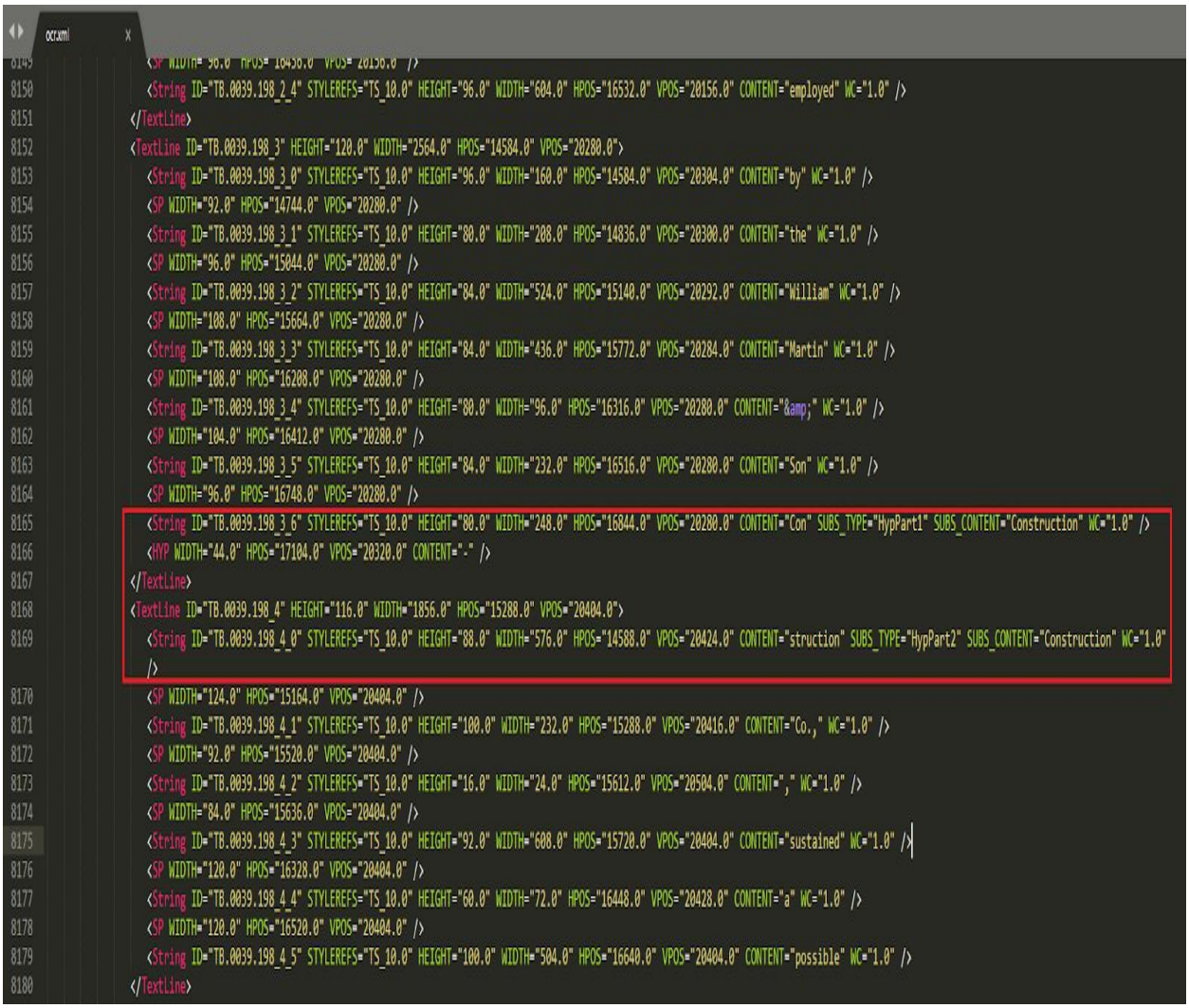
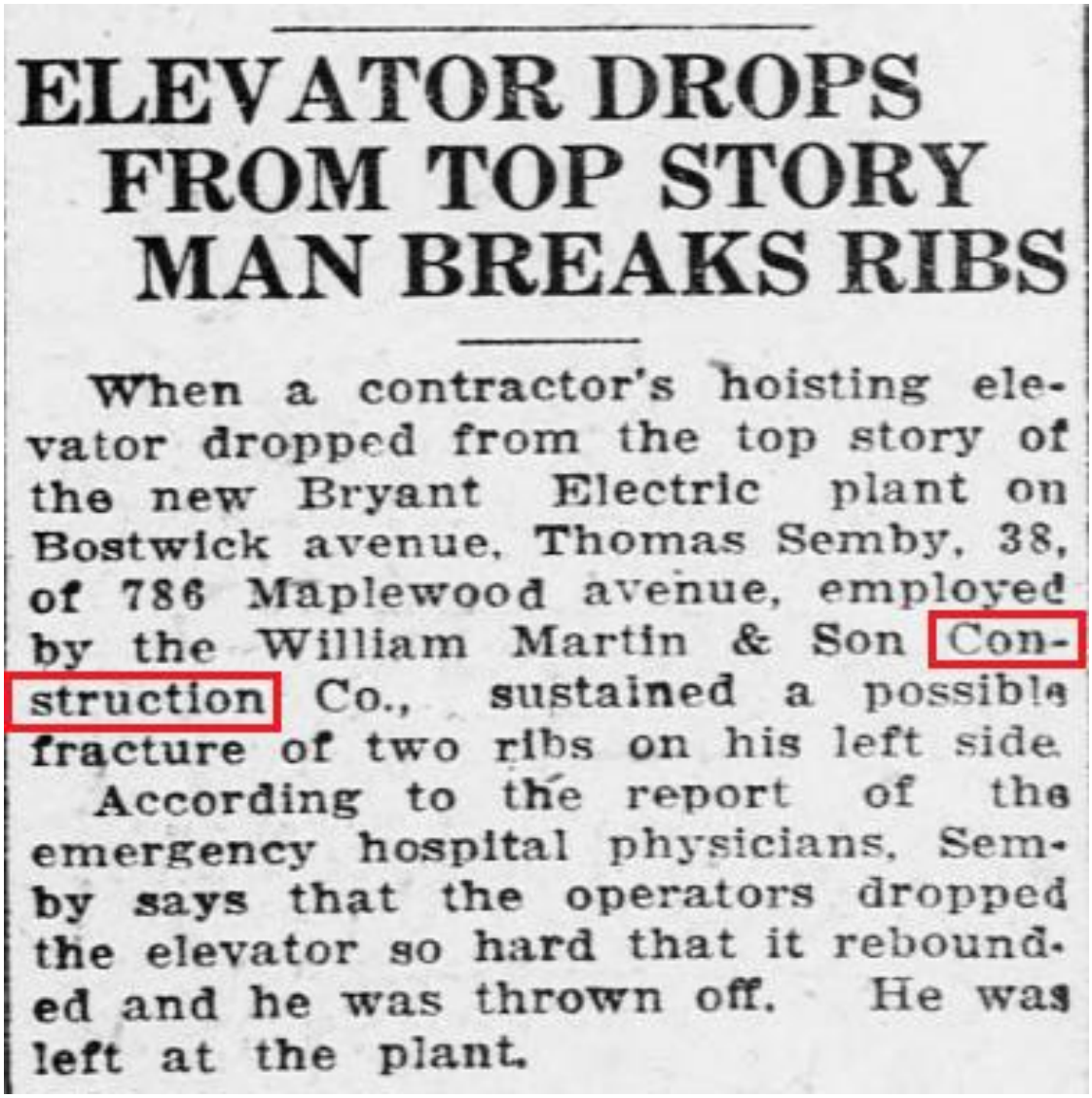


