The Pennsylvania State University

The Graduate School

THE MIGRATION OF DATA AND REFACTORING OF LARGE SCALE DIGITAL LIBRARIES: A CASE STUDY FOR CITESEERX

A Thesis in

Information Sciences and Technology

by

Sean Parsons

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

May 2020

The thesis of Sean Parsons was reviewed and approved* by the following:

C. Lee Giles

David Reese Professor of Information Sciences and Technology

Thesis Advisor

Dinghao Wu

Associate Professor of Information Sciences and Technology

Edward Glantz

Teaching Professor of Information Sciences and Technology

Mary Beth Rosson

Professor of Information Sciences and Technology

Head of Graduate Program of Information Sciences and Technology

Abstract

CiteSeer^x is one of the first academic digital libraries in the world and currently contains data on over 10 million academic documents. While the current technical architecture of CiteSeer^x has scaled well to this point, there is a need to ingest more papers and utilize modern tools to increase efficiency. NoSQL datastores are examined in this thesis as well as new ways to represent relational data in non-relational databases. Additionally, in this thesis we compare the performance between Elasticsearch and MongoDB for our dataset and we propose a new indexing system for CiteSeer^x.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my fellow lab members and leaders including Dr. Lee Giles, Dr. Jian Wu, Shaurya Rohatgi, Bharath Kandimalla, and Jason Chhay. Without their help, I would not have been able to complete this thesis. Additionally, I must thank Dr. Ed Glantz and Dr. Dinghao Wu for helping me on my academic journey. Last but not least I have to thank my friends and family including Valeria and all of the extended 507 group for being patient and motivating me to complete my work.

Dedication

This thesis is dedicated to my loving parents Walt and Kim, as well as my brother Nick.

Chapter 1

Introduction

1.1 Background

The CiteSeer project began in 1997 at Princeton University as the first digital library and search engine to provide automated citation indexing and citation linking using autonomous citation indexing [1]. Overall, the project encompasses a scientific literature digital library and search engine that allows researchers and public users to find relevant publications pertaining to their query [2]. With features including author disambiguation and reference linking, CiteSeer^{x} has become more robust over the years. [3]. The system architecture of CiteSeer^{x} has evolved over many years, and the current architecture is spread out across many different virtual machines which will be discussed later.

CiteSeer^{x} has ingested and indexed over 10 million academic papers which it

queries every time a user interacts with the search bar [3]. Much like a Google Scholar, CiteSeer^x will return relevant academic papers to the student, researcher, or academic who is searching [4]. Something that differentiates CiteSeer^x from many other academic digital libraries is that it consistently offers the ability to download the full PDF version of any individual paper [5].

CiteSeer^x is interested in migrating from a traditional XML based scheme to a metadata scheme that supports JSON capabilities in order to use software like Elasticsearch. The reasons behind this migration include increases in speed and distribution of data across multiple production machines [6]. In order to make this migration and determine if it is worth pursuing, it is important to observe what other academic digital libraries are implementing.

1.2 Other Digital Libraries

There are many other academic digital libraries similar to CiteSeer^x including Semantic Scholar, Google Scholar, and DBLP [7]. Out of the many scientific digital repositories, Semantic Scholar is one of the most similar to CiteSeer^x. For one, there is a similar layout, citation graph, and search feature [8]. Additionally, Semantic Scholar uses a JSON metadata scheme that can be leveraged by Elasticsearch to provide faster and more distributed indexing/searching capabilities [8]. As CiteSeer^x moves forward with the transition from a XML based metadata scheme to a JSON based metadata scheme, it is important to consider various

metadata standards in addition to what others have done with similar scientific digital repositories.

There are many academic digital libraries in existence today. Some of these projects include Google Scholar, DBLP, PubMed, Web of Science, and Semantic Scholar. A few of these academic digital libraries have publicly released their data format schemas, allowing other researchers in this field to see how best to store academic publication metadata. In the case of DBLP, its team has released the basic makeup of their nested XML-based data schema that stores different information pertaining to article descriptions [9]. PubMed, a leading digital library for medical related publications, also uses an XML based schema and has also released the different data fields from publications that it currently indexes today [10]. Web of Science currently uses a variant of the XML data schema, utilizing .XSD file formats to store metadata about its publication dataset [11]. While many of these academic digital libraries are using XML based systems, there is one that currently uses JSON with Elasticsearch.

SemanticScholar is an academic digital library created by the Allen Institute for Artificial Intelligence to serve as an artificial intelligence powered academic search engine [12]. SemanticScholar is able to index the metadata of over 125 million academic publications [12]. The publications themselves may be hosted on other websites such as ArXiv or PubMed. SemanticScholar also hosts a large and complex literature graph with all of the metadata that it holds. This academic digital

library utilizes a JSON based schema and openly states that it uses Elasticsearch as its indexing engine [12]. While the team from the Allen Institute for AI releases what software they use, they do not detail how to conduct a large scale data format migration as well as an indexing platform migration.

1.3 Metadata Standards

Given the substantial growth of information and data available online, there is a directed focus towards making this information valuable. By using metadata and other features, experts can draw conclusions about massive amounts of data and leverage the resource of information to answer previously unanswerable questions. Metadata is traditionally defined as "data about data"; although this may be understood by some, it is also helpful to think of metadata as "information about an object, be it physical or digital" [13]. With the heterogeneous nature of data and information across different fields, drawing conclusions or searching through information can be very difficult. This is why it is necessary for different groups to convene and form metadata standards [14]. Some of these groups include ISO, the Dublin Core Metadata Initiative, and the World Wide Web Consortium [14].

Very little academic work has been done on comparisons between different metadata standards. Some of the most common metadata schemes include the MAchine Readable Cataloging (MARC), Dublin Core, Digital Object Identifier (DOI), and Resource Description Framework (RDF) [15]. A few of these metadata

schemes are relevant to the work of the CiteSeer^x team as it relates to a migration from an older metadata schema to a new one. As an example, the DOI is relevant to CiteSeer^x as it relates to the unique identification of different scientific publications, whereas the Library of Congress metadata scheme, MARC, may not be relevant.

1.3.1 Metadata Formats

Many of these metadata standards utilize the Standard Generalized Markup Language (referred to as SGML) or, more recently, XML (Extensible Markup Language) to ensure syntax consistency across its many uses [15]. In order to index and search through these metadata, different software can be used to derive value from overwhelmingly large sets of metadata. One of the most common XML based index and search tools is Apache Solr, and it is used across many different domains [16]. While Apache Solr has traditionally been very fast, it struggles with unstructured data in JSON (Javascript Object Notation) format [16]. In the indexing and searching community, JSON is becoming the preferred data format compared to XML [17].

Many studies have compared the JSON and XML data formats for performance and resource utilization [18]. Across the board, these studies have found JSON to be significantly faster than XML in its transmission as well as parsing by different applications [18]. Seeing how JSON is a native data structure in Javascript, it is often seen as the modern de-facto data format standard for web and mobile

applications. Both JSON and XML have a focus of human readability, although the lack of tags in JSON is usually seen as an improvement in readability. An application traditionally requires more CPU utilization to parse JSON data over XML but the parsing and transmission of JSON is significantly faster than XML [18].

In relation to different metadata standards and the means by which a user can convert data from one format to another, there is limited literature for a couple of reasons. While various patents exist that specify exactly how companies like IBM and others have built XML to JSON software and vice versa, many programmers end up coding a custom conversion method to satisfy their needs [19]. This is the best solution as it allows programmers to tailor the conversion to their specific hardware capabilities. While the conversion itself, using logic in programming languages like Python, Java, or C++, is not very difficult, it becomes increasingly harder to map different tags to specific key-value pairs in JSON.

1.4 Current Architecture of CiteSeer^{x}

1.4.1 Overview

In order for CiteSeer^{x} to display results to a user after searching a collection of data, it must first build a comprehensive dataset of academic papers to index [5]. To do this, CiteSeer^{x} crawls the web, and once it finds an academic paper of

interest, it extracts all data and metadata from the paper to be stored and later indexed [5]. By following this method, CiteSeer^x has been able to steadily grow this unique dataset. Once the data has been ingested and extracted, it is stored in a database where that information is pulled and indexed, ready to be searched for by the user. On the user-most facing components of CiteSeer^x, users interact with the front end which is served using the Java Spring web framework [20]. A query is sent to the indexing system which then returns the most relevant results to the front end.

While one of the topics of this thesis is the CiteSeer^x architecture, the real focus is towards the indexing system architecture and migrating to an entirely different indexing system. To understand how the indexing part of the system fits in CiteSeer^x overall, a breakdown of each layer in the technology stack is described below.

1.4.2 Front End

The front end of CiteSeer^x is comprised of a variety of web servers and load balancers which ensure that a large amount of traffic will not overload one specific web server [2]. Apache Tomcat is the software used to serve the core CiteSeer^x Java Sever Pages (.jsp) code to the user [21]. By utilizing the Java Spring Framework, it is easy to do some core logic in Java then display the result of that logic with HTML-like formatting in a JSP file [20]. While there are currently some

changes occurring with the deployment of a new web server, the front end system has largely stayed the same.

1.4.3 Data Storage and Indexing

The CiteSeer^x data and metadata is currently stored using three different strategies across different technologies. When the initially ingested PDF file is processed, CiteSeer^x, makes sure to save the PDF and all associated data so it can be displayed to the user. This works by assigning each paper a unique CiteSeer^x document ID (DOI) following the syntax of this example here: 10.1.1.4.102. By storing the original PDF with additionally extracted files like a .txt file containing the paper full text and a .xml file containing metadata, no data is lost after extraction [5]. These files are located in a regular Linux directory structure where each number in the DOI of the paper, delimited by periods, represents another child directory to the paper and its data. After opening the directory named "10" on the CiteSeer^x file repository server, then opening "1", "1", "4", and "102", one would find all of the files associated with this specific paper with ID 10.1.1.4.102. This is how CiteSeer^x is able to store the PDF of the original paper and the full text of that paper.

The next layer of the data stack is the MySQL database which is responsible for storing all the metadata about papers, authors, clusters, and the related citation graph [22]. For production purposes, there are 2 main databases which

contain many tables with all the relational data needed to build the index and to provide the user with relevant information. In the citeseerx database, all metadata on papers, authors, citations, acknowledgements and more is stored. In the papers table stored here, there are more than 10 million rows, meaning more than 10 million unique papers. In the other core database, named csx_citegraph, lives all the citation graph information about various clusters. The concept of a cluster is an important one because it defines one of the core objects or nodes on the citation graph itself. A cluster as it is used in CiteSeer^{x}, is a collection of similar authors grouped by their papers citing one another. In the citation table of the csx_citegraph database, which represents all citations stored in the CiteSeer^{x} system, there are over 207 million rows. There are two main database servers that are synced regularly to ensure consistency and backup capabilities, if needed.

