THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

Design and Implementation of an Information Retrieval System for Academic Publications

JASON CHHAY
SPRING 2021

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Dr. C. Lee Giles
David Reese Professor of Information Sciences and Technology
Thesis Supervisor

Ting He
Associate Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

This thesis outlines the design and implementation for a web-based information retrieval system specifically for academic journals and publications. More specifically, this thesis will focus on improving and sustaining the existing infrastructure of CiteSeerX, hosted by the Penn State College of Information Sciences and Technology. The current implementation of CiteSeerX is analyzed from the process of document crawling, information extraction and ingestion, document indexing, and a web-based search interface face. A selection of new potential features is implemented and prototyped through COVIDSeer, a small-scale search interface built on the CORD-19 dataset to assist the global COVID-19 pandemic research effort. These features are then transferred into a prototype for a future iteration of CiteSeerX that incorporates modern programming languages and frameworks for more efficient querying and a more maintainable codebase. This thesis should also serve to highlight the design and implementation challenges of COVIDSeer and the new system to assist in future work with developing similar search engines for academic publications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Lee Giles and Dr. Jian Wu for their excellent guidance throughout the work carried out for this thesis, the opportunity to work on these fantastic and challenging projects, and the trust in the ownership and project management for the development effort. I would also like to thank Shaurya Rohatgi for the wonderful support and mentorship across all my work with CiteSeerX, COVIDSeer, and development of the new system. I would also like to thank other members of the CiteSeer team, many whom have graduated at the time of this writing or are still studying at Penn State, for their dedicated work in making the implementation of these systems possible.

Chapter 1

Introduction

The SEER Labs group at Penn State is a research group within the College of Information Sciences and Technology that specializes in building systems for information retrieval. This incorporates the entire architecture of crawling and indexing many documents across a variety of servers and virtual machines, extracting the metadata from these documents, and maintaining a web user interface to query these documents. The most notable search engine produced by SEER Labs is CiteSeerX, a search engine that specializes in academic publications across a wide variety of topics. Launched in 2008, CiteSeerX has undergone many modifications in its infrastructure. However, to compete with similar academic search engines such as SemanticScholar there is now a demand for a new system of CiteSeerX to be implemented that takes advantage of modern technologies for more efficient processing and querying of documents as well as a more powerful and responsive web interface. The new system for CiteSeerX must also integrate new user features as well, and that it should provide a framework to build specialized search engines should the need arise.

Before divesting into the implementation of the new prototype system of CiteSeerX, temporarily named NextGenSeer (NGS), it is important to elaborate on the process of information retrieval and how modern search engines continuously provide relevant and useful results to queries instantaneously across an extremely large dataset. It is also important to elaborate on CiteSeerX's history, its current features, personal work to improve the system, and current limitations. Afterward, this thesis will detail the personal development efforts of the

specialized search engine COVIDSeer and how it influences the development efforts for NGS. Before divesting into the implementation of the new prototype system of CiteSeerX, temporarily named NextGenSeer (NGS), it is important to elaborate on the process of information retrieval and how modern search engines continuously provide relevant and useful results to queries instantaneously across an extremely large dataset. It is also important to elaborate on CiteSeerX's history, its current features, personal work to improve the system, and current limitations. Afterward, this thesis will detail the personal development efforts of the specialized search engine COVIDSeer and how it influences the development efforts for NGS.

Chapter 2

Background

In the modern age of information, search engines are a necessary tool to traverse the trillions of web pages that exist on the Internet. With so much information readily available, the challenge persists with how one can navigate through this information to find what they need. Today, popular search engines like Google, Bing, and DuckDuckGo exist that serve to allow users to find the information they need in the form of links to webpages. Specialized search engines also exist, such as Expedia which searches across airplane flights and hotels, or SemanticScholar which is used for finding academic publications. Information retrieval systems are extremely useful for finding useful results instantaneously across an extremely large and unstructured dataset. Also, unstructured information will need to continuously be added to the dataset to ensure that it stays relevant. For this to occur, there are several key processes that a modern search engine will need to undergo to make this feasible.

Crawling

The first stage in the information retrieval process is crawling for the documents. The crawling stage is the process in which the documents that are being queried in the information retrieval process are located and collected. The challenge with crawling is that large amounts of data across a variety of different subjects must be located continuously to ensure that modern and relevant documents are being collected. Crawling must also be performed at frequent intervals to locate documents or pages that are newly created. On top of that, the pages need to be continuously crawled in case they are updated or deleted, as it is not practical to return pages to

the user that don't exist. An example of crawling is with the Google web search engine locating trillions of pages across the Internet. As there is no specific location where the URLs to every single site created exist, Google will need to find a means of locating web pages.

Crawling is feasible using programs called "spiders". Appropriately named, spiders continuously traverse the Web to aggregate documents by exploring links from other documents, then exploring the links within those documents, and so on. Spiders are alternatively called "robots", "crawlers", "agents", and "worms". Spiders begin their traversal on an initial source that will contain a large number of links to unrelated sources. The spider then traverses through each link, collects the document information, locates the links within that document, and traverses through them.

It should be noted that in this case, the term "traversal" does not mean that each web page is being physically navigated to in the same way that a human would open a website on their browser. The "traversal" from Page A to Page B means that, either through a web client or a robot, the URL on Page A to Page B is extracted. Using that URL, the content from Page B would then be fetched by issuing a request to the webserver that Page B is hosted. That being said, some programs do physically travel from one page to another. A discovery robot is a type of bot that physically explores the web to collect URLs.

It should be noted that multiple factors go into what links to traverse to and the process of how the spider should traverse. A useful information retrieval may need to have a variety of documents across a diverse array of sources and subjects. If a spider allocates too many resources trying to traverse every single link within a document or every possible path possible that originates from a document, then the spider will likely end up collecting documents that are very similar to each other and will not collect documents from multiple subjects. The challenge

of navigating to each document can be modeled as a graph theory problem, in which spiders will either utilize a breadth-first search approach or a depth-first search approach.

Both approaches have their advantages and disadvantages. In a breadth-first search approach, the order in which every link in a traversed document is added to a queue for further traversal. Links to documents from an unexplored host or hosts that were not explored recently are given higher priority. For instance, a spider may prefer to navigate from www.psu.edu to the unexplored www.pa.gov as opposed to www.psu.edu/apply. This is effective for maximizing website coverage. Generally, this is the preferred method for crawling. Depth-first search employs a different method in which each link in a document will have its child links explored until there are no longer any unique links before backtracking and locating other links to traverse to. Depth-first search is more important for covering documents that are deemed “important”, in which the criterion is defined by the spider’s developer. Depth-first search is useful for maximizing the website coverage for specific subjects.

One of the bigger challenges in crawling is ensuring that spiders do not overwhelm the web servers that the documents are hosted on. While generally a reckless and uncourteous practice, it may lead to the hosting provider blacklisting the IP address of the spider’s host, preventing additional crawling. Web hosts can limit the extent that spiders crawl their pages as well as what specific spiders are allowed using a file named “robots.txt”, which specifies what pages or resources a spider may or may not retrieve from a site.

Tokenization

Before diving into how the crawled documents are stored, it would be best to explain how the documents that are crawled are processed for retrieval. It is not that simple to do an exact text match for a query within a document. It might be the case where the query does not have the exact capitalizations as the text that appears in the document body, or that it might contain whitespace and other words interlaced between terms (for instance, you might want to query the term “computer vision cars” in a document that contains the phrase “computer vision used in cars”.) One might also want to query for synonyms of words, such as querying “vehicle” instead of “car”.

The process of converting the sequence of characters in a document is known as “tokenization.” There are many ways that a sentence can be tokenized, depending on the specifications of the IR system. For instance, early methods of tokenization involved using words composed of alphanumeric characters and have a length greater than three, converted into lowercase. The string “Bigcorp’s 2007 bi-annual report showed profits rose 10%.” would be tokenized to “bigcorp 2007 annual report showed profits rose”. However, there are several limitations to this approach. Small words composed of one or two characters might be important for queries when combined with other words, such as “world war ii”, “el paso”, and “j lo”. Words with hyphens and apostrophes are common, such as “e-bay”, “wal-mart”, “rosie o’ donnell”, and “england’s tallest cities”. Capitalizations of words can denote different meanings than their lowercase equivalent, such as “Bush” and “Apple”. There is a multitude of other limitations. In the case of TREC collections, apostrophes are omitted within a word such that “O’Connor” is tokenized as “oconnor”. Abbreviations would be interpreted as a series of single characters with period sin between, so “I.B.M.” becomes “ibm”.

There also exists a natural language processing component to tokenization. This is for the use case where a user might query for a word or phrase that is in a different tense within the document. For instance, a user might be querying “Michael Phelps swimming in Olympics” whereas the documents might only contain phrases like “Michael Phelps swam”. Either a dictionary or an NLP algorithm that detects the past conjugation of words would need to be used to detect that “swimming” and “swam” refer to the same keyword.