Figure: Case 3 XML Representation

Case 3: Newspaper Representation

Red box indicates split word



ELEVATOR DROPS FROM TOP STORY MAN BREAKS RIBS

When a contractor's hoisting elevator dropped from the top story of the new Bryant Electric plant on Bostwick avenue, Thomas Semby, 38, of 786 Maplewood avenue, employed by the William Martin & Son Construction Co., sustained a possible fracture of two ribs on his left side.

According to the report of the emergency hospital physicians, Semby says that the operators dropped the elevator so hard that it rebounded and he was thrown off. He was left at the plant.

Figure: Case 3 Newspaper Representation

Conclusion

Once this project is completed it will certainly be a powerful and efficient research tool for academics who want to use the information from Chronicling America. Using the several complex parts of this project, a research tool can be constructed. It is important to remember that this project's purpose is not to replace the Chronicling America website, but instead is meant to build an add-on tool that makes the information contained within the Chronicling America website more easily analyzed by researchers.

By using the parser designed and built during this project, it will be possible to break each page contained within the Chronicling America website into articles in a relatively accurate manner. This allows the information within the pages to be queried for more easily, and ensures queries will better reflect the data within the texts of these newspapers.

Once the pages are broken down into articles they can be easily analyzed using various programming tools because users will be able to access query results in CSV files. On top of the CSV files, users will be able to visually see the geographic locations that query results originate from using this project's query heatmap that can be downloaded as a PNG file. These are important fragments of this project, because they further cement this project as a useful tool for serious academic research.