The last part of the system which directly processes the CiteSeer^{x} data also happens to be the focus of this thesis: the indexing platform. CiteSeer^{x} uses Apache Solr to index, or process and store data about an object so it can be easily search-able, information about authors and papers alike [16]. Apache Solr is built on top of Apache Lucene, an open source search engine software from 1999 [16]. Apache Solr brought new features like full text search, real time indexing, and dynamic clustering for scaling [16]. Additionally, Apache Solr uses XML as its metadata format, so it is also classified as a NoSQL datastore. Apache Solr is written in Java and it has extensive Java support in the library Solrj. For this

reason and many others, CiteSeer^x adopted it many years ago as its main indexing tool. There are 2 main index servers which are currently running Apache Solr today, serving metadata about papers and authors to the users when they search on CiteSeer^x.

1.4.4 Data Ingestion

CiteSeer^x has a multi-step process for finding and extracting information that is found on the public internet. First, the crawling system must be fed a list of URLs to visit and once it navigates to a website, it reads the robots.txt file that is commonly offered by websites to explicitly state what is allowed to be crawled and extracted. For instance, to see what Google allows in terms of crawling and extracting, we can navigate to www.google.com/robots.txt in a web browser and see all the different crawling policies listed. Once there is a list of websites that allows crawling and the URLs are known, the crawler goes and downloads the according PDFs that it finds [5]. Then, it is time for the extracting software to take the PDF as an input and generate the output of many metadata related files for the ingestion process to proceed. Currently, CiteSeer^x uses PDFMEF to extract information from the crawled PDF files [23]. This tool combines the functionality of other tools like Grobid and PDFBox [23]. The output of PDFMEF is many different metadata files including the full text .txt file and the metadata .xml file. Additionally, PDFMEF has the ability to directly load the relevant metadata to

the MySQL databases previously mentioned.

1.5 Goals and Approach

An overarching goal of this thesis is to find areas of CiteSeer^x which can be improved or modernized using current technologies. Another high level goal is to make the indexing system as efficient as possible to allow for the indexing of many more papers. By building a system that utilizes modern technology and testing various systems for efficiency, we hope to accomplish this. As can be seen in the previous section of this thesis about system architecture, there are many separate systems which culminate into what CiteSeer^x is today. By using containerization technologies like Docker containers, we hope to make our experiments and systems more replicable than ever before [24].

1.5.1 Experiments on NoSQL Datastores

After seeing that JSON based systems perform better than XML based ones, the team at CiteSeer^x is looking to conduct experiments to find the best new index system for CiteSeer^x. By comparing the performance metrics of Elasticsearch and MongoDB, this thesis will provide the information the team needs to decide upon the new indexing system.

1.5.2 Generate New Schema

Once a new index system is selected, a new JSON data schema must be created. Additionally, index architecture will be discussed as it pertains to running an index in a production environment. Creating a new schema for the index system is important because field mappings are what an index system is built on top of, and native data structures are one of the reasons JSON schemas are so powerful.

1.5.3 Data Migration

After the CiteSeer^x lab group decides on the new index system, the process of migrating the data must be dealt with accordingly. While many JDBC connectors exist to simply migrate the data over to NoSQL datastores from a system like MySQL, formatting becomes an issue. One of the main goals of this thesis was to find the best way to migrate the CiteSeer^x dataset over to Elasticsearch so it can be used as the primary index system.

1.5.4 Refactoring or Rebuilding Front End

As the data migration process was happening, the CiteSeer^x team was faced with a difficult decision: is it best to refactor the current front end to make calls to ElasticSearch instead of MySQL and Solr or create a new front end altogether? After much discussion, it was decided that there would be a new front end created for the query results from Elasticsearch. More on this topic can be found in the

Future Work chapter of this thesis.

1.5.5 Approach

The approach of this thesis is to improve CiteSeer^{x} by configuring Elasticsearch to be used as the primary indexing system and to migrate all of the data to Elasticsearch. Additionally, by introducing containerization to CiteSeer^{x}, we can build a more scalable microservice architecture in the new Next Generation CiteSeer.

Chapter 2

Experiments and Comparison of NoSQL Databases

2.1 Introduction

Once the CiteSeer^x lab group decided to migrate to another NoSQL datastore instead of MySQL and Apache Solr, there were a few popular options to choose from. The two systems which were considered the most were Elasticsearch and MongoBD. While both of these systems use a JSON based key-value pair schema, they can act very differently in large production systems [25]. Graph databases like Neo4j were not considered in this study because they do not provide production search functionality and would require a different formatted testing schema. In order to decide which system to use for the indexing and metadata storage for

CiteSeer^x, experiments were conducted, and both of the systems were evaluated. This chapter begins with general descriptions of the two systems and then describes the various experiments.

2.1.1 Elasticsearch

Elasticsearch is an open source search engine built on top of Apache Lucene which allows developers and website owners to utilize the powers of search [17]. The product itself allows for a highly distributed, full-text search engine that comes pre-built with a REST API for easy data manipulation [17]. While Elasticsearch may be thought of as a pseudo-database, it is really a distributed JSON document store. For system administrators and database administrators, managing millions of records and having access to that data in real time is a hard feat. Leveraging the open source nature of Elasticsearch, teams from all around the world at organizations like Uber, StackOverflow, Shopify, CodeAcademy, SoundCloud, and Expedia all deploy Elasticsearch for their searching needs [26].

Elasticsearch is written primarily in Java, and it runs as a daemon service on production or development servers. It can be accessed in a variety of different ways because of the Elasticsearch API, which is a RESTful API capable of handling complicated requests and queries [27]. For this experiment, the team used the Python Elasticsearch library, which is a Python wrapper for the REST API, to conduct all requests [28].

2.1.2 MongoDB

MongoDB is currently the most popular NoSQL database on the internet [29]. It is a general purpose, document-based, object datastore that operates on JSON formatted data. A MongoDB instance can be expanded by introducing sharding and clustering on the data. Additionally, MongoDB can store actual files like images or videos by serializing them first. Compared with a traditional SQL database, MongoDB scales exceptionally well with unstructured data and queries that require multiple join operations.

MongoDB is written in many languages including C++ and Javascript and similar to Elasticsearch, it has client APIs for almost all programming languages. In this experiment, the team will use the Python MongoDB library which acts as a client to the MongoDB server daemon service [30].

2.2 Experiment and Evaluation

2.2.1 Experiment Design

The goal of this experiment is to comparatively determine which NoSQL database would suit the needs of CiteSeer^{x} moving forward. By studying the CPU usage and memory usage during indexing, as well as total indexing speeds, the team hopes to to arrive at a conclusion as to which system is faster and more efficient for the 10 million papers in the CiteSeer^{x} dataset. The team is focused on picking

Operating System	CPU	Memory
CentOS Linux 7	Intel Xeon Gold 5118 2.30GHz 8 cores	16GB

Table 2.1: Virtual machine hardware specifications.

the system with the most efficient migration process possible. Additionally, only default configurations will be used in this study to compare Elasticsearch and MongoDB out of the box.

To accomplish this comparative study, 1 million papers will be indexed by both Elasticsearch and MongoDB, and their performance utilization and times will be compared. The 1 million papers would be the same papers for each system, so that there would be no discrepancies in the data. Additionally, the indexing monitoring and analysis had to be done on the same virtual machine to ensure hardware consistency during the timed trial. The hardware of the virtual machine is seen in Table 2.1.

While the intention of the experiment was to observe the indexing measures as the two systems index 1 million papers, the indexing on MongoDB slowed to a halt. Only after numerous trials were the researchers able to collect data on the MongoDB system indexing only 200,000 papers because otherwise the script would run weeks with little progress. More on this will be said in the Evaluations and Conclusions part of this thesis, but it must be mentioned that the sample size of the MongoDB indexing experiments was significantly smaller than Elasticsearch based out of necessity.

Another important note to make is about how the data is directly inserted into both Elasticsearch and MongoDB. The official Python libraries were used to automate the indexing of the papers from the MySQL database into the new systems. In Elasticsearch, there are upserting capabilities which enabled the team to update and insert/append when needed with the author and cluster index all with one request. For MongoDB, there was no single-query upserting/appending capabilities so the upserting functionality had to be separated into two queries: one checking if the document exists and the other appending it or inserting it.

While MongoDB does have the most basic upserting capabilities, it was impossible to upsert into an update for a document where a value could be appended to an array in the document. This had to be done in a completely different update statement where append operations could take place.

While the experiment below measures only the performance metrics used during the insertion of documents into Elasticsearch and MongoDB, it is advised to monitor other operations like updating, deleting, and querying documents.

2.2.2 CPU Usage

CPU utilization rates are a very important factor when working with large indexing systems and something that the research group wanted to study [31]. By leveraging the psutils Python library, it was easy to attach to a certain process running on a virtual machine to study the CPU utilization of that process [32]. The CPU

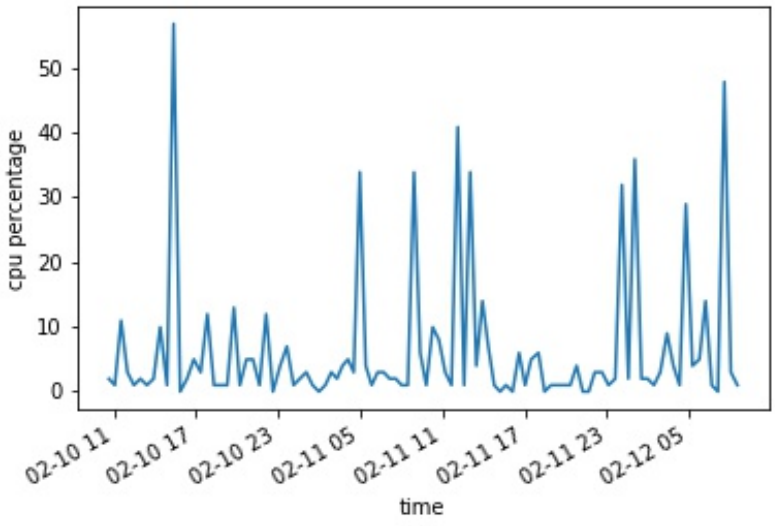

Figure 2.1: CPU utilization by Elasticsearch during indexing.

percentage returned by the psutils library is not split evenly between all cores of the system. By measuring the CPU utilization rates of both Elasticsearch and MongoDB, we can make a more informed decision on the more efficient index system to use.