Another consideration with tokenization is the use of stopwords. Stopwords are a list of words that do not contain any meaning outside their connection with more interesting keywords. Stopwords include determiners like “the”, “a”, “an”, “that”, and “those” which are used to describe a noun in terms of qualifiers like location or quantity, pairing it with words like “over” and “under”. As stopwords are not generally considered useful or important to the user’s query, they are usually omitted entirely during the tokenization process. Removing these words will decrease the index size, improve retrieval speed, and ensure that results are more effective.

Indexing

The next stage in the search engine process is the indexing stage. This is the stage in which the documents that are crawled will be stored in such a way that they can be retrieved easily. Consider the analogy where an index is like a library. A library is composed of a very large number of books on a variety of topics written by many different authors. Ordinarily, one would not look through every single book in a library until they find the book that they are looking for. Fortunately, the books in the library are usually sorted by subject then by author in alphabetical order. We can consider an index to be organized similarly except for the documents that are crawled.

The most common form of an index is the inverted index, from which all modern search engines are derived because of its flexible structure and efficiency. An inverted index is analogous to the index at the back of a textbook which lists important keywords in alphabetical order with the pages they can be found in. In an index, these keywords are referred to as the "index term". An inverted index, named because it associates documents to words as opposed to intuitively associating words to documents like in a conventional index, is a hash table that corresponds to all the index terms within a document to a specific metric. These metrics are referred to as an "index list" and determine how an index term is associated with each document. Index lists could include which documents an index term is stored in, the frequency of an index term in a document, and the positions in which an index term appears in a document.

Table 1. Example of document collection, taken from Wikipedia entry for *tropical fish*

Document Number	Document Content
1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salter water species.
2	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
3	Tropical fish are popular aquarium fish, due to their often bright coloration.
4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Table 2. Example of an inverted index, based on word appearance

and	**1**	often	**2, 3**
aquarium	**3**	only	**2**
are	**3, 4**	pigmented	**4**
around	**1**	popular	**3**
as	**2**	refer	**2**
both	**1**	referred	**2**
bright	**3**	requiring	**2**
coloration	**3, 4**	salt	**1, 4**
derives	**4**	saltwater	**2**
due	**3**	species	**1**
environments	**1**	term	**2**
fish	**1, 2, 3, 4**	the	**1, 2**
fishkeepers	**2**	their	**3**
found	**1**	this	**4**
fresh	**2**	those	**2**
freshwater	**1, 4**	to	**2, 3**
from	**4**	tropical	**1, 2, 3**
generally	**4**	typically	**4**
in	**1, 4**	use	**2**
include	**1**	water	**1, 2, 4**
including	**1**	while	**4**
iridescence	**4**	with	**2**
marine	**2**	world	**1**

Indexing usually occurs on distributed systems, in which an index occurs on multiple computers or sites across a network. This is to increase performance, as distributing the indexes for a portion of documents, known as document distribution, allows for concurrent indexing and querying. Distributing the indexes across terms, known as term distribution, allows for queries to be processed concurrently as well. Copies of indexes can also be stored across multiple sites to reduce communication delays for querying, a process known as replication. There is also a form of distribution known as peer-to-peer search in which each index and collection of documents are stored within their own node on a network.

Querying and Retrieval

Following the construction of an index, the next stage in the IR process is the retrieval of documents after a search query is made. There is a multitude of ways that a query can be formatted and for documents to be retrieved correspondingly. As mentioned previously, an effective document retrieval system must be able to return a list of relevant results to the user's query across large storages of data. Queries should also be intuitive to the user, such that a user will be able to easily understand how to utilize them and that they may be combined with other search queries for more effective results.

The earliest and most simple model for querying is the Boolean retrieval model. In this model, documents are queried based on exact matches. If a document does not exactly match the query, then it is omitted from the results. Unlike other retrieval models, Boolean retrieval does not perform any ranking of the results as it assumes that all documents are equal in ranking. As the name suggests, Boolean retrieval queries hinges on the retrieval have an outcome of either True or False, and that queries can be combined using AND, OR, and NOT operators. Boolean retrieval is very intuitive for the user to make queries for, and it can be utilized for any metadata field such as creation dates or document types. However, Boolean retrieval entirely relies on the way that the user makes the query, and that results will likely not be very relevant.

An alternative approach to document retrieval is with the vector space model. This model has the advantage of being very straightforward but also opening the door for incorporating ranking, weighting, and relevance feedback into the rankings. The vector space model represents the index as a k-dimensional vector space in which k is the number of index terms. Every document D within the vector space is represented by a vector of index term weights. A document Di would be represented as the following:

$$Di = (di1, di2, \ldots, dik - 1, dik)$$

in which *dij* represents the weight of the jth index term for the ith document. A matrix of n documents that contain t unique terms across the entire collection would be represented as below, where each row represents an indexed document, and each column represents the weight for each index term.

	$Term\ 1$	$Term\ 2$	$\cdots$	$Term\ t$
$D1$	$d11$	$d12$	$\cdots$	$d1t$
$D2$	$d21$	$d22$	$\cdots$	$d2t$
$\vdots$	$\vdots$			
Dn	$dn1$	$dn2$	$\cdots$	dnt

A query is also represented as a vector of term weights, identically to that of a document. A query Q would be represented as the following:

$$Q = (q1, q2, \ldots, qk - 1, qk)$$

where *qj* represents the weight of the jth index term in the query. Consider the following example of an index with the corresponding weights for each index term. In this example, the term weights are the number of occurrences an index term appears in a document. Stopwords are not included within this vector space.

Table 3. Example of a term-document matrix for a collection of documents, based on frequency

D_1	Tropical Freshwater Aquarium Fish
D_2	Tropical Fish, Aquarium Care, Tank Setup.
D_3	Keeping Tropical Fish and Goldfish in Aquariums, and Fish Bowls.
D_4	The Tropical Tank Homepage – Tropical Fish and Aquariums

Terms	**Documents**			
	D_1	**D_2**	**D_3**	**D_4**
aquarium	1	1	1	1
bowl	0	0	1	0
care	0	1	0	0
fish	1	1	2	1
freshwater	1	0	0	0
goldfish	0	0	1	0
homepage	0	0	0	1
keep	0	0	1	0
setup	0	1	0	0
tank	0	1	0	1
tropical	1	1	1	2

In this model, documents are ranked base on how similar they are to the query. The most successful metric to measure similarity is the cosine correlation similarity measure, in which the cosine of the angle between the query and document vectors are calculated. If the vectors are normalized, meaning that the query and document vector lengths are identical, then the cosine of the angle between two vectors with identical attributes will be 1. Conversely, the cosine of the angle between two vectors that do not share any terms will be 0. The cosine measure is derived from the following equation:

$$Cosine(D_i, Q) = \frac{\sum_{j=1}^{t} d_{ij} \cdot q_j}{\sqrt{\sum_{j=1}^{t} {d_{ij}}^2 \cdot \sum_{j=1}^{t} {q_j}^2}}$$

There are different methodologies in determining the weights for each index term. One methodology is the tf-idf method, which assumes that the frequency of a term correlates with its importance. There are two components to the tf-idf methodology. The first is the term frequency (tf) of the document, which reflects how important a term is within a specific document based on if it appears very frequently. This is simply the number of times that a term appears in a document, normalized with the total number of terms that appear within the document or query. Normalization for term frequency will allow for documents of different lengths to be treated similarly.

$$tf_{ik} = \frac{f_{ik}}{\sum_{j=1}^{t} f_{ij}}$$

The second component is the inverse-document frequency (idf), which reflects how important the index term is across the entire collection of documents. If an index term frequently appears in multiple documents, then the assumption is that the index term is trivial and will not be useful in retrieval. The inverse-document frequency can be calculated as follows:

$$idf_k = \log \frac{N}{n_k}$$

Together, these terms can be combined by multiplying them together for a collective weight, hence the name “tf.idf”.

$$d_{ik} = \frac{(\log(f_{ik}) + 1) \cdot \log\left(\frac{N}{n_k}\right)}{\sqrt{\sum_{k=1}^{t} [(\log(f_{ik}) + 1.0 \cdot \log\left(\frac{N}{n_k}\right)]^2}}$$

Evaluating the Performance of Search Engines

Two metrics are considered when it comes to evaluating a search engine: effectiveness and efficiency. Effectiveness refers to whether the results that are returned are useful. This is mostly subjective, as it is defined as how much the rankings returned from a query match closely with that of the user's self-identified rankings. Efficiency refers to how optimized a search algorithm is in terms of time and storage. Generally, an IR system is developed prioritizing effectiveness first and finding methodologies to improve effectiveness. When a satisfactory model is developed, additional steps will be taken to improve its efficiency.

While effectiveness is usually determined by the user, there are several ways to quantify how effective is a search engine. One very useful methodology is through query logs. Query logs provide a large amount of data on how a user interacts with a system. A query log will compose of some form of user identification or user session identification, query terms, user results for a given query, and timestamp. Clickthrough data can be parsed from these logs that can determine how long a user stays on a page, which results that a user clicks on, and when they exit a page. It is generally assumed that a document that is clicked on by a user will rank higher in their preference.