This project's query tool will also make slight enhancements to Chronicling America's query tool. First, because the pages will be separated into articles before being entered into the database and inverted index, our query tool will be allowed to analyze smaller portions of text for relevancy to inputted search terms. This should make queries more accurate and relevant to what users want. This project's query tool will also allow users to save their searches, allowing them to refer back to them while conducting their research. By using an inverted index and a prudent relevancy ranking system, search results will be returned quickly and will be ordered in a way that users will find convenient and useful. Also, this query tool will allow users to narrow down their results using a generous amount of search options, further ensuring the relevancy of the results that will be returned to them. This query tool will be put together using Solr which will allow it to run smoothly and efficiently on any server it will be run on.

This project's website is also organized and built in a logical manner that will make researching easier and more pleasant for users. It is also responsive, meaning that it will adapt to the user's screen size. This allows users to access the website on their computers, phones, or tablets and still have a similar experience without having to zoom in on parts of the website, making their experience more convenient and pleasurable.

The three main portions of this project are the parser, the website, and the query tool. On their own, they do not amount to a useful project, but when they are united

they create a sum larger than their individual parts. By working in sync with one another they create a comprehensive and effective research tool for users wanting to analyze historical data.

References

[1] tutorialspoint.com. “Java DOM Parser Overview.” *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/java_xml/java_dom_parser.htm.

[2] tutorialspoint.com. “Java SAX Parser Overview.” *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/java_xml/java_sax_parser.htm.

[3] tutorialspoint.com. “Java StAX Parser Overview.” *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/java_xml/java_stax_parser.htm

[4] tutorialspoint.com. “Python XML Processing.” *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/python/python_xml_processing.htm.

[5] tutorialspoint.com. “MySQL INDEXES.” *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/mysql/mysql-indexes.htm.

[6] “Understanding Storage Sizes for MySQL TEXT Data Types.” Chartio, chartio.com/resources/tutorials/understanding-strorage-sizes-for-mysql-text-data-types/.

[7] Corcoran, Liam. “How Long Are The Most Shared Stories On Social Media?” NewsWhip, 27 Feb. 2018, www.newswhip.com/2017/01/long-shared-stories-social-media/

[8] “English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU.” English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU, norvig.com/mayzner.htm

[9] “MySQL Data Types.” *W3Schools*, www.w3schools.com/sql/sql_datatypes.asp

[10] Fingas, Jon. “Microsoft officially ends support for Windows Phone.” *Engadget*, www.engadget.com/2017/07/11/windows-phone-support-ends/

[11] “Haxe.” *Wikipedia*, en.wikipedia.org/wiki/Haxe

[12] “Chronicling America - About API: JSON Views.” *chroniclingamerica.loc.gov*, Chronicling America, chroniclingamerica.loc.gov/about/api/#json-views

[13] “Chronicling America - About.” *Chronicling America*, chroniclingamerica.loc.gov/about/

[14] Rubin, Alexander. “Increasing slow query performance with the parallel query execution.” *Percona*, www.percona.com/blog/2014/01/07/increasing-slow-query-performance-with-parallel-query-execution/

[15] “Salt (cryptography).” *Wikipedia*, en.wikipedia.org/wiki/Salt_(cryptography)

[16] The Library of Congress, *Chronicling America*: Historic American Newspapers site. https://chroniclingamerica.loc.gov/help/

[17] Apache Software Foundation. https://lucene.apache.org/solr/

[18] Tan, Kelvin. 2018. http://www.solrtutorial.com

[19] Tutorials Point. *Lucene Tutorial.* https://www.tutorialspoint.com/lucene/

[20] Tutorials Point. *Apache Solr Tutorial.* https://www.tutorialspoint.com/apache_solr/

[21] Google LLC. *How Search Works.* https://www.google.com/search/howsearchworks/

[22] Jain, Saurav. Geeks for Geeks. *Inverted Index.* https://www.geeksforgeeks.org/inverted-index/

[23] Palavuzlar, Mehper C. “What Are the Rules for Splitting Words at the End of a Line?”, *English Language & Usage Stack Exchange*, english.stackexchange.com/questions/385/what-are-the-rules-for-splitting-words-at-the-end-of-a-line

[24] Martin, John. “Difficulties in OCR classification.” *Beyond Recognition*, beyondrecognition.net/limitations-using-ocr-file-classification/

[25] “OCR for detecting bills.” *StackOverflow*, stackoverflow.com/questions/44112843/ocr-for-detecting-bills

[26] Duckett, Chris. “Lack of accessibility should kill CAPTCHAs, but where's the replacement?” *TechRepublic*, www.techrepublic.com/blog/australian-technology/lack-of-accessibility-should-kill-captchas-but-wheres-the-replacement/

[27] Earl, Crista. “Can CAPTCHAs be made accessible?” *American Foundation for the Blind*, www.afb.org/blog/afb-blog/can-captchas-be-made-accessible/12

[28] “Inaccessibility of CAPTCHA - Alternatives to Visual Turing Tests on the Web”, *W*