While the sample size of the MongoDB indexing experiment was only 200,000 instead of 1 million, there are still very interesting results of the study. Included here is a simple graph of the CPU utilization percentage rate for Elasticsearch (Figure 2.1) then MongoDB (Figure 2.2).

An additional observation made during the indexing experiment for both these systems is that Elasticsearch uses multiple cores to speed up the inserting of new documents into the index. MongoDB allows for the use of only one core during insertion, which greatly limited the indexing speed [33]. During initial trials of

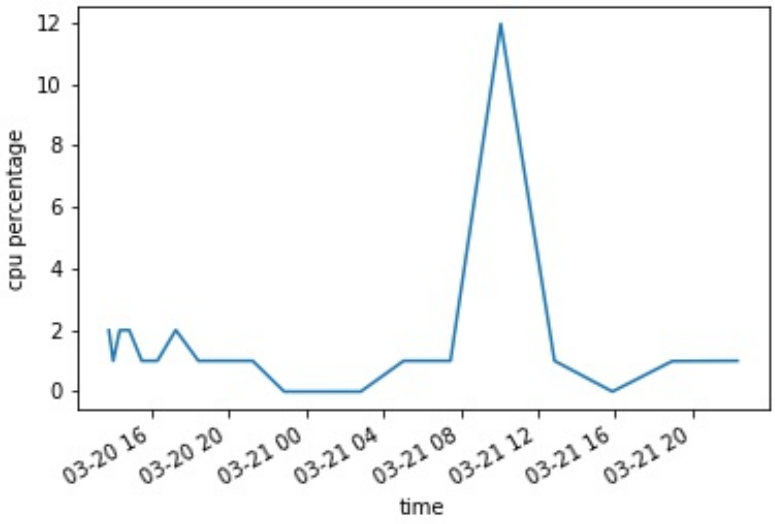

Figure 2.2: CPU utilization by MongoDB during indexing.

indexing 1 million papers in MongoDB, after only a few days the CPU utilization of MongoDB would be 99% on the one core it occupied.

2.2.3 Memory Usage

Memory utilization rates are one of the most commonly used metrics when researchers do comparisons of different indexing systems [31]. Similar to the CPU utilization tracking, the psutils Python library was used to collect the memory data from the virtual machine where these experiments took place. It is important to note that a monitor did not need to be attached to a process ID in order to collect memory utilization rates, as the rates shown are the total memory utilization rates of the machine as it was indexing the documents, not of the process itself.

Again with the memory utilization study, the MongoDB system did not have

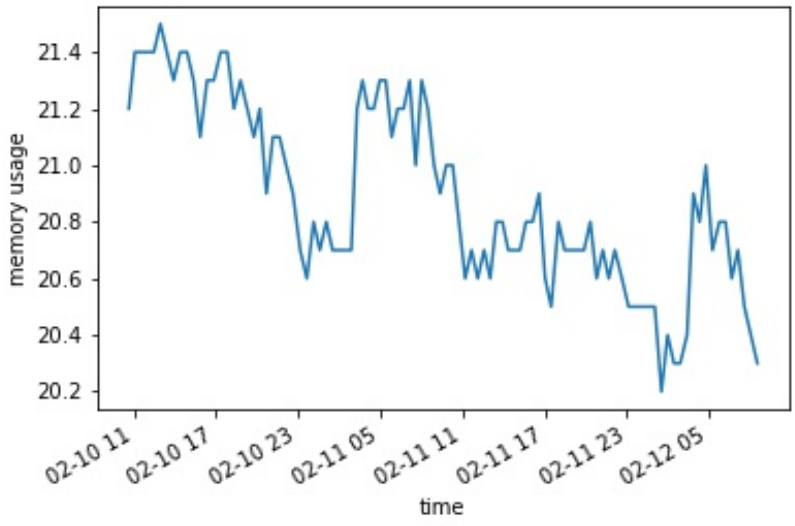

Figure 2.3: Memory utilization by Elasticsearch during indexing.

the ability to index 1 million papers and so the effects of 200,000 papers are observed. While this is not ideal, it does show insight into the scalability of MongoDB given our dataset and relational schema. Figure 2.3 below shows the total memory utilization rate of the VM during the indexing of 1 million papers into Elasticsearch while Figure 2.4 shows the memory utilization rate of the VM during the indexing of 200,000 papers into MongoDB.

2.2.4 Indexing Speed

The final metric monitored throughout the duration of this comparative study was the total time it took to insert the documents into each system. This metric was the most interesting to observe due to the figures below which reflect different time complexity curves. This metric is also the most important to CiteSeer^{x} because

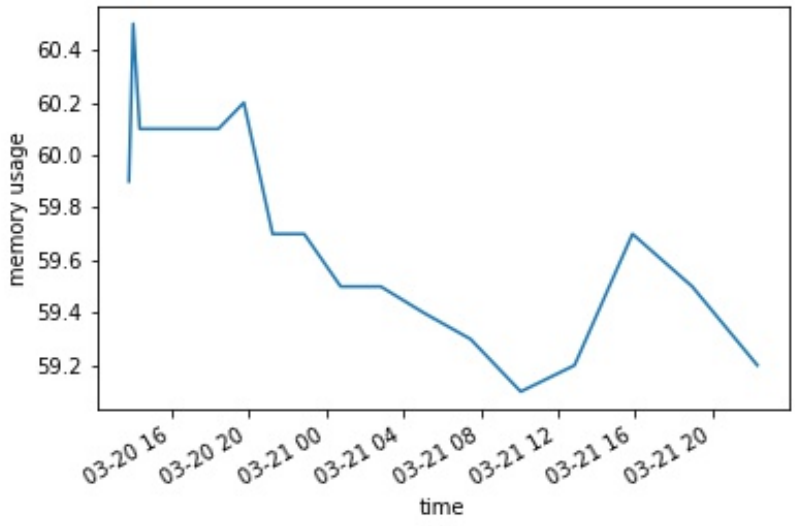

Figure 2.4: Memory utilization by MongoDB during indexing.

System	Time	Number of Documents
Elasticsearch	1 day, 21 hours, 52 minutes, and 11 seconds	1,000,000
MongoDB	1 day, 8 hours, 34 minutes, and 15 seconds	200,000

Table 2.2: Overview of indexing times across systems.

with a dataset of over 10 million papers and many more authors and clusters, migrating to a new indexing system must be a relatively quick process. Table 2.2 outlines how long it took in total to index the accompanying number of documents.

While memory utilization and CPU utilization both may effect the total time it takes to index a certain number of documents, it is important to remember the effects that the indexing code may have in the total indexing time. Since MongoDB does not have the upserting capabilities in one query like Elasticsearch, it needed two queries which likely contributed greatly to the total time it took to index papers.

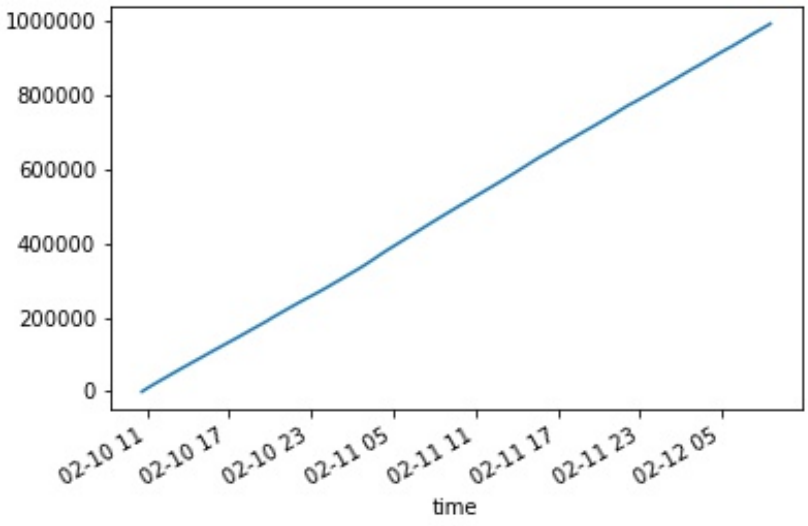

Figure 2.5: Total papers indexed across time in Elasticsearch.

System	Mean CPU %	Mean Memory %	Documents per Second
Elasticsearch	6.35	20.892	6.06
MongoDB	1.546	59.75	1.7

Table 2.3: Overview of experiment results.

Below are Figure 2.5 and Figure 2.6 which show the number of papers indexed over time for both Elasticsearch and MongoDB, respectively.

It can be observed that the indexing speed for Elasticsearch may be represented by a linear time complexity or O(n) time. Comparatively, MongoDB indexes with a quadratic or O(n^2). This is most likely reflected in the one versus two query trade-off that needed to be made in order to keep the schema the same across the two systems.

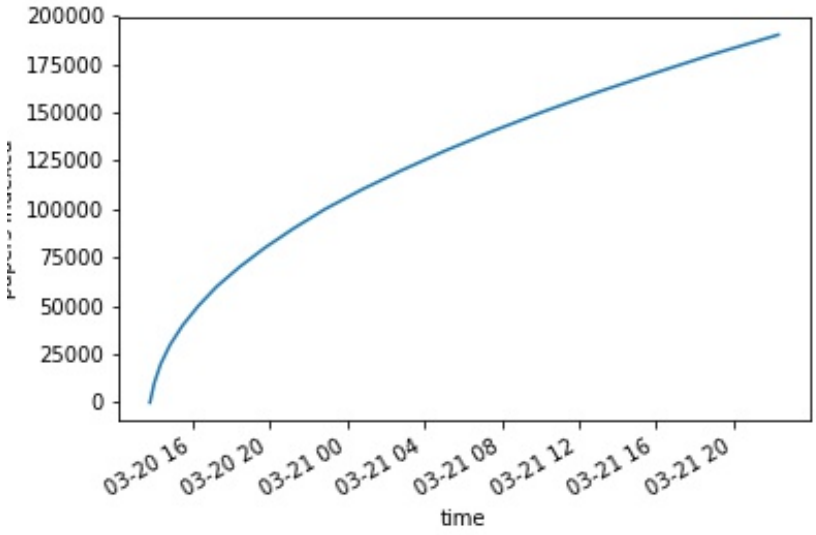

Figure 2.6: Total papers indexed across time in MongoDB.

2.2.5 Evaluation

A couple important comparisons can be made between Elasticsearch and MongoDB as it relates to their performance utilization rates and speed of indexing. Table 2.3 shows the summary data from the experiments. For one, the CPU utilization rate of Elasticsearch during indexing is about 4 times greater than the CPU utilization rate of MongoDB during index. Inversely, the memory utilization of MongoDB during indexing is about 3 times greater than the memory utilization of Elasticsearch during indexing.