Effectiveness is measured in terms of recall and precision. These measures are based on the notion that given a collection of documents and a query, there is a set of documents that are retrieved and a set of documents that are not retrieved given a query. There is also a set of documents that are relevant to the query and a set of documents that are irrelevant to the query. We can refer to the set of relevant documents as *A*, the set of irrelevant documents as *A'*, the set of retrieved documents as *B*, and the set of unretrieved documents as *B'*. Recall and precision would then be defined as the following:

$$Recall = \frac{|A \cap B|}{|A|}$$

$$Precision = \frac{|A \cap B|}{|B|}$$

Table 4. Classifications for sets of documents retrieved by an IR system

	Relevant	**Not Relevant**
Retrieved	$A \cap B$	$A' \cap B$
Not Retrieved	$A \cap B$’	$A' \cap B$’

Based on Table 4, we can also evaluate two types of errors from a retrieval: false positives (the number of retrieved documents that are irrelevant) and false negatives (the number of unretrieved documents that are relevant). The general idea is that an effective search engine will be able to deliver all the relevant results given a query and that any irrelevant result should not be retrieved.

Compared to effectiveness, efficiency has more tangible and quantifiable metrics to measure it. Table 5 lists some of the metrics that are utilized in determining efficiency.

Table 5. Metrics for Efficiency

Metric name	**Description**
Elapsed indexing time	The amount of required time in order to build an index for a collection of documents.
Indexing processor time	The amount of CPU seconds in order to build an index for a collection of documents, not including time delays for I/O or utilizing parallelism.
Query throughput	Number of processed queries per second
Query latency	The amount a time it takes between a user issuing a query and receiving a response.
Indexing temporary space	Amount of temporary disk space when an index is being built.
Index size	Amount of allocated storage for an index

A desirable search engine should have low latency and high throughput. However, these metrics trade-off with each other and cannot be maximized in tandem. An additional metric is hardware cost. A search engine could have a target latency that can be achieved by implementing vertically or horizontally adding hardware to improve individual storage and processing power or distribute them across multiple machines.

Tools for Developing Search Engines

Implementing a large-scale search engine is not a trivial process at any stage of information retrieval. Fortunately, several powerful tools implement search engines much more practically through the use of powerful and intuitive frameworks. These frameworks can be used for IR systems of any non-specific document type and come with additional features aside from querying results to provide a more effective user experience, such as faceted searching. Two widely used search frameworks are Apache Solr and ElasticSearch. Apache Solr is an open-sourced search platform based on Apache Lucene, which is utilized for indexing documents, that is effective for text-based searching for enterprise applications. Elasticsearch is another powerful open-sourced framework developed by Elastic that allows for indexing and querying and is very useful for text search and data analytics through its aggregation features. Both of these frameworks are free to use and are easy to install, removing significant overhead in developing the indexing and retrieval processes and allowing developers to focus on the practical usages of these frameworks.

Chapter 3

CiteSeerX

CiteSeerX is an academic search engine hosted at Penn State's College of Information Sciences and Technology. It is maintained by the CiteSeer group under the supervision of Dr. Lee Giles from Penn State and Dr. Jian Wu from Old Dominion University. It is currently the main search engine that is maintained by the CiteSeer group, however, there is a need to update the existing codebase to utilize more modern frameworks and technologies for it to compete with similar search engines.

History

The original iteration of CiteSeerX, named CiteSeer, was developed in 1997 under Lee Giles, Kurt Bollacker, and Steve Lawrence as the first digital library and search engine that provides citation linking through autonomous citation indexing. Since then, CiteSeer would grow to over 750,000 indexed documents and 1.5 million user requests. After ten years of public accessibility, a new iteration of CiteSeer known as "CiteSeerX" was implemented to address shortcomings of the original system and provide a more robust solution to several needs of the research community.

Features

CiteSeerX provides a myriad number of user features. The principal feature is autonomous citation indexing, in which citations are automatically extracted and indexed from a

publication for querying and evaluation. These citations are also automatically linked across papers. This can be very useful for determining how many times a paper is cited in other publications. The motivation is that a document that is cited a high number of times is considered very influential, and thus very useful for the user.

Additional features for CiteSeerX include providing users notifications of new papers, summaries from papers within the context of query terms, submissions and automatic harvesting of papers, metadata extraction of PDFs, disambiguation of authors from other authors, citation statistics, full-text indexing of papers, and a personal content portal known as “MyCiteSeer” in which users can create personal collections of papers.

Infrastructure

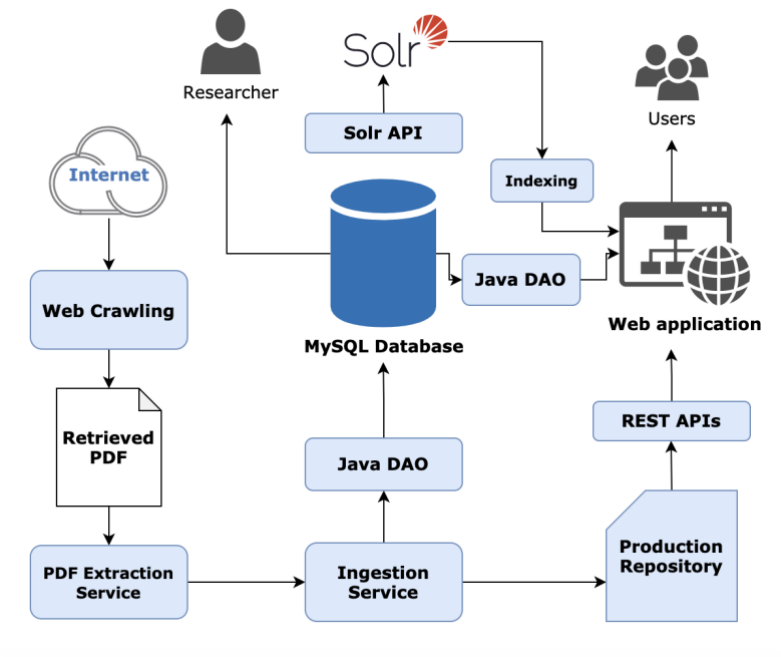

Figure 1 Architecture for CiteSeerX

Figure 1 provides an overview of the current architecture for CiteSeerX. The system is initiated by the crawling of PDF papers that are available from open access sources. The PDF documents are converted into text and a classification service discards any PDFs that are not academic documents. A machine-learning extraction system then retrieves the full text, metadata, citations, key phrases, acknowledgments, and any information not represented in a text format such as figures and tables. All of the textual information is stored in a MySQL database with additional files written into a production repository. An Apache Solr-powered search implementation retrieves full text for documents by searching across the MySQL database and the production repository to be displayed in the web user interface.

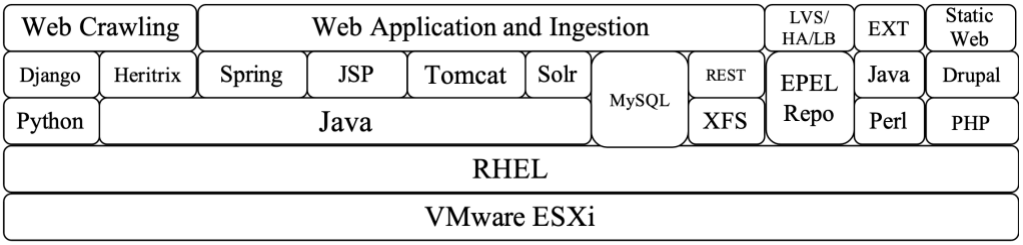

Figure 2. CiteSeerX software stack

In terms of the actual technologies being used to implement the current system, the entirety of the system exists across multiple servers and virtual machines existing on a VMWare ESXi private cloud utilizing a Red Hat Enterprise Linux (RHEL) operating system. The web crawler is implemented using the Python-based Django framework and Heritrix, a Java-based web crawler. The web application itself is built through the Spring Framework, with the frontend served through JSP files and utilizing Solr for the search functionality. The web application is then deployed on a Tomcat web service. An XFS file system and an internal REST API is used to retrieve documents from the production repository. A high availability (HA) load balancer (LB) is used to provide the Linux virtual service (LVS) to direct users to the virtual IP of the web

application's landing page. The extraction service is built off of Java and Perl and a static website that provides background information on CiteSeerX is built using Django.

Design Challenges

From an architecture standpoint, the current system faces a major scalability issue. As the number of ingested academic papers is rapidly increasing, the storage requirements will eventually reach an unsustainable level of demand that will severely negatively impact the user experience, hinder document retrieval performance, and create the risk of permanent data loss. At the moment of writing, these flaws do not pose a major issue on the system but will need to be addressed as soon as possible.