Even though Elasticsearch was able to index 5 times more data in roughly the same amount of time as MongoDB was able to index 200,000 unique papers, the researchers are more interested in the rate of indexing and factors that could have caused the MongoDB slowdown. Of course the limitation that MongoDB can only

use one core of the CPU during insertion, in addition to the lack of its ability to upsert and append onto an existing document, were certainly factors in the slowdown.

Overall, the ability to index 1 million documents in less than two days with a completely new JSON schema makes Elasticsearch very attractive as an index system for CiteSeer^{x}.

Chapter 3

ElasticSearch Schema Design

3.1 Introduction

By choosing a new index system for CiteSeer^{x} that uses JSON to store documents instead of XML or a relation database, a new schema must be developed. JSON is unique compared to XML or relational databases in that data structures like lists or arrays may be used, in addition to nested JSON objects. It is important to note that while Elasticsearch is considered schema-less and that a schema is not inherently necessary for Elasticsearch to function properly, it was preferable to have a structure to the storing of these documents because a front end will be making calls to Elasticsearch and must know the format of the data it is receiving.

Because various JDBC connectors and automated migration features did not work with Elasticsearch, the team had to write software that would manually query

MySQL	Elasticsearch
column	field
row	document
table	type
database	index

Table 3.1: Mappings of MySQL terms and objects to the Elasticsearch ones.

and upload all of the data stored in the MySQL databases and format the result in a input format for Elasticsearch.

3.2 Cluster Design

To begin the schema design of the new Elasticsearch system, it was important to first understand the core design differences between a relational database and Elasticsearch. A table outlining the different terms for each of the respective data objects in the datastores can be seen in Table 3.1.

One of the goals of this project was to modernize the CiteSeer^{x} system and in doing so, only the newest versions of software were to be used on the new indexing system. To that extent, Elasticsearch 7.4.0 was used because it was the newest version during the system development. This choice of using Elasticsearch 7.4.0, compared to previous versions of Elasticsearch, proved to affect our cluster design greatly. In Elasticsearch versions before 6.0, it was possible to have different document types in the same index. Since we are using a newer version, this was not an option.

There is limited academic work detailing the ways to store relational data in a NoSQL datastore like Elasticsearch. The CiteSeer^x dataset is relational in nature, in having data about different papers, different authors, and different clusters. Each paper has many authors and those authors, who write other papers as well, belong to clusters. While this may seem counterintuitive because many use a NoSQL datastore for non-relational data, these datastores are still valuable for relational data. There are two main approaches which are explained in detail below.

3.2.1 Nested Object Structure

One approach for dealing with relational data in a non-relational datastore is to nest the related objects all within one document. By leveraging the power of ordered arrays and nested objects, it is possible to build a complete document with many nested fields. The drawback from this approach is that the queries to find related documents would take significantly longer to execute [34]. An example of a nested paper document can be seen in Figure 3.1 and shows the nested nature of both the cluster and author information. Additionally, a nested document datastore would include many copies of the same information. In the case of CiteSeer^x data, there would be many duplicates of author information and cluster information across the many paper documents.

```
{
   "paper_id": "10.1.1.4.201",
   "title": "The Migration of Data",
   "authors": [
       {
           "name": "Sean Parsons",
           "author_id": "123456",
           "cluster": "9876544"
       },
       {
           "name": "Andrew Warner",
           "author_id": "321455",
           "cluster": "9876544"
       }
   ]
}
```

Figure 3.1: Nested JSON example.

3.2.2 Parent-Child Structure

The other approach uses the built in parent-child document feature in Elasticsearch by using the special field join [35]. The parent and child documents must all be in the same index in order to use the join functionality in Elasticsearch. This is an important distinction and one that eliminates the use of this method for CiteSeer^{x} because we have different document types, being papers, authors and clusters. If a version of Elasticsearch prior to 6.0 was being used, then it would be possible to have different document types in the same index thus opening the possibility of having parent and child documents. Additionally, each child document can only have one parent document, which limits architecture possibilities. In Elasticsearch,

```
{
   "mappings": {
       "properties": {
           "my_id": {
               "type": "keyword"
           },
           "my_join_field": {
               "type": "join",
               "relations": {
                    "cluster": "author"
               }
           }
       }
   }
}
```

Figure 3.2: Parent-Child configuration example.

it is necessary to configure an index to allow for join functionality. To do so, a PUT request must be sent to the Elasticsearch index you are trying to configure with the payload shown in Figure 3.2. Here, it is assigning cluster as the parent document and author as a child document. This is consistent with the assumption that a cluster has many authors within it in the CiteSeer^{x} dataset.

3.2.3 Our Approach

After studying both forms of storing relational data in Elasticsearch, it was decided to use a hybrid method that optimized for query performance downstream. Even though there were restrictions on the architecture flexibility because the newest version of Elasticsearch was being used, the architecture employed optimizes for

anticipated queries. Each document type must have its own index in Elasticsearch, which means that there must be a papers index, authors index, and clusters index. The reason it is best to have different document types is because the queries that will be made pull information from papers, authors, and clusters and there are different metadata associated with each.

Therefore, each index needed some way of linking to the other two indices in the case of a more complicated query. By combining the nesting of objects and the linking of keys similar to what would be seen in a relational database, we propose a hybrid cluster architecture that can be broken down into the individual indices.

3.3 Individual Index Schemas

3.3.1 Paper Index

The schema for the paper index was the most important because it would be queried the most, it held the most information, and it contained linking information to the other indices. Much of the information contained within this index is conveyed to the user in some way. Many of the fields in this index were converted directly from the MySQL databases with minimal logic. Since the full text is not included within the MySQL databases, it must be read from the file system. The index schema of SemanticScholar served as inspiration for the new design of various index schemas in CiteSeer^{x} [36]. This is covered in greater detail in Chapter

4.

The nesting structure of having author information and keyword information in arrays is helpful in more than one way. While it may seem redundant to have things like author name and author ID be in the same nested object in a paper document, the goal is to maximize query efficiency downstream while balancing duplicity in the index. Figure 3.3 shows the complete paper index schema where all the fields and some example data are displayed.

3.3.2 Author Index

The schema for the author index contains significantly less information but is harder to generate in practice. Each paper is independent of other papers but each author has many papers. This is a classic example of the one to many relationship commonly seen in database systems today. Things become more challenging when there is a need to create a list of all the papers an author has written. This utility is needed for specific author pages in CiteSeer^{x}.

Since the migration occurs one paper at a time, there needed to be a way to append certain papers to a list contained within the author index anytime that author came up in a paper. The result of this functionality is commonly referred to as upserting data. The upserting operation is two common operations combined into one: updating a document if it exists and creating a new document if the document does not exist. This functionality is extremely helpful in many areas and

```
{
    "paper_id": "10.1.1.4.201",
    "title": "The Migration of Data",
    "cluster": "9876544"
    "authors": [
        {
            "name": "Sean Parsons",
            "author_id": "123456",
            "cluster": "9876544"
        },
        {
            "name": "Andrew Warner",
            "author_id": "321455",
            "cluster": "9876544"
        }
    ],
    "keywords": [
        {
            "keyword": "data",
            "keyword_id": "12345"
        }
    ],
    "abstract": "This is the full abstract."
    "year": 2020,
    "venue": "PSU Thesis"
    "ncites": 0,
    "scites": 0,
    "doi": ""
    "incol": Null,
    "authorNorms": Null,
    "text": "This is the full text of the paper!"
    "cites": [
        "9074080",
        "9074081"
    ],
    "citedby":[
    ],
    "vtime": "03/19/2020 10:15:31"
}
```

Figure 3.3: Paper index schema.

```
{
      "author_id": "123456"
      "name": "Sean Parsons"
      "cluster": "9876544"
      "papers": [
          "10.1.1.4.201",
          "10.1.1.21.17",
          "10.1.1.7.34"
          ],
          "affiliation": "Penn State University"
          "address": Null,
          "email": "seanpars98@gmail.com"
}
```

Figure 3.4: Author index schema.

limits the amount of queries we need to make to build out a completely relational system. Upserting is very easy to do in Elasticsearch with its updating API and the upsert flag. In figure 3.4, the full author index schema can be seen with the special upserted field being papers.

3.3.3 Cluster Index

The schema for the cluster index is very similar to the author index because it does not contain much information but it does utilize the upserting operation. In fact, it uses the upserting operation twice, appending to both the authors and papers data fields. The concept of a cluster is not necessarily relayed to the user but it is used to generate similar papers for the user and can be used for a recommendation system [37]. The concept of a cluster is at the core of the citation graph for CiteSeer^{x}.

```
{
    "cluster_id": "9876544"
    "included_papers": [
        "10.1.1.4.201",
        "10.1.1.21.17"
    ],
    "included_authors": [
        "Sean Parsons",
        "Andrew Warner"
    ]
}
```

Figure 3.5: Cluster index schema.

When linked with the data contained within the paper index, the entire citation graph can be formed. Figure 3.5 details the complete cluster index schema.

Chapter 4

CiteSeer^{x} Data Migration Process

4.1 Introduction

When migrating relational data from MySQL to Elasticsearch, there are a few different options that exist. Traditionally speaking, MySQL is one of the most common flavors of SQL and therefore it is not uncommon for users to want to migrate data from within MySQL to another data technology [29]. To do so, there are a few different methodologies that can be used. The main factors when choosing between methodologies in the case of CiteSeer^{x} were efficiency and accuracy in formatting.

Once a methodology was chosen, it needed to be implemented in the most efficient way possible. If configurations needed to be changed, then they were edited using the YAML markup language. The main configuration file for Elasticsearch

is located in /bin/elasticsearch.yaml. When manual changes needed to occur, the Python programming language was chosen to automate the querying and formatting necessary to move data from MySQL to Elasticsearch. Because a goal of this process was to use modern software engineering techniques, the team containerized the migration code necessary for replication purposes.

4.2 Migration Methodologies

Below we describe the three different methods that we tried to accurately migrate the data from MySQL to the new Elasticsearch instance. While some methods did not work well with our data set and formatting needs, they are all described in detail.

4.2.1 Third Party MySQL to Elasticsearch Syncing Tool

Because MySQL is such a common technology, there are many related third party plugins and scripts which the database community has contributed to over many years. One such tool is a MySQL to ElasticSearch Syncing tool created by the Github user siddontang [38]. This particular tool is written in the programming language Go and it interfaces with the ElasticSearch Go client. While this tool may prove valuable to users who do not have millions of entries of data and strict formatting needs, our team encountered issues while trying to use this tool.

For one, this tool requires the use of a version of Elasticsearch which is less than

6.0.0. Additionally, it does not provide any more field mapping support than the vanilla version of Elasticsearch. This third party tool also did not seem to work with multiple document types and multiple indices, which was the new schema decision for CiteSeer^x data in Elasticsearch. While it was helpful to run the tool and test out the MySQL migrating capabilities, it does not seem that this tool can provide the flexibility and the formatting needs of CiteSeer^x.

4.2.2 MySQL JDBC Connector

One of the most popular methods of migration from any SQL based database to Elasticsearch is by using Logstash and the Java Database Connector (JDBC). Logstash is a tool developed by the creaters of Elasticsearch to provide log parsing and it is commonly used with Elasticsearch [39]. By using this tool, it is possible to periodically sync data from MySQL and index it in Elasticsearch. Any Logstash version greater than 5.0.0 comes with the ability to use the JDBC tool. This being said, the specific jar file for the version of SQL being used must be downloaded beforehand. Additionally, there is an assumption that MySQL is running on port 3306 and the machine that Logstash is running on must have access to the MySQL instance in question. By configuring the settings located in a configuration file in the same directory as Logstash, it is easy to point the input SQL instance to a machine different than the one running Elasticsearch or Logstash.

There are many configurations of the JDBC tool and they all depend on the

architecture of Elasticsearch. For example, users must define what SQL query they want to run on the MySQL database. Logstash executes this query on the MySQL database, given that the user provided valid MySQL credentials in the configuration file.

While the Logstash JDBC connector seems to be helpful in simple cases, it also does not scale well to multiple indices and multiple data types. Small tests were done to test the efficacy of the Logstash JDBC tool by running the command "SELECT * FROM PAPER LIMIT 10000;" to retrieve 10,000 papers from the MySQL database and bring them over to Elasticsearch. After spending hours trying to properly configure the JDBC driver for MySQL, the process finally began to work. While it took 38 minutes to migrate 10,000 papers, the formatting was not consistent with what was in the MySQL database. Additionally, there was no support for building the authors and cluster indices with no repeat entries.

4.2.3 Custom Manual Migration

While our group tried to use many pre-built solutions to transfer the CiteSeer^x data from MySQL to Elasticsearch, eventually we decided to build our own migration software. This entailed using the Python programming language to interface with the MySQL databases as well as the Linux file system in order to index the data properly into Elasticsearch. A diagram of the migration process can be seen in Figure 4.1. A multitude of Python files were written to separate the logic of

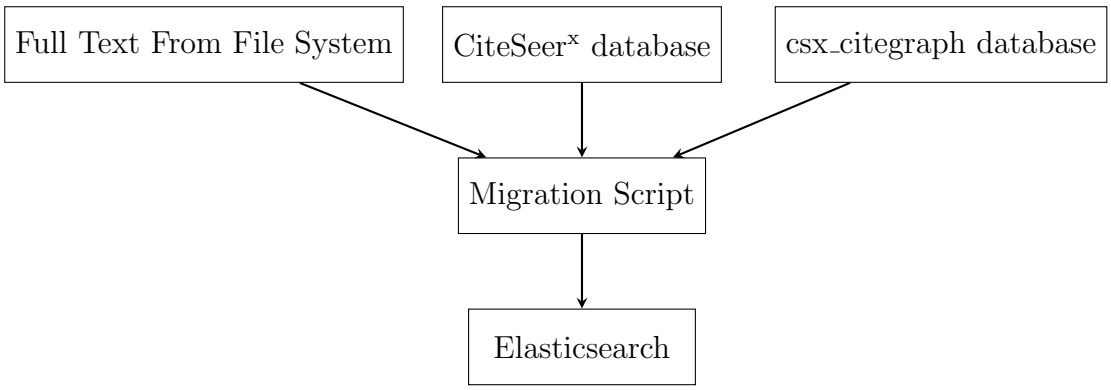

Figure 4.1: Data migration flowchart.

Filename	Description
es_migration.py	Main script, establishes db connections
elasticpython.py	Interafaces Elasticsearch
paper.py	Paper index manipulations
author.py	Author index manipulations
cluster.py	Cluster index manipulations

Table 4.1: Filenames and descriptions of migration system logic.

querying the MySQL database and the index schema initialization for each data type. A table showing the Python files needed to run the migration with a short description of the logic contained within the files can be seen in Table 4.1.

In es_migration.py, the user must change one line before the migration is ready to commence. The line that must be changed affects the number of papers to index from MySQL and into Elasticsearch. To date, over 1 million papers have been indexed by the new Elasticsearch system. After the n number of papers to be indexed from MySQL is changed, the script queries the MySQL databases and retrieves that n number of unique paper IDs. Each of these IDs is iterated through, as the software finds the full text of the paper and queries the databases

for remaining metadata. Once all the previous information from the paper is compiled, the program inserts a paper document in the paper index in Elasticsearch. With the remaining metadata, the software must use the upsert operation on the author data and cluster data in their respective indices. This way, there are no repeat cluster IDs in the cluster index and no repeat author IDs in the author index because if a record exists, then the new information will be appended to an array.

It is important to note that the only way to do this upserting logic is by using a Elasticsearch built-in scripting language that can be used during query execution called Painless [40]. An example of the upserting logic using the Painless scripting language can be found in Figure 4.2. If the record cannot be found in Elasticsearch, then it will take the arguments given and append them to the document in the authors table in this specific case.

As the migration script is iterating through papers and adding authors and clusters, it is also monitoring CPU and memory usage across time. By attaching to the process ID of Elasticsearch, the team is able to get accurate measurements of the performance metrics during the time of indexing. In order to run the migration script on the CiteSeer^{x} migration server, the terminal multiplexer (tmux) command line tool was used to detach from a terminal session so the script could run completely. As discussed earlier in the paper, to index 1 million papers into Elasticsearch, it took 1 day, 21 hours, 52 minutes and 11 seconds.

```
new_data = {}

new_data['script'] = {
    "source": "ctx._source.papers.add(params.new_papers);
               ctx._source.papers.add(params.new_clusters)",
    "lang": "painless",
    "params": {
        "new_papers": data['papers'][0],
        "new_clusters": data['clusters'][0]
    }
}

new_data['upsert'] = {
    "papers": data['papers'],
    "author_id": data['author_id'],
    "cluster": data['clusters'],
    "name": data['name'],
    "affiliation": data['affiliation'],
    "address": data['address'],
    "email": data['email']
}

update1 = es.update(index=index, doc_type=doc_type, id=doc_id,
                        body=new_data)

```

Figure 4.2: Upserting logic using the Painless and Python.

4.3 Running the Migration

To start the migration process, one can either install the Python dependencies located in the requirements.txt file within the code repository or a Docker container can be ran. By containerizing the migration script, the team has started the process of containerizing certain parts of CiteSeer^{x}. The container is very simple and is built upon the Python 2.7 base image. Once an image is produced by the Dockerfile

in the repository, the container can be run without the need of mapping any ports.

Chapter 5

Results

5.1 Experimental Results

As a result of the comparative study of this thesis, it is clear that with a relational schema and many millions of documents, Elasticsearch is a better indexing choice for CiteSeer^{x}. Even though it may have more CPU utilization compared to MongoDB, it indexes many times quicker than MongoDB. Because of the experiments conducted, the team recognizes the limitations of MongoDB in only allowing for one core to be used when inserting documents as well as the inability to upsert append in one single query. This led to a quadratic time complexity model for MongoDB, which removed the possibility of inserting 1 million papers to study.

The results of the experiments of this thesis contribute to the systems community as a whole, giving a performance metric comparison between two of the

biggest NoSQL datastores with our example schema. When system administrators and database administrators are choosing which new system to migrate data to, they are looking at performance metrics as well as overall migration duration.

In the case of CiteSeer^{x}, it is expected that if we run the same migration script but instead this time set it to migrate all 10 million papers to Elasticsearch, this operation would take just over 19 days. This is a reasonable time given the scale of data and the number of indices that must be populated.

5.2 Indexing

After the indexing experiments in this thesis, the CiteSeer^{x} lab has decided to keep the indexed 1 million papers in the Elasticsearch system and to build upon it. With Elasticsearch being the main indexing system now, there are plans to refactor CiteSeer^{x} or to change the entire system from a dependency on Apache Solr and eventually MySQL. While the metadata dataset of CiteSeer^{x} is often a valuable dataset for researchers, steps must now be taken to turn Elasticsearch into the main indexing tool system-wide.

Migrating all of the data over to Elasticsearch is just one step in changing the indexing system officially from Apache Solr.

5.3 System Updates

Now the CiteSeer^x lab must integrate Elasticsearch within its full technology stack. Various discussions have ensued and there are a few possible solutions which will be described at length in the next chapter. If the system is going to use Elasticsearch to its fullest extent, then there will be many sweeping system changes done to the architecture of CiteSeer^x.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we investigated the feasibility of using Elasticsearch and MongoDB as the new indexing systems for CiteSeer^{x}. Additionally, we analyzed various schemas and configurations that would optimize the migration process to a new system. By comparing the performance of two of the most popular NoSQL datastores in existence today, the CiteSeer^{x} team decided to use Elasticsearch because of its quick indexing times and flexibility during migration.

Additionally, this thesis proposed a hybrid method for storing relational documents in a NoSQL datastore. By deferring the use of nested objects in documents and by not using the out-of-the-box parent-child relationships in Elasticsearch, we propose the new schema representing papers, authors, and clusters. With this new

schema, we migrated significant portions of data to the new Elasticsearch system by containerizing custom migration software and deploying the migration container to our index servers.

Indexing is at the core of CiteSeer^{x} and by doing a data migration and integrating the new Elasticsearch system, work from this thesis prompted the development of a new next generation CiteSeer. This new search engine will use the Elasticsearch instance described in this thesis as its core indexing system.

Despite the overwhelming use of large NoSQL datastores in technology stacks today, there is very limited literature on the quantitative analysis of these tools. This thesis and the work of the CiteSeer^{x} team hopes to add to the knowledge of the systems and search engine community.

6.2 Future Work

There are many ways that future researchers can build off of the work that is represented in this thesis. The current CiteSeer^{x} research group only experimented on the initial indexing of documents and compared the performance metrics of just the inserting operation. Future researchers can also compare the update, delete, and query functionality of Elasticsearch and MongoDB.