The ingestion module which stores the documents into their respective databases serves as a major bottleneck for scalability. In comparison with the web crawler and extraction modules, which have a throughput of 500,000 and 200,000 documents per day respectively, the ingestion module only has a throughput of 50,000 documents per day. This is a result of being a single-threaded system and that the process of near-duplication checking requires an initial read from the MySQL database. MySQL and other relational databases tend to suffer in performance proportionately as their size increases. Storing more documents will result in a greater proportion for document keys to be read directly from the disk as opposed to the cache, decreasing the hit rate and causing the entire module to slow down. The MySQL database limitations can be resolved by distributing data across multiple shards. However, this will result in increased latencies for queries that involve multiple table joins. As a result, it could take several days up to a week to dump and import a database.

Data sharing and backups will be heavily impacted by the scalability demands as well. As the number of collected open-accessed documents grows from 10 million to approximately 35 million, the total size of the file collection will be 53 TB. Incorporating the database and index, which has a disk size of 1.8 TB and 1.2 TB respectively, and including their backups, the total disk space requirement will be approximately 109 TB. The physical requirement for storing this much memory is not an issue, as it can be easily achievable by distributing it across multiple servers although it could be feasibly stored on a single server. The bottleneck lies with network bandwidth for data transfer, in which the average speed between servers is 20 MB/s could lead to the entire repository taking at least 32 days to create a backup.

In terms of web infrastructure, the current CiteSeerX codebase has several limitations. Currently, the codebase for CiteSeerX exists on production web servers hosted by the College of Information Sciences and Technology. While a git repository exists for the codebase, it is not being source tracked on the production server. As such, any additional changes that will need to be made to the web infrastructure have to be made manually on each production web server. Portability is another challenge, in that if a user wants to host the CiteSeerX web application they will need to manually install all the dependencies, which can lead to versioning issues if not being installed on the correct operating system.

The frontend is fairly limited and does not provide much room to learn user behaviors through query logs and a restrictive user experience. The current user interface only allows the user to text-based query without any additional functionality to narrow down user results. The user is unable to filter their search query through aggregated facets, nor are they able to determine what subject is the publication aside from the extracted metadata. The document view page is also fairly unintuitive. There are tabs for other papers that either cite the same citations or

are cited by the same citations under the “Active Bibliography” and “Co-citation” tabs respectively but do not list any descriptive text that describes what these fields are or how they are represented. This can lead to confusion for first-time users in such a way that they may not fully take advantage of these features.

There are also safety considerations that must be made as well. Cross-site scripting is a type of exploit in which malicious inline-JavaScript code can be executed on the browser side by injecting it through the application. This can be accomplished by having a malicious user sends a request to the web page that contains the JavaScript code to be injected, which is then rendered and executed on the web page. An example of this is a text field in a form in which the page adds the input in a paragraph element in the HTML DOM. A malicious user could add in a piece of JavaScript code in that text field, which would then be injected into the HTML DOM and executed. This usually happens when a website does not validate its user input. Several solutions can be used to prevent this. Currently, CiteSeerX removes any text within a query that is encapsulated between angle brackets, such that no DOM elements can be inserted in a search query.

Aside from technical challenges, CiteSeerX also poses some legal considerations that it must make. Due to the nature of how papers are crawled and indexed, some papers will likely be available for reading in which the author or publisher will not want it to be accessed through CiteSeerX. As such, CiteSeerX must comply with any takedown notices to respect the Digital Millenium Copyright Act. To facilitate this process, a link to Penn State’s page that details the process for reporting copyright infringement is located on every single page.

Comparison with Similar Search Engines

The comparison with competitor search engines will primarily focus on the user experience and functionality of the web application as opposed to the overarching architecture. Semantic Scholar is a similar academic search engine developed by the Allen Institute for Artificial Intelligence. Semantic Scholar provides a wide array of features that CiteSeerX lacks. The search engine contains a list of suggestions for papers by title as the user makes a query in the search bar. CiteSeerX can make queries for authors and papers, however, they are in two separate search fields. These queries are consolidated into a single search bar for Semantic Scholar such that the user sees a list of author and paper results on the same search engine results page. There are additional facets for filtering results by field of study, date range, whether a PDF is available, publication type, author names, and journal and conferences. The document results page for Semantic Scholar also provides additional features and information. Both CiteSeerX and Semantic Scholar provide the PDF for the full paper, cross-linked documents that cite the paper, and extracted metadata including the paper's references which are cross-linked to other ingested papers. However, only Semantic Scholar displays a list of related papers to recommend to the user as well as utilizing classification to determine what topics were included.

Google Scholar is another academic search engine, however,r instead of retrieving results from ingested papers, it retrieves external links to the paper from the publisher site. As such, Google Scholar does not provide the full extracted metadata for a given paper. Much like CiteSeerX however, they do provide a list of citations for a paper and author disambiguation, in which a list of all papers written by a specific author are listed. Much like Semantic Scholar, they allow for the user to narrow down their results further by publication date, a feature that CiteSeerX lacks.

Chapter 4

Implementation and Design of COVIDSeer

COVIDSeer is a specialized academic search engine built by the CiteSeer group in mid-March 2020 that focuses on publications relating to SARS-CoV-2 and similar coronaviruses. These papers include research on different treatments and preventive measures. While the primary goal of COVIDSeer was to implement a system that would be useful for the ongoing medical research effort, two technical goals were hoped to have great implications for future CiteSeer projects: having a framework for a specialized search engine such that the web infrastructure could be extrapolated to a different search engine albeit with a different index to query from, and the implementation of useful features that could also be utilized in the new CiteSeer system.

Motivation

On March 16, 2020, The White House announced a call to action to the machine learning community to develop a solution that could help biomedical domain experts and policymakers identify effective treatments and preventive measures for COVID-19, at the onset of the pandemic's global spread. As such, the Allen Institute of AI (AI2) collaborated with The White House Office of Science and Technology, the National Library of Medicine, the Chan Zuckerberg Initiative, Microsoft Research, and Kaggle to compile and release the first version of a large dataset of publications and preprints, extracted into a JSON format, on COVID-19 and relevant coronaviruses such as SARS and MERS. This dataset is referred to as CORD-19 and has since grown from an initial set of 28,000 papers to 140,000 papers. Given the background of the

CiteSeer group in maintaining an information retrieval system for academic papers, there was a strong interest in implementing a unique search engine that would utilize this dataset.

Features

COVIDSeer is a minimalistic yet practical search engine that contains a decent number of features that focus primarily on making it easier to query for specific papers and find relevant information to what the user may be looking for. There are three main pages: a home page where the user is presented with the COVIDSeer logo and search bar, a search results page in which the user is presented with a list of documents that match their query that they can paginate through alongside a window to filter their results, and a document view page which displays the metadata for a specific document in a readable user interface as well as a downloadable JSON format. The document view page also displays a list of papers that are similar to that of the one that the user is viewing, with links to each of their document view pages.

On the search results page, the user can perform a faceted search. Given a search query, the user will be provided with a list of the ten most frequent author names, journals, and keywords that are aggregated from the search results. The user may click on these terms to filter the search results such that only papers that have that term in the respective metadata field will display. A year slider is also implemented to narrow down the publication year for each result. In addition, each result will display the metadata for authors, journal, publication year, and keywords which can be clicked on to also filter the results.

As the CORD-19 dataset only contains document metadata in a JSON representation, COVIDSeer provides a link to the PDF of the paper alongside its digital object identifier (DOI)

which can be used to redirect the user to the paper's publisher's page by appending the DOI as a URL path for https://doi.org/. The publisher page will provide more robust metadata for the paper than what the CORD-19 dataset initially provides.

Implementation

The web user interface for COVIDSeer is built utilizing the Django web framework. Django was selected due to its ease-of-use in serving web files, its Python backend which allows for the integration with Elasticsearch's Python library, and the utilization of internal Django boilerplate code for implementing web-based search engines for more rapid prototyping and development. The Django REST Framework was also utilized to make certain functionalities asynchronous. Django follows a conventional Model-View-Controller structure such that all information retrieved from the backend is processed on the initial page load. This can lead to very long loading times for pages that require a lot of information to be retrieved, such as the search results page. As such, these features are made to trigger asynchronous calls to the backend such that they can execute concurrently with other features.

The user interface was built using a combination of HTML, CSS, Bootstrap, the Django template language (a language used by Django web applications to dynamically load data retrieved from the backend), and Vue. Vue was integrated into the frontend development as it was already planned to be utilized for the new CiteSeer system. As such, having an implementation of the UI in Vue would allow for reusability for when the existing features need to be implemented for future projects. Vue, like other modern JavaScript frameworks, is

incredibly efficient for building frontend codebases due to its reusable component-based structure and better code readability compared to vanilla JavaScript or jQuery. Because the architecture of the codebase was already created and deployed, Vue was integrated as a CDN to avoid investing too many resources in setting up a Vue loader.