Additionally, it would be valuable to have a robust comparison between all systems in question: MySQL, Apache Solr, MongoDB, Elasticsearch, in addition to more datastores like Reddis and DynamoDB. If this is completed, this would

provide the systems community with significant information on where each of these platforms stands from a performance and timing point of view.

Another possible area of work could be the packaging of various CiteSeer^x systems and assets into containers to be deployed on other virtual machines on the cloud. With the onset of cloud popularity and reduced costs, containerization can only help with mobility moving forward.

Additionally, the work of this thesis has propelled a conversation about a completely refactored version of CiteSeer^x described below.

6.2.1 Next Generation CiteSeer

After the new schema was developed and the migration to Elasticsearch began, discussions of integration with the current system had to be addressed. The current CiteSeer^x system today uses both MySQL and Apache Solr for two slightly different reasons. MySQL acts as a metadata storage for analysis and backups, while Apache Solr is the searching and indexing tool responsible for serving results for user submitted paper queries. As the Elasticsearch work was coming to a close, it was important to know how Elasticsearch was going to be integrated with the rest of CiteSeer^x, namely the front-end which must query the index system frequently.

To that end, the CiteSeer^x research group developed two possible solutions to the issue surrounding Elasticsearch integration. One solution is to refactor the legacy code base and remove all calls to Apache Solr and MySQL from the

front-end and to replace them with REST API calls to Elasticsearch. The other solution was to rebuild CiteSeer^x with a completely different front end by using new Javascript frameworks and by breaking down the back end of CiteSeer^x into scalable microservices. Both solutions are described in detail below.

6.2.2 Refactor Legacy Code

The CiteSeer^x system is very complex, totaling over 45,000 lines of code and having many moving parts. The front end of CiteSeer^x utilizes the Java Spring framework with dynamically generated content on Java Server Page files (.jsp) [41]. The initial solution by the CiteSeer^x team was to go into the legacy front end code and refactor all the calls to MySQL and to Apache Solr and to replace these calls with requests to Elasticsearch. While this is a great idea, the front end is very complicated with hundreds of .jsp files serving content to the user and there is no central place where all the data calls are.

Reviewing and becoming acquainted with the code was a challenge, given that many of the current members of the CiteSeer^x lab team are moderately new and also unfamiliar with the legacy code base. While this method may have proven to be more simple because we would just be changing a set number of calls in the front end and slowly migrating away from MySQL and Apache Solr, determining where these calls were and how best to change them proved difficult. All of these issues eventually resulted in the solution to be deemed unfeasible for the time being.

This shifted the focus of the CiteSeer^x research group to build a new system from the bottom up.

6.2.3 New CiteSeer

After much discussion, the decision was made to build a new modern CiteSeer^x with a new front end and back end. By leveraging relatively new tools like Vue.js, Django, Elasticsearch, and containers, the team was confident that a new CiteSeer^x product that leverages the new index system could be built in a timely manner.

The new system architecture of the next generation CiteSeer^x is made up of a few of the same parts as the current production CiteSeer^x. PDFMEF is used for extraction, Elasticsearch is used for indexing, and Django and Vue are used to display results to the user in record speeds. More details on this new system will be disclosed shortly.

Appendix

Code

1 GitHub

All of the code mentioned throughout this thesis can be found on the GitHub Repository located at this link on the elasticsearch branch:

https://github.com/seanpars98/CiteSeerX

2 Schema Files

```
# Import SQL capabilities
import MySQLdb

# Import basic system libraries
import sys
import time

# Reload sys and make sure the encoding is set properly to utf8
reload(sys)
sys.setdefaultencoding('utf8')

```

```
class paper:

    def __init__(self, paper_id):
        ''' Input: The specific paper ID of a paper
            Output: None
            Method: Build a value dictionary with all of the
            ↪  relevant schema information
        '''

        self.paper_id = paper_id
        self.values_dict = {

            "paper_id": self.paper_id,
            "title": '',
            "cluster": '',
            "authors": [
                {
                "name": '',
                "author_id": '',
                "cluster": ''
                }
            ],
            "keywords": [
                {
                    "keyword": '',
                    "keyword_id": ''
                }
            ],
            "abstract": '',
            "year": 0,
            "venue": '',
            "ncites": 0,
            "scites": 0,
            "doi": '',
            "incol": None,
            "authorNorms": None,
            "text": '',
            "cites": [
                    None,
                    None
            ],
            "citedby":[
                    None,
                    None
            ],
```

```
56              "vtime": None,
57
58          }
59
60
61      def paper_table_fields(self, cur):
62          ''' Input: MySQL database connection
63              Output: None
64              Method: Query the MySQL database for a specific paperID
                ↪  and properly organize
65                      the data returned in the values_dict data
                        ↪  structure.
66
67          '''
68
69          statement = "SELECT title, abstract, year, venue, ncites,
            ↪  selfCites, cluster, versionTime FROM papers WHERE id='"
            ↪  + self.paper_id + "';"
70
71          cur.execute(statement)
72
73          result_tuple = cur.fetchall()[0]
74
75          self.values_dict['title'] = str(result_tuple[0])
76          self.values_dict['abstract'] = str(result_tuple[1])
77          if result_tuple[2]:
78              self.values_dict['year'] = int(result_tuple[2])
79          self.values_dict['venue'] = str(result_tuple[3])
80          self.values_dict['ncites'] = int(result_tuple[4])
81          self.values_dict['selfCites'] = int(result_tuple[5])
82          self.values_dict['cluster'] = int(result_tuple[6])
83          self.values_dict['vtime'] =
            ↪  result_tuple[7].strftime('%Y-%m-%d %H:%M:%S')
84
85
86      def authors_table_fields(self, cur):
87          ''' Input: MySQL database connection
88              Output: None
89              Method: Query the MySQL database (authors table
                ↪  specifically) for a specific
90              paperID and properly organize the author data returned
91              in the values_dict data structure.
92
93          '''
94
```

```
95          statement = "SELECT name, id, cluster FROM authors WHERE
            ↪  paperid='" + self.paper_id + "';"
96
97          cur.execute(statement)
98
99          result_tuple = cur.fetchall()
100
101         for author in result_tuple:
102
103             temp_dict = {    "name": str(author[0]),
104                              "author_id": int(author[1]),
105                              "cluster": int(author[2])
106                         }
107             self.values_dict['authors'].append(temp_dict)
108
109         del self.values_dict['authors'][0]
110
111
112     def keywords_table_fields(self, cur):
113         ''' Input: MySQL database connection
114             Output: None
115             Method: Query the MySQL database (keywords table
                ↪  specifically) for a specific
116             paperID and properly organize the keyword data returned
117             in the values_dict data structure.
118
119         '''
120
121         statement = "SELECT keyword, id FROM keywords WHERE
            ↪  paperid='" + self.paper_id + "';"
122
123         cur.execute(statement)
124
125         result_tuple = cur.fetchall()
126
127         for keyword in result_tuple:
128             temp_dict = {    "keyword": str(keyword[0]),
129                              "keyword_id": int(keyword[1])
130                         }
131             self.values_dict['keywords'].append(temp_dict)
132
133         del self.values_dict['keywords'][0]
134
135
136     def csx_citegraph_query(self, cur):
```

```
137         ''' Input: MySQL database connection for the csx_citegraph
            ↪  database
138             Output: None
139             Method: Query the MySQL database for the citegraph data
                ↪  based off of
140             clusterID.
141
142         '''
143
144         #this statement grabs the cluster ids who have cited this
            ↪  cluster
145         statement = "SELECT citing FROM citegraph WHERE cited=" +
            ↪  str(self.values_dict['cluster']) + ";"
146         cur.execute(statement)
147
148         result_citedby_tuple = cur.fetchall()
149
150         #this statement grabs the cluster ids who are cited by this
            ↪  cluster
151         statement2 = "SELECT cited FROM citegraph WHERE citing=" +
            ↪  str(self.values_dict['cluster']) + ";"
152
153         cur.execute(statement2)
154
155         result_cites_tuple = cur.fetchall()
156
157         self.values_dict['citedby'] = [int(cite[0]) for cite in
            ↪  result_citedby_tuple]
158         self.values_dict['cites'] = [int(cite[0]) for cite in
            ↪  result_cites_tuple]
159
160
161     def retrieve_full_text(self):
162         ''' Input: None
163             Output: None
164             Method: We traverse through the local filesystem to find
                ↪  the full text
165                     .txt file. Then, we open this file and populate
                        ↪  the values
166                     dictionary with the full text.
167
168         '''
169
170         d_path = self.paper_id.split('.')
171
```

```
172         text_file_path = "/mnt/rep1/%s/%s/%s/%s/%s/%s.txt" %
            ↪  (d_path[0], d_path[1], d_path[2], d_path[3], d_path[4],
            ↪  self.paper_id)
173
174         try:
175
176             with open(text_file_path, "r") as text_file:
177
178                 contents = text_file.read()
179                 resp = ''.join(contents)
180                 self.values_dict['text'] = str(resp)
181
182         except IOError:
183             print("full text file could not be found")
184
185
```

Listing 1: Paper index schema declaration.

```
1
2   class author:
3
4       def __init__(self, author_id):
5           ''' Input: The specific author ID of an author
6               Output: None
7               Method: Build a value dictionary with all of the
                ↪  relevant schema information
8           '''
9
10          self.author_id = author_id
11          self.values_dict = {
12
13              "author_id": self.author_id,
14              "name": None,
15              "clusters": [
16
17              ],
18              "papers": [
19
20              ],
21              "affiliation": None,
22              "address": None,
23              "email": None
24
```

```
        }



    def authors_table_fields(self, cur):
        ''' Input: MySQL database connection
            Output: None
            Method: Query the MySQL database (authors table
            ↪  specifically) for a specific
            authorID and properly organize the author data returned
            in the values_dict data structure.

        '''

        statement = "SELECT affil, address, email FROM authors WHERE
        ↪  id='" + str(self.author_id) + "';"

        cur.execute(statement)

        result_tuple = cur.fetchall()[0]

        self.values_dict['affiliation'] = result_tuple[0]
        self.values_dict['address'] = result_tuple[1]
        self.values_dict['email'] = result_tuple[2]

```

Listing 2: Author index schema declaration.

```


class cluster:

    def __init__(self, cluster_id):
        ''' Input: The specific cluster ID of a cluster
            Output: None
            Method: Build a value dictionary with all of the
            ↪  relevant schema information
        '''

        self.cluster_id = cluster_id
        self.values_dict = {

            "cluster_id": self.cluster_id,
            "included_papers": [
```

```
                None,
                None
            ],
            "included_authors": [
                None,
                None
            ]

        }
```

Listing 3: Cluster index schema declaration.

3 Experiment Monitoring and Automation

```
# Import capabilities to make HTTP requests to ElasticSearch
import requests

import zlib
# Import ability to work with JSON objects in Python
import json

# Import ElasticSearch API for Python
from elasticsearch import Elasticsearch


def establish_ES_connection():
    ''' Input: None
        Output: ElasticSearch connection
        Method: Using the ElasticSearch Python API

    '''

    es = Elasticsearch([{'host': '130.203.139.151',
                         'port': 9200
                        }])

    return es


def test_ES_connection():
    ''' Input: None
        Output None
```

```
29          Method: Test Python's connection to ElasticSearch and print
            ↪ the response
30
31      '''
32
33      req = requests.get('http://130.203.139.151:9200')
34      content = req.content
35      parsed = json.loads(content)
36      print_response(parsed)
37
38
39  def print_response(response):
40      ''' Input: None
41          Output: None
42          Method: Prints the JSON of the response from ElasticSearch
            ↪ to test connection
43
44      '''
45
46      print(json.dumps(response, indent=4, sort_keys=True))
47
48
49  #If the document exists already, update the document where the
    ↪ doc_id's are the same
50  def update_authors_document(es, index, doc_id, doc_type, data):
51      ''' Input: ElasticSearch instance, index name (authors),
        ↪ document id, document type (authors), and data dictionary
52          Output: None
53          Method: First we properly format scripts to be ran on
            ↪ ElasticSearch in
54                  order to upsert the correct values using the
                    ↪ painless scripting
55                  language. My formatting the dictionaries in such a
                    ↪ way that will
56                  allow ElasticSearch to upsert the document into the
                    ↪ authors index,
57                  we don't need to worry about if the document exists
                    ↪ already. Then we
58                  use the traditional 'update' command for
                    ↪ ElasticSearch to apply the upsert.
59      '''
60
61      new_data = {}
62
```

```
63      source = "ctx._source.papers.add(params.new_papers);
        ↪  ctx._source.papers.add(params.new_clusters)"
64      # We also need to add a script to the JSON to check and add the
        ↪  associated data appropriately
65      new_data['script'] = {
66              "source": source,
67              "lang": "painless",
68              "params": {
69                  "new_papers": data['papers'][0],
70                  "new_clusters": data['clusters'][0]
71              }
72       }
73
74      new_data['upsert'] = {
75                      "papers": data['papers'],
76                      "author_id": data['author_id'],
77                                          "cluster": data['clusters'],
78                                          "name": data['name'],
79                      "affiliation": data['affiliation'],
80                      "address": data['address'],
81                      "email": data['email']
82
83      }
84
85      # Update the specific document located by the ID
86      update1 = es.update(index=index, doc_type=doc_type, id=doc_id,
87                          body=new_data)
88
89
90  def update_clusters_document(es, index, doc_id, doc_type, data):
91      ''' Input: ElasticSearch instance, index name (clusters),
        ↪  document id, document type (clusters), and data dictionary
92          Output: None
93          Method: First we properly format scripts to be ran on
            ↪  ElasticSearch in
94                  order to upsert the correct values using the
                    ↪  painless scripting
95                  language. My formatting the dictionaries in such a
                    ↪  way that will
96                  allow ElasticSearch to upsert the document into the
                    ↪  clusters index,
97                  we don't need to worry about if the document exists
                    ↪  already. Then we
98                  use the traditional 'update' command for
                    ↪  ElasticSearch to apply the upsert.
```

```
99      '''
100
101     new_data = {}
102
103     source = "ctx._source.included_papers.add(params.new_papers);
        ↪  ctx._source.included_authors.add(params.new_authors)"
104     new_data['script'] = {
105             "source": source,
106             "lang": "painless",
107             "params": {
108                 "new_papers": data['included_papers'][0],
109                 "new_authors": data['included_authors']
110             }
111     }
112
113
114     new_data['upsert'] = {
115                     "cluster_id": data['cluster_id'],
116                     "included_papers": data['included_papers'],
117                     "included_authors": data['included_authors']
118     }
119
120
121     update1 = es.update(index=index, doc_type=doc_type, id=doc_id,
        ↪  body=new_data)
122
123
124 def create_document(es, index, doc_id, doc_type, data):
125     ''' Input: ElasticSearch instance, index name (papers), document
        ↪  id, document type (papers), and data dictionary
126         Output: None
127         Method: For each paper, we need to create a document in
            ↪  ElasticSearch.
128
129     '''
130
131     # Begin indexing the data in the correct index
132     index1 = es.index(index=index, id=doc_id, doc_type=doc_type,
        ↪  body=data)
133
134
135
136
137
```

Listing 4: Custom wrapper for the Elasticsearch library.

```
from pymongo import MongoClient
from pprint import pprint
from paper import paper
from author import author
from cluster import cluster
from monitoring import Monitor

class Mongo():

    def __init__(self):
        self.client = None
        self.db = None

    def establishMongoConnection(self):
        client = MongoClient('localhost', 27017)
        self.client = client
        self.db = self.client['citeseerx']

    def getCollection(self, colName):
        collection = self.db[colName]
        return collection

    def createDocument(self, collection, data):
        col = self.db[collection]

        # Did not assign ID, therefore mongo will give us a
        ↪  generated one
        result = col.insert_one(data)


    def checkIfDocExists(self, collection, idType, idValue):

        if self.db[collection].find({idType: idValue}).count() > 0:
            return True
        else:
            return False


    def updateAuthorHelper(self, collection, data):

        col = self.db[collection]
        response = col.update_one(
            {
```

```
43                  "author_id": data['author_id']
44              },
45              {    "$addToSet": { "clusters": { "$each":
                ↪  data['clusters'][0]},
46                              "papers": { "$each": data['papers'][0]}
47                      }
48              })
49
50      def insertAuthorHelper(self, collection, data):
51
52          col = self.db[collection]
53          response = col.insert_one(data)
54
55      def upsertAuthor(self, paper, collection, db):
56
57          for auth in paper.values_dict['authors']:
58
59              author1 = author(auth['author_id'])
60
61              author1.values_dict['clusters'] = [auth['cluster']]
62              author1.values_dict['name'] = auth['name']
63              author1.values_dict['papers'] =
                ↪  [paper.values_dict['paper_id']]
64
65              author1.authors_table_fields(db)
66
67              # Now that author is prepared, time to switch logic
                ↪  depending on if the
68              # entry exists already
69              if self.checkIfDocExists("authors", "author_id",
                ↪  author1.values_dict['author_id']):
70                  # Append paper and cluster to author entry!
71                  self.updateAuthorHelper(collection,
                    ↪  author1.values_dict)
72              else:
73                  # Insert the brand new document!
74                  self.insertAuthorHelper(collection,
                    ↪  author1.values_dict)
75
76
77
78      def updateClusterHelper(self, collection, data):
79          col = self.db[collection]
80
81          result = col.update_one(
```

```
        {
            "cluster_id": data['cluster_id']
        },
        {   "$addToSet": { "included_papers": { "$each": data['included_papers']},
                           "included_authors": { "$each": data['included_authors']}
                         }
                        })

    def insertClusterHelper(self, collection, data):

        col = self.db[collection]
        response = col.insert_one(data)

    def upsertCluster(self, paper, collection):
        cluster1 = cluster(paper.values_dict['cluster'])
        cluster1.values_dict['included_papers'] = [paper.values_dict['paper_id']]
        list_of_author_names = [auth['name'] for auth in paper.values_dict['authors']]
        cluster1.values_dict['included_authors'] = list_of_author_names


        if self.checkIfDocExists("clusters", "cluster_id", cluster1.values_dict['cluster_id']):
            # If the document exists, then append values
            self.updateClusterHelper(collection, cluster1.values_dict)
        else:
            # Create the document from scratch!
            self.insertClusterHelper(collection, cluster1.values_dict)

```

Listing 5: Custom wrapper for the MongoDB library.

```
import psutil
from datetime import datetime
from pprint import pprint
import csv
import time
import pandas as pd
```

```
7  import pickle
8
9  class Monitor:
10
11     def __init__(self, pid):
12
13         self.pid = pid
14         self.cpu_usage = {'cpu_usage': []}
15         self.memory_usage = {'memory_usage': []}
16
17
18     def getData(self):
19         timestamp = datetime.now().strftime("%d-%m-%Y
           ↪  (%H:%M:%S.%f)")
20         self.getCPU(timestamp)
21         self.getMemory(timestamp)
22
23     def getCPU(self, timestamp):
24         cpus = []
25         p = psutil.Process(pid=self.pid)
26         for i in range(10):
27             p_cpu = p.cpu_percent(interval=.1)
28             cpus.append(p_cpu)
29         self.cpu_usage['cpu_usage'].append([timestamp,
           ↪  float(sum(cpus))/len(cpus)])
30
31
32     def getMemory(self, timestamp):
33         self.memory_usage['memory_usage'].append([timestamp,
           ↪  dict(psutil.virtual_memory()._asdict())])
34
35
36     def toCSV(self):
37
38         with open('cpus.p', 'wb') as f:
39             pickle.dump(self.cpu_usage, f)
40
41         with open('mem.p', 'wb') as f:
42             pickle.dump(self.memory_usage, f)
43
44     lines = []
45     avail = 'available'
46     for i in range(len(self.cpu_usage['cpu_usage'])):
47         temp = str(self.cpu_usage['cpu_usage'][i][0]) + ',' +
           ↪  str(self.cpu_usage['cpu_usage'][i][1]) + ','
```

```
48          temp +=
            ↪  str(self.memory_usage['memory_usage'][i][1]['percent'])
            ↪  + ',' +
            ↪  str(self.memory_usage['memory_usage'][i][1]['active']) +
            ↪  ','
49          temp += str(self.memory_usage['memory_usage'][i][1][avail])
            ↪  + ',' +
            ↪  str(self.memory_usage['memory_usage'][i][1]['free']) +
            ↪  ','
50          temp +=
            ↪  str(self.memory_usage['memory_usage'][i][1]['inactive'])
            ↪  + ',' +
            ↪  str(self.memory_usage['memory_usage'][i][1]['total']) +
            ↪  ','
51          temp += str(self.memory_usage['memory_usage'][i][1]['used'])
52          lines.append(temp)
53
54          filename = str(self.pid) + '.csv'
55
56          with open(filename, 'w') as f:
57              w = csv.writer(f, delimiter=',')
58              w.writerows([x.split(',') for x in lines])
59
60          df = pd.read_csv(filename)
61          df.columns = ['timestamp', 'cpu', 'percent', 'active',
            ↪  'available', 'free', 'inactive', 'total', 'used']
62          df.to_csv(filename, index=False)
63
```

Listing 6: Monitoring script that captures memory and CPU data.