[UML diagram for application flow]

When a user submits a query into the search bar, they are directed to the search results page. The query that the user is stored as a URL query parameter. When the search results page is mounted into the browser, a lifecycle event function is triggered which retrieves the query from the URL and makes an asynchronous call to an API endpoint to retrieve the documents from the Elasticsearch index with that query string as a parameter. The search API removes all punctuation and stopwords from the query and makes a query across the paper's body text, abstract, and title. The score retrieved from querying the abstract is multiplied by 2 and the score from the title is multiplied by 3, putting more emphasis on papers that contain query tokens in those fields. Pagination was added to limit the number of results that are returned to a digestible level. Whenever a user navigates to a different page, the page number is also added as a URL query parameter and is passed into the search API again to retrieve the list of documents for that page.

After a search query is finished, aggregations are collected for the journal, source, author, and year metadata fields to display the most frequent items in each of those fields. These aggregations are used for faceted searching, which they will display in a container on the search results page in a checkbox list. A user can click on a checkbox for each field they want to narrow down their results for that query. The filtering works through a listener function that triggers whenever any checkbox changes state. The function stores each of the

items to be filtered in a hash map where the key is the field to filter by and the value is a list of items to filter within that field. This hash map and query string are passed as a parameter to the filtering API, which filters down the results for a given query string using a Boolean match to only include documents that contain these fields. The filtering is the intersection of these results for these facets, such that every result should contain every single selected facet. In addition to the facet container in the UI, a list of authors will also appear for each search result item that the user can click on to add that author to the filter list and retrigger the filtering API. There is also a list of keyphrases extracted for each paper, which can also be clicked to filter by them as well. The keyphrases provide a very high-level description of an information-dense paper, extracted using a keyphrase extraction model known as citation-enhanced keyphrase extraction (CeKE). CeKE uses a variety of different metrics to retrieve the keyphrases, such as tf-idf, parts-of-speech tagging, and relative position of text alongside text surrounding citations of the paper. From there, a ranking is determined and the top 10 key phrases from those rankings are indexed for each document.

Table 6. Examples of COVIDSeer extracted keyphrases

Paper Title	**Top 10 Keyphrases**
Evolution and variation of 2019-novel coronavirus	nCoV, nucleotide, coronavirus, nucleotide substitution, outbreak, substitution rate, nucleotide substitution rate, phylogenetic trees, amino acid, amino
Self-assembly of Severe Acute Respiratory Syndrome Coronavirus Membrane Protein	Membrane, Protein, Coronavirus, VLP, SARSCoV, Syndrome Coronavirus, Coronavirus Membrane, Golgi, Respiratory Syndrome, Selfassembly
Respiratory viral infections in institutions from late stage of the first and second waves of pandemic influenza A (H1N1) 2009, Ontario, Canada	outbreak, LTCF, HN, influenza, pdm, late stage, viral infections, Ontario, Respiratory viral infections, wave

Biogenesis and Dynamics of the Coronavirus Replicative Structures	viruses, viral RNA synthesis, vesicle, synthesis, infection, Structures, Replicative, RNA viruses, Replicative Structures, RNA
Infection with human coronavirus NL63 enhances streptococcal adherence to epithelial cells	adherence, coronavirus NL, human coronavirus, coronavirus, bacterial pathogens, human coronavirus NL, pathogens, Infection, HCoVNL, NL

When a user clicks on the paper title of a search results item, they are navigated into the document view page for that paper. The document view page ordinarily contains the unique ID of the document as a part of the URL path. When the page is requested, the document ID is extracted from the URL in the backend and an Elasticsearch query is made to retrieve the metadata given that document's ID. This information is then passed to the frontend for it to be displayed to the user.

When the page is initially mounted into the browser, an asynchronous function call is made to another API endpoint to retrieve the list of similar documents to a paper given its document ID for recommendations of other indexed papers. The list of similar documents is retrieved by first obtaining an initial list of top 10 candidate papers to be used as recommendations using a tf-idf representation and cosine similarity with the abstracts and titles. These candidate papers are then ranked with each other using SciBERT, a tool for vectorizing scientific text, to generate vectors of the abstracts and titles and comparing their cosine similarity with the initial tf-idf similarity from the first step.

Table 7. Examples of COVIDSeer Similar Paper Rankings

Paper Title	**Top 5 Similar Papers**
Middle Eastern Respiratory Syndrome Coronavirus (MERS-CoV)	1. Middle East Respiratory Syndrome coronavirus (MERS CoV): Update 2013

	2. A novel coronavirus capable of lethal human infections: an emerging picture 3. Structure, Immunopathogenesis and Vaccines Against SARS Coronavirus 4. Hantaviruses in the Americas and Their Role as Emerging Pathogens 5. Zika fever and congenital Zika syndrome: An unexpected emerging arboviral disease
Coronavirus Receptors	1. Crystal structure of murine sCEACAM1a[1,4]: a coronavirus receptor in the CEA family 2. The nucleocapsid protein of the SARS coronavirus is capable of self-association through a C-terminal 209 amino acid interaction domain 3. C-terminal domain of the MERS coronavirus M protein contains a trans-Golgi network localization signal 4. Structural and functional analysis of the S proteins of two human coronavirus OC43 strains adapted to growth in different cells 5. Species-specific evolution of immune receptor tyrosine based activation motif-containing CEACAM1-related immune receptors in the dog

Personal Contributions

Principal personal contributions to COVIDSeer were with ownership of the web application development and deployment process. Initially, COVIDSeer was built off of a rudimentary Django application template that simply contained the functionalities for search and a search results page. The majority of the upfront development time was dedicated to refactoring the existing code to be more maintainable, utilize better code practices, and provide a better user interface experience. Personal contributions to the development of the application included a strong portion of this refactoring effort especially on the frontend side, integration of faceted searching, the document view page, pagination, and implementing Vue as a CDN to improve page load times and remove development overhead. When development of the system was ready for production use, the web application was deployed utilizing Apache. Afterward, the web application was configured to provide separate settings for production and development as means to make deployment as easy and rapid as possible.

In addition to the technical scope of the project, a modern and sleek user interface was designed to enhance the user experience in a way unique from the original CiteSeerX website while taking inspiration from similar specialty search engines. Given the limited scope and deadline of this project, the user interface was designed to be simple to implement yet professional and intuitive. Also, COVIDSeer's user interface was made to be responsive to provide an equally effective experience for smaller screen sizes and mobile devices.

Design Challenges

The most significant challenge with the development of COVIDSeer was the urgency to deploy a working system to aid the ongoing COVIDSeer effort. An initial timeline was to deploy the system within a week. As such, strong project management and collaboration efforts were required to effectively meet this goal. The application was initially deployed as a minimum value product, providing the basic search engine functionalities for querying across the corpus and providing a document view page. After its release, iterative updates were made to improve the system and provide more useful features such as faceted searching, publisher links, key phrase extractions, and paper recommendation.

Another significant hurdle with this project was that most of the web application had to be built from scratch. While there was a Django project template that had the pages and backend APIs already in place for querying, it was very limited compared to this project's requirements and as such was drastically modified until the system met all specifications. In contrast, other information retrieval systems made for the CORD-19 dataset were built using pre-existing web application project files that were slightly modified to use a different index and some specific features to make it more useful for the specific subject.

Comparison with Similar Search Engines

Given the constraints previously listed, there are many similar search engines built off the CORD-19 dataset that provided more expansive features than what COVIDSeer provides from a web application user experience perspective. Neural Covidex (covidex.ai) allows users to make questions as queries, in which useful and relevant questions appear as

autocomplete fields in the search bar to suggest for the user. Examples of the questions include “What is the incubation period of COVID-19?” and “What is the effectiveness of chloroquine for COVID-19?”. Vespa’s CORD-19 Search incorporates its own query language. For example, +chloroquine +(covid-19 coronavirus) is used to search for documents that contain “chloroquine” and “covid-19” OR “coronavirus”. From a user experience, perspective, CORD-19 Search allows for searching between the title and abstract of a document, or through all metadata fields. There are different options for page ranking, including Vespa’s own page ranking algorithm or sorting by date.

The biggest strength of COVIDSeer is that it expands upon the existing CORD-19 dataset to be more useful. The PDFs for the CORD-19 dataset were retrieved using the COVID-19 fatcat dataset released by Internet Archive. The PDFs were then extracted for their metadata, including figures, tables, abstracts, and keyphrases. In essence, COVIDSeer as a web application serves primarily as a tool for the user to interact with this rich and enhanced version of the CORD-19 dataset, and that while the functional specifications the system itself still has great implications.

Chapter 5

Implementation and Design of New System

The new prototype system relies on the work of the systems previously listed and serves to address as a modern solution for information retrieval for academic publications. The main goals for the new system are to provide better code maintainability, ease of deployment, an enhanced and more effective user experience, and a more scalable ingestion

system for papers. That being said, it should still carry over many of the existing useful features of the current system but improve them in such a way that they can provide better functionalities or are more intuitive to use. At the time of this writing, the new system is currently still in development however it is fully intended that this will be deployed into production in place of the current system. That being said, a strong foundation has been laid out for future development for the system such that new features can be rapidly built and tested without having to invest in development overhead in researching and implementing new technology stacks.

Features

The new system integrates all the basic required functionalities of a search engine, namely querying for papers, displaying the rankings on a search results page, and having individualized pages that display the metadata for a given paper. Document view pages also contain citation links that display a list of indexed documents that cite a given paper. That being said, there are new features that serve to build upon the current system and provide a more enhanced user experience.

As previously mentioned, one of the primary motivations for the development of COVIDSeer was to extend the implementation of features over into the new system. The primary feature that was carried over is faceted searching. On the search results page, a user is presented with a card in which they may narrow the results down either by year, through the use of a range slider, or through a list of checkboxes that contain the top 10 aggregations for a specific facet. For instance, a list of checkboxes for the top 10 authors who appear in the

results for a given query can be interacted with by the user such that only authors who have been toggled will appear in the results.

A completely new functionality is autocompletion. As a user inputs a character to formulate their query in the search bar, they will automatically be presented with a dropdown that provides a list of papers that match their query by title. This dropdown acts as a list of suggestions such that the user will easily see papers that may be very relevant to what they are looking for without having to navigate through a list of search results. Clicking on one of these suggestions will route the user to the document view page for that paper. As the user makes changes to their query, they will be provided with a new list of suggestions based on the new query they have inputted.

Table 8. Example of autocompletion in the new system

Query	**Autocompletion Suggestions**
com	1. COM-Lex Syntax 2. COM-MIT at SemEval-2016 task 5: Sentiment analysis with rhetorical structure theory 3. COM-PENDIUM: a text summarisation tool for generating summaries of multiple purposes, domains, and genres 4. Com piling and Using Finite-State Syntactic Rules 5. Com-mandTalk: A Spoken-Language Interface for Battle eld Simulations
comp	1. Compact Bilinear Pooling

	2. Compact Data Structures for Querying XML 3. Compact Isolated Word Recognition System for Large Vocabulary 4. Compact and Interpretable Dialogue State Representation with Genetic Sparse Distributed Memory 5. Compact and Robust Models for Japanese-English Character-level Machine Translation
computer	1. Computer adaptive practice of maths ability using a new item response model for on-the-fly ability and difficulty estimation 2. Computer aided correction and extension of a syntactic wide-coverage lexicon 3. Computer aided correction and extension of a syntactic wide-coverage lexicon *(Note: This is a duplicated paper in the index)* 4. Computer aided translation and the Arabic language, First Arab school on science and technology 5. Computer and Cognitive Science MCCS-91-206

Similar to COVIDSeer, the document view page displays a list of papers that are similar. In addition to displaying the citations for a paper, this list of similar papers will serve as useful recommendations for the user if they wish to investigate into the subject even further.

One feature that has been extended from the original system but enhanced drastically is a personal content portal and account management system. A user can register an account by inputting a form with their email address, password, and other personal information such as their name and affiliation. Afterward, an account activation link will be sent to their email in which the user will need to navigate to the link to authorize their account and login. When a user is logged in, they may create collections and add papers to them to group desired papers. They may also send a request to update document metadata should any field be incorrect. Users may also request a password reset link sent to the email associated with their account should they forget their login information.

The user interface has also been massively overhauled as well. As the client is decoupled from the backend API, the user interface will render in the user's browser while data is being retrieved asynchronously. This provides a much cleaner user experience, as the user will receive faster page load times and visual indicators that show that data is being retrieved. If one of the APIs used in a page does not work for any specific reason, such as displaying the aggregations for a search result, the user will still be able to use the rest of the features available on the page without the entire user interface crashing. Mobile responsiveness is also made a higher priority and as such, the frontend will be adjusted based on the screen size.

Implementation

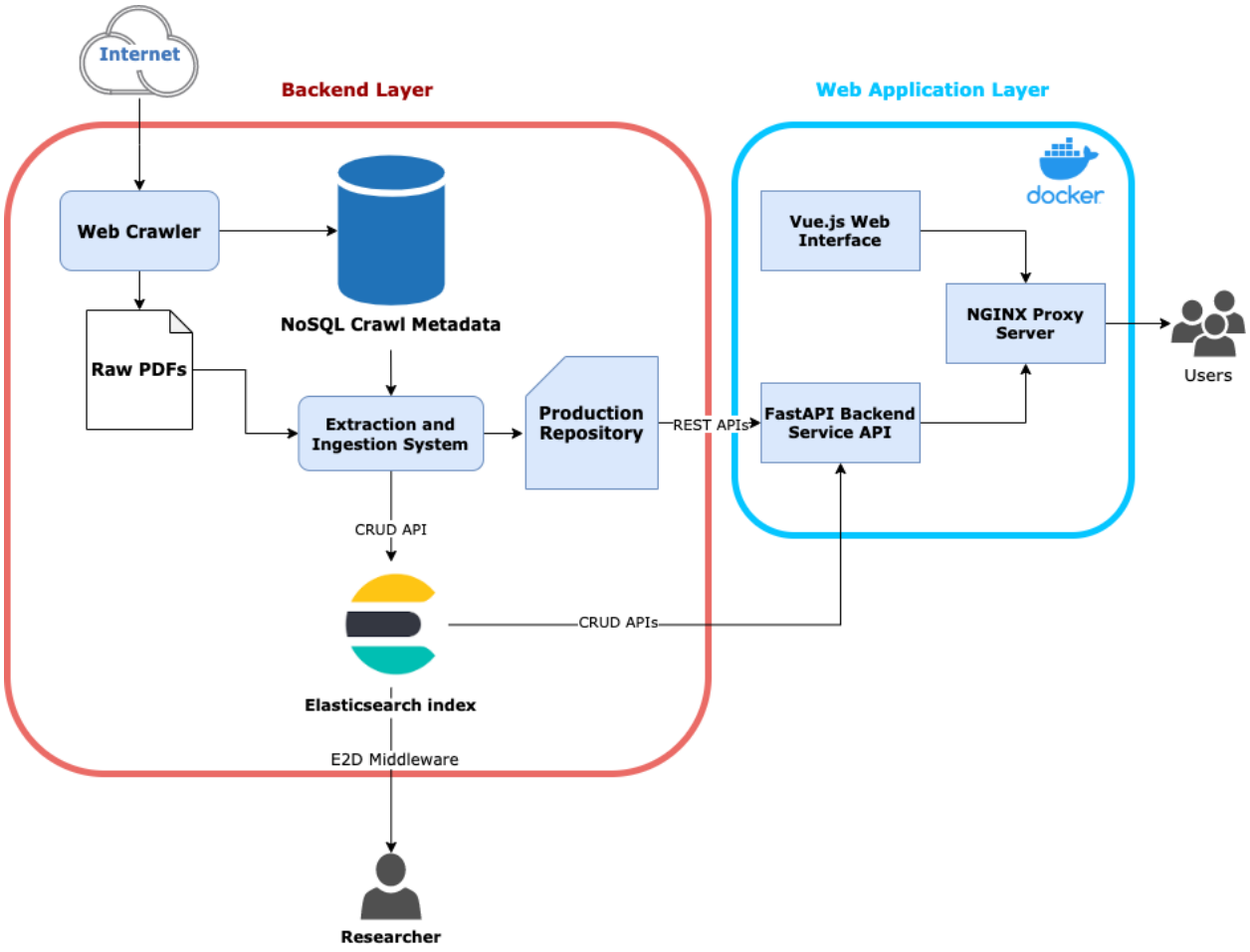

Figure 3. Architecture for new system

The new system utilizes Elasticsearch as the primary ingestion and search framework. Elasticsearch has high-performance scalability due to its design utilizing all available cores. As such, ingestion can be performed on multiple threads. Elasticsearch is not the fastest NoSQL database for indexing documents, as Apache Solr and MongoDB are faster. However, Elasticsearch is still utilized for its very fast search times, horizontal scalability, thorough documentation, and integration with visualization and log analysis tools like Kibana and Logstash.

The web application technology stack utilizes FastAPI for the backend. FastAPI is a Python library for developing REST APIs with high-speed performance times. The Python backend also allows for integration with libraries that easily integrate with Elasticsearch, such as elasticsearch-dsl. The use of a custom-built backend API allows for more customizability for the application's functionality and focus on the separation of responsibilities and abstraction for the frontend and backend as opposed to having the frontend interact with the Elasticsearch index directly. The frontend is built using Vue.js through the Nuxt.js framework to serve the web files. Vuetify is used as the UI library because of its robust documentation, popularity, a large number of powerful and customizable out-of-the-box components, and adherence to Google's Material Design Principles. Vuetify allows for rapid development of the user interface to prioritize integration between the client and backend APIs without spending development time on designing component logic and layout.

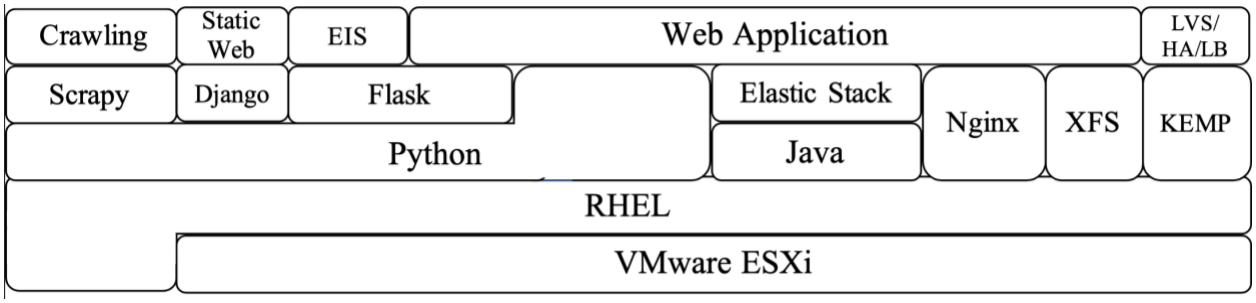

Figure 4. New system software stack

To take full advantage of the decoupled web architecture and to address the issue of multiple project dependencies for both the backend and frontend, the entire web application is containerized using Docker Compose. Docker removes significant overhead in resolving dependencies and configuring multiple environments for a project by containerizing services into virtual environments that interact with each other. As long as the machine that is hosting

the web application have Docker installed, the entire setup process for the application can easily run automatically. Not only does this make deployment much faster and easier, but it also allows makes the system portable as well. The backend and frontend services are separated into different containers. A publicly accessible NGINX proxy server is also configured in its own container, which intercepts requests and directs them to either the backend or frontend service depending on the URL pattern. As the frontend and backend services are exposed internally on separate ports, the proxy server allows for a single port on the machine to be made public to access the entire application.

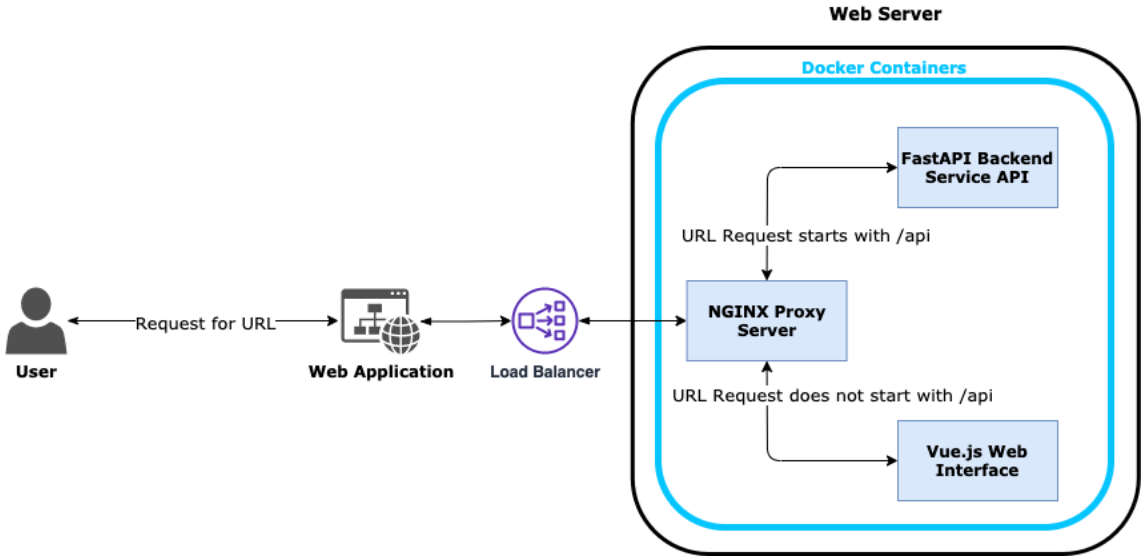

Figure 5. URL parsing schema for new system

Faceted searching is implemented in two steps, similarly to COVIDSeer. First, when a user makes a search query, an API to retrieve the aggregations given the query string is called concurrently with the search API. This list of aggregations is then displayed on the search results page. When a user makes a filter based on these aggregations, the facets they are trying to filter for are grouped and are passed into the search API which is then called

again without leaving the page. The new filtered results will then display in the search results.

Autocomplete is implemented through a client-side function that listens for any change in the text input within the search bar. This function then calls an autocomplete API on the backend, which utilizes the Elasticsearch Suggesters API. The Elasticsearch Suggesters API is a powerful tool for real-time autocompletion, which gives a list of potential results the user may be querying for based on an incomplete search query and a field to search across for.

User authentication is implemented in a series of stages. When a user submits a request to register an account through an encrypted API, a query will be made across the Elasticsearch index to see if a user with the inputted email already exists. If not, then the user information will be stored in the Elasticsearch index. A field for if the user has verified their account will also be stored alongside a hash for the user's password. A link to the user will then be sent to their email to verify their account. The URL for account activation contains a unique token for the user. When the user clicks on this link, the client sends an API to the backend containing this token which is then interpreted and verifies the user if the token is valid. If so, the user will be marked as verified in the Elasticsearch index and they will be able to log in. The password reset process follows a similar process to account activation, in which a link containing a unique token for the user will be emailed to the user. When the user clicks on the link and enters a new password, the client will pass along the encrypted password alongside the token. The backend API then validates the token, and if successful the user's password will be changed. When a user is successfully logged in, a unique token will be passed to the client. This token is stored in the user's browser, which will then be passed along to every API call that requires authentication such as accessing the user's profile page.

As an additional security feature, Google reCAPTCHA is implemented on the user authentication pages. reCAPTCHA monitors for any suspicious behaviors on the client-side, such as if the user is a bot. A reCAPTCHA site key is stored on the client-side and a reCAPTCHA secret key is stored on the server. When a user attempts to either create an account or log in, the client will generate a token using the site key and pass it into an API on the server that validates the token using the corresponding secret key. The API will then determine if the token is received from an authorized client and that the user is likely to be human. The API passes a response to the client that determines if the reCAPTCHA was successful or not, in which the client then determines if the user's login or account information should be submitted for authentication. This allows the system to ensure that the user attempting to use the service is a human and not a malicious robot. The reCAPTCHA tokens are valid for an extremely short period and expire after one use, so they cannot be intercepted and reused multiple times. reCAPTCHA v3 is utilized for this system, which invisibly tracks the user's behavior to determine if they are human as opposed to prompting the user with a test (for instance, clicking on images that contain a streetlight) whenever the user attempts to sign in.

Personal Contributions

Much like with COVIDSeer, the majority of the personal contributions for the new system were in the development of the web application. This included leadership in the frontend development process, designing the user experience and application flow for the user, designing the user interface, developing reusable components, and determining how components will be

nested and communicate with each other, designing the API contracts, transitioning the frontend to utilize Vuetify and development for faceted searching, and necessary client-side functionalities for user authentication.

Aside from development, additional contributions were in the deployment process and containerization of the entire application through Docker. This included designing the web architecture, transitioning the backend and frontend codebases for Docker, forming the configuration files for the entire project to utilize Docker Compose, setting up the proxy server, and setting up different environmental variable files and configurations based on whether the system will be deployed on production or for testing.

Design Challenges

One of the initial challenges with the development of the new system was in the ramp-up process to learning all the necessary languages and frameworks. To speed up this effort and to ensure that the development of the system can begin as soon as possible, Python was chosen as the primary language for the backend API as there was already a strong background with web development using Python. Vue.js was also chosen as the frontend framework because of its relatively approachable learning curve, strong documentation, and other frameworks such as React were considered too complex for the scope of this project.

While designing the frontend stack, a major challenge that was posed was whether to utilize server-side rendering (SSR) or client-side rendering (CSR). A major challenge in web development using modern JavaScript frameworks is that the frontend code needs to be rendered into HTML code for the browser to load. SSR and CSR are two approaches in accomplishing

this. Both approaches have their strengths and weaknesses. SSR is where the client-side is rendered in the server. SSR is useful for offloading the rendering process from the user such that they do not need to rely on their processing power before they can access the page. SSR is also regarded as better for search engine optimization (SEO) than CSR, making links in the new system more likely to appear in the results pages for web search engines like Google. However, SSR can cause an increase in the amount of time between a user navigating to a page and the first stream of content to arrive. CSR requires the user's browser to render the JavaScript into HTML. As such, all routing, templating, and data fetching are handled by the client. However, CSR tends to be worse for SEO than SSR and will inefficiently scale in performance as the size of the JavaScript code begins to grow. Nuxt.js leverages the advantages from both methodologies, as static pages will be pre-rendered and loaded to prevent unnecessary processing time while still providing a dynamic and efficient user experience with good SEO. As such, Nuxt.js was chosen as the primary framework for the client.

Another challenge in developing the frontend was in the selection of a UI library. Bootstrap's Vue library was initially used as the primary UI library due to Bootstrap's immense popularity in building user interfaces. However, Bootstrap Vue proved to be very limited in many features that were required for the new system, such as having a dropdown for text fields to display autocomplete results. Utilizing Bootstrap would require building many of these components from scratch. Vuetify proved to be more useful in terms of available components to work with alongside being more popular for Vue projects. As such, the entire user interface was converted into using Vuetify from Bootstrap. This allowed for further development in the frontend to be more efficient going forward.

The backend API also faced some revisions in the technology stack. Instead of FastAPI, the backend was to utilize Django through the Django REST Framework. Django is great for abstracting away a large amount of datastore management such that a developer does not need to worry about writing raw queries to make CRUD methods instead of using built-in libraries. However, Django's model structure was designed for using a SQL database. Because the primary data store for the new system is with Elasticsearch, the strong advantages to using Django were either not relevant or made redundant as external libraries would have to be used to interface with the NoSQL Elasticsearch index. Based on these considerations, the backend was transitioned into using the FastAPI framework instead.

One of the more prevalent challenges with building the new system was in considerations of how to deploy it. Because the backend and frontend services are operating on different ports, a user would somehow need access to both of these ports to reach the web application. Opening multiple ports onto production servers was not recommended as it will become increasingly difficult to manage. The proxy server was set up such that it could serve as a single endpoint for the user to interact with the backend and frontend services through URL patterns. However, there was a major concern in potential abuse of the backend API if it were to be publicly accessible outside of the web interface. Due to the design of Nuxt.js making external API calls through the client, it was not possible to make it so that the backend APIs could only be accessed internally through the server. The solution to this was to require a token to be passed along for every single API call. A valid one-time use token could only be generated through using the web interface, which is then expected and validated with every API call. This limits the use of the system to exclusively the web interface, preventing malicious users from abusing the API.

Future Work

As a majority of the development effort for this system at this moment was in designing the foundation for the frontend and backend services, the eventual goal is that new features could be implemented much more easily. Several features are hoping to be implemented for future iterations of the new system, leading up to and after deployment on production.

One of the primary features is integration with MathDeck, a tool developed and maintained by the Rochester Institute of Technology which allows for mathematical formulas and equations to be queried. These formulas and equations would be extracted and interpreted alongside other document metadata. This would allow users to query for papers that contain a specific formula that they might be looking for.

In terms of the user application flow, it is also intended for tables and figures extracted from papers to be included in their document view page. Additionally, each author will have a page that displays all the ingested papers published by them. Users will be able to query for specific authors alongside papers. An additional feature is if a user clicks or highlights a key term that may be ambiguous to them, a window will appear that contains a definition of the key term and additional context. This will improve the way that a user, especially those who may be curious but inexperienced in the subject that they are researching, will be able to learn new information and understand papers.

An admin console will also be integrated as a part of the user authentication process. Internal accounts will be created and designated with admin privileges. These admin accounts will have access to a console page in which they may update documents directly, remove documents from the index, and view requests to update document metadata. As CiteSeerX

currently receives many requests from authors to remove or update the papers listed in the system, this will provide a method for internal admins to efficiently address these user concerns.

Finally, in terms of the development effort, it is intended for a Jenkins pipeline to be integrated into the system. Jenkins is an open-source automation server that provides continuous integration and deployment. Jenkins would integrate with GitHub webhooks to run build tests on commits, ensuring that a majority of the codebase has unit tests in place for them and that any new changes will not break other features in the system. Jenkins can also make deployment easier. As the new system will take place on multiple web servers, Jenkins can automatically reload and redeploy the web application whenever changes are pushed into a master branch without requiring an admin to manually pull the changes and restart the service on each web server.

Chapter 6

Conclusion

This thesis serves to outline many design and engineering challenges that go into building and maintaining a web-based information retrieval system for academic papers. The initial limitations of the current system were addressed, from a user experience, scalability, and development perspective especially in the context of similar academic search engines. The development of COVIDSeer serves to highlight the interesting key aspects of building a specialty search engine from the ground up, with many features proving to be useful and extendible to the upgraded version of the current CiteSeerX system. The new system was then developed using the lessons learned from these previous challenges in mind alongside modern technology frameworks to provide the foundation for a new architecture that is more maintainable, easier to deploy, and more effective in terms of user experience and scalability. The new system will continue to undergo extensive development efforts before public deployment and eventually becoming available as an open-sourced framework.

BIBLIOGRAPHY

1. Croft, W. Bruce, et al. Search Engines: Information Retrieval in Practice. Addison-Wesley, 2010.
2. Curlie. “Computers: Internet: Searching: Search Engines.” Curlie, curlie.org/Computers/Internet/Searching/Search_Engines.
3. Docker. “Why Docker?” Docker, 2021, www.docker.com/why-docker.
4. Elastic. “Suggesters.” Elastic, 2021, www.elastic.co/guide/en/elasticsearch/reference/current/search-suggesters.html.
5. Google. “ How Search Algorithms Work.” Google, Google, 2021, www.google.com/search/howsearchworks/.
6. Google. “Introduction to Robots.txt.” Google, Google, 2021, developers.google.com/search/docs/advanced/robots/intro.
7. Greengass, Ed. Information Retrieval: A Survey. University of Maryland, 2000.
8. Miller, Jason, and Addy Osamini. “Rendering on the Web | Google Developers.” Google, Google, 26 Nov. 2019, developers.google.com/web/updates/2019/02/rendering-on-the-web.
9. Lu Wang, Lucy et al. “CORD-19: The Covid-19 Open Research Dataset.” ArXiv arXiv:2004.10706v2. 22 Apr. 2020 Preprint.
10. NuxtJS. “Nuxt.js - The Intuitive Vue Framework.” NuxtJS, 2021, nuxtjs.org/.
11. Rohatgi, Shaurya, et al. “COVIDSeer: Extending the CORD-19 Dataset.” Proceedings of the ACM
12. Symposium on Document Engineering 2020, 2020, doi:10.1145/3395027.3419597.

13. Yigal, Asaf. “Solr vs. Elasticsearch: Who's The Leading Open Source Search Engine?” Logz.io, 7 Aug. 2020, logz.io/blog/solr-vs-elasticsearch/.

Academic Vita

Jason Chhay
jasonchhay@gmail.com
jasonchhay.com

Education

Pennsylvania State University, University Park – Graduated May 2021
B.S. Computer Science
Minor in Engineering Leadership Development & Mathematics

Experience

Research Assistant May 2018 – May 2021
Pennsylvania State University University Park, PA

- Led project management for front-end development amongst graduate and undergraduate students for Elastic search engine built with Python and Vue.js.
- Developed and deployed informative website on CiteSeerX search engine through Django web framework, utilizing Python, HTML/CSS, and SQL.
- Utilized Docker to containerize decoupled web application to improve efficiency of installation and deployment on production web servers.

UX Lead Developer April 2019 – April 2021
Penn State Dance Marathon University Park, PA

- Managing 2 other developers in UX design and front-end development for internal web tools through HTML, CSS, SCSS, jQuery, Vue.js, and Bootstrap.
- Overhauled frontend for online quiz platform to utilize decoupled Webpack structure, improve user experience, and make codebase more maintainable.
- Overhauled frontend for online quiz platform to utilize decoupled structure, improve user experience, and make codebase more maintainable.

Software Development Intern May 2019 – August 2019
CommScope, Inc. Horsham, PA

- Collaborated with interns, product owners, scrum masters, and senior software engineers to develop customer technical solutions through Agile development.
- Built Java Android app that dynamically load custom user interfaces in smart TV set-top boxes for customer demos.
- Improved image memory efficiency by at least 60% through Python script that prevents extraneous files from packaging onto a digital media catalog.

Projects

QualityMix for Spotify August 2020

- Application that creates custom playlists based on artists, songs, genres, and musical qualities built with React, Express, Node.js and Spotify's Web API.

COVIDSeer March 2020

- Full-stack web developer for open-research search engine with Django, Elasticsearch, and Vue for publications relating to COVID-19.

Publications

- Rohatgi, Shaurya, et al. "COVIDSeer: Extending the CORD-19 Dataset." Proceedings of the ACM Symposium on Document Engineering 2020, 2020, doi:10.1145/3395027.3419597.

Skills

Languages: JavaScript, Python, Java, Swift, C, HTML, CSS

Frameworks: React, Vue.js, Django, SwiftUI, Node.js, Android, Express

Tools & Programs: Redux, Docker, Figma, Git source management, Linux, Microsoft Office, Adobe Photoshop, Adobe Illustrator

Relevant Courses

- Application Development
- Data Structures & Algorithms
- Operating Systems
- Artificial Intelligence
- Project Management
- Technical Writing

Awards/Achievements

- 3rd place in HackPSU Spring 2018 TLT Challenge
- 2nd place in Kohl's Business with Integrity Case Competition
- Tau Beta Pi (Engineering Honors Society)