4 Migration Files

```
1   # Import SQL capabilities
2   import MySQLdb
3
4   # Import ElasticSearch capabilities
5   import elasticpython
6
7   # Import MongoDB capabilities
8   #import mongo
9
```

```
10  # Import each of the schemas and associated methods for each index
11  from paper import paper
12  from author import author
13  from cluster import cluster
14  from monitoring import Monitor
15
16
17  def get_ids(cur, n):
18      ''' Input: Database cursor (database connection), n number of
        ↪  papers to retrieve
19          Output: Returns a list of first 'n' number of paper ids from
            ↪  the SQL DB
20          Method: Queries the database for the paper ids and returns a
            ↪  list of length 'n'
21
22      '''
23
24      statement = "SELECT id FROM papers LIMIT %d;" % (n)
25
26      cur.execute(statement)
27
28      return [tup[0] for tup in cur.fetchall()]
29
30
31  def connect_to_citeseerx_db():
32      ''' Input: None
33          Output: Returns the cursor (connection) to the citeseerx
            ↪  database
34          Method: Using the python MySQL API, establishes a connection
            ↪  with the citeseerx DB
35
36      '''
37
38      db = MySQLdb.connect(host="",
39                           user="",
40                           passwd="",
41                           db="",
42              charset='utf8')
43
44      return db.cursor()
45
46
47  def connect_to_csx_citegraph():
48      ''' Input: None
```

```
49          Output: Returns the cursor (connection) to the csx_citegraph
            ↪ DB
50          Method: Using the python MySQL API, connects to the
            ↪ csx_citegraph database
51
52      '''
53
54      db = MySQLdb.connect(host="",
55                           user="",
56                           passwd="",
57                           db="",
58              charset='utf8')
59
60      return db.cursor()
61
62
63  def authorHelperUpsert(paper, citeseerx_db_cur):
64      ''' Input: Paper object with it's values dictionary, and
        ↪ citeseerx database connection
65          Output: None
66          Method: Iterate through each author on a given paper,
            ↪ prepare the dictionary
67                  for upsertion into the authors index in
                    ↪ ElasticSearch.
68                  Upserting means insert if the object doesn't already
                    ↪ exist, update if it does
69
70      '''
71
72      for auth in paper.values_dict['authors']:
73
74              author1 = author(auth['author_id'])
75
76              author1.values_dict['clusters'] = [auth['cluster']]
77              author1.values_dict['name'] = auth['name']
78              author1.values_dict['papers'] =
                ↪ [paper.values_dict['paper_id']]
79
80              author1.authors_table_fields(citeseerx_db_cur)
81
82              elasticpython.update_authors_document(es,
                ↪ index='authors',
                ↪ doc_id=author1.values_dict['author_id'],
83                  doc_type='author', data=author1.values_dict)
84
```

```
85
86  def clusterHelperUpsert(paper):
87      ''' Input: Paper object with it's values dictionary
88          Output: None
89          Method: Prepare the clusters dictionary for upsertion into
            ↪  ElasticSearch
90
91      '''
92
93      cluster1 = cluster(paper.values_dict['cluster'])
94
95      cluster1.values_dict['included_papers'] =
        ↪  [paper.values_dict['paper_id']]
96
97      list_of_author_names = [auth['name'] for auth in
        ↪  paper.values_dict['authors']]
98
99      cluster1.values_dict['included_authors'] = list_of_author_names
100
101     elasticpython.update_clusters_document(es, index='clusters',
        ↪  doc_id=cluster1.values_dict['cluster_id'],
102         doc_type='cluster', data=cluster1.values_dict)
103
104
105
106
107 if __name__ == "__main__":
108     ''' Main Method
109         Method: Call all above methods then sets the number of
            ↪  papers to index.
110                 Iterates through each paper and indexes the paper,
                    ↪  all authors, and the cluster
111                 of said paper.
112
113
114     '''
115
116     # Establish connections to databases and ElasticSearch
117     citeseerx_db_cur = connect_to_citeseerx_db()
118     csx_citegraph_cur = connect_to_csx_citegraph()
119     es = elasticpython.establish_ES_connection()
120     elasticpython.test_ES_connection()
121
122     # Set the number of papers to index by this migration script
123     number_of_papers_to_index = 1000000
```

```
124
125     moni = Monitor(66912)
126     # Retrieve the list of paper ids
127     list_of_paper_ids = get_ids(citeseerx_db_cur,
        ↪  number_of_papers_to_index)
128
129     # Set counter so we can keep track of how many papers have
        ↪  migrated in real-time
130     paper_count = 0
131
132     # Iterate through each of the paper_ids selected and add them to
        ↪  the index
133     for paper_id in list_of_paper_ids:
134
135         # Every 100 papers print out our current progress
136         if paper_count % 100 == 0:
137             print('Total paper count: ', str(paper_count))
138
139         # Every 10,000 papers, record the metrics we want
140         if paper_count % 10000 == 0:
141             moni.getData()
142
143
144         # Extract all the fields neccessary for the paper type from
            ↪  the MySQL DBs
145         paper1 = paper(paper_id)
146         paper1.paper_table_fields(citeseerx_db_cur)
147         paper1.authors_table_fields(citeseerx_db_cur)
148         paper1.keywords_table_fields(citeseerx_db_cur)
149         paper1.csx_citegraph_query(csx_citegraph_cur)
150         paper1.retrieve_full_text()
151
152         # Load the paper JSON data into ElasticSearch
153         elasticpython.create_document(es, index='citeseerx',
            ↪  doc_id=paper1.values_dict['paper_id'], doc_type='paper',
            ↪  data=paper1.values_dict)
154
155
156         # We also need to update the other indices like author and
            ↪  cluster
157         # By using the update and upserts command in ElasticSearch,
            ↪  we can do this easily
158         authorHelperUpsert(paper1, citeseerx_db_cur)
159         clusterHelperUpsert(paper1)
160
```

```
161         # Increment counter so we can keep track of migration
            ↪  progress
162         paper_count += 1
163
164     moni.toCSV()
```

Listing 7: Elasticsearch migration script.

```
1  # Dockerfile
2
3  FROM python:2.7
4
5  COPY . /migration_app
6
7  WORKDIR /migration_app
8
9  RUN pip install -r requirements.txt
10
11 ENTRYPOINT ["python"]
12
13 CMD ["es_migration.py"]
```

Listing 8: Dockerfile used to containerize migration app.

Bibliography

[1] Giles, C. L., K. D. Bollacker, and S. Lawrence (1998) "CiteSeer: An Automatic Citation Indexing System," ACM Press, pp. 89–98.

[2] Li, H., I. Councill, W.-C. Lee, and C. L. Giles (2006) "CiteSeerx: an architecture and web service design for an academic document search engine," in *Proceedings of the 15th international conference on World Wide Web*, WWW '06, Association for Computing Machinery, Edinburgh, Scotland, pp. 883–884.
URL https://doi.org/10.1145/1135777.1135926

[3] Wu, J., K. M. Williams, H.-H. Chen, M. Khabsa, C. Caragea, S. Tuarob, A. G. Ororbia, D. Jordan, P. Mitra, and C. L. Giles (2015) "CiteSeerX: AI in a Digital Library Search Engine," *AI Magazine*, **36**(3), pp. 35–48, number: 3.
URL https://www.aaai.org/ojs/index.php/aimagazine/article/view/2601

[4] Jacsó, P. (2005) "Google Scholar: the pros and the cons," *Online Information Review*, **29**(2), pp. 208–214, publisher: Emerald Group Publishing Limited.
URL https://doi.org/10.1108/14684520510598066

[5] Caragea, C., J. Wu, A. Ciobanu, K. Williams, H.-h. Chen, Z. Wu, and L. Giles (2014) "CiteSeerX: A scholarly big dataset," in *Proceedings of the 36th European Conference on Information Retrieval*, pp. 311–322.

[6] Ochoa, X. and E. Duval (2009) "Automatic evaluation of metadata quality in digital repositories," *International Journal on Digital Libraries*, **10**(2), pp. 67–91.
URL https://doi.org/10.1007/s00799-009-0054-4

[7] Ortega, J. L. (2014) *Academic Search Engines*, Elsevier.
URL https://linkinghub.elsevier.com/retrieve/pii/C20130232268

[8] Ammar, W., D. Groeneveld, C. Bhagavatula, I. Beltagy, M. Crawford, D. Downey, J. Dunkelberger, A. Elgohary, S. Feldman,

V. Ha, R. Kinney, S. Kohlmeier, K. Lo, T. Murray, H.-H. Ooi, M. Peters, J. Power, S. Skjonsberg, L. L. Wang, C. Wilhelm, Z. Yuan, M. van Zuylen, and O. Etzioni (2018) "Construction of the Literature Graph in Semantic Scholar," *arXiv:1805.02262 [cs]*, arXiv: 1805.02262.
URL http://arxiv.org/abs/1805.02262

[9] Ley, M. (2009) "DBLP - Some Lessons Learned." *PVLDB*, **2**(2), pp. 1493–1500.

[10] "MEDLINE®PubMed® XML Element Descriptions and their Attributes," Library Catalog: www.nlm.nih.gov Publisher: U.S. National Library of Medicine.
URL https://www.nlm.nih.gov/bsd/licensee/elements_descriptions.html

[11] "Web of Science Core Collection Schema," .
URL http://help.incites.clarivate.com/wosWebServicesExpanded/wosSchemaWoSCCGroup/wosSchema.html

[12] Xiong, C., R. Power, and J. Callan (2017) "Explicit Semantic Ranking for Academic Search via Knowledge Graph Embedding," in *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, International World Wide Web Conferences Steering Committee, Perth, Australia, pp. 1271–1279.
URL https://doi.org/10.1145/3038912.3052558

[13] Duval, E. "Metadata standards: What, who & why," *Journal of Universal Computer Science*.
URL www.academia.edu/1163669/Metadata_standards_What_who_and_why

[14] Greenberg, J. (2005) "Understanding Metadata and Metadata Schemes," *Cataloging & Classification Quarterly*, **40**(3-4), pp. 17–36, publisher: Routledge _eprint: https://doi.org/10.1300/J104v40n03_02.
URL https://doi.org/10.1300/J104v40n03_02

[15] Amorim, R. C., J. A. Castro, J. Rocha da Silva, and C. Ribeiro (2017) "A comparison of research data management platforms: architecture, flexible metadata and interoperability," *Universal Access in the Information Society*, **16**(4), pp. 851–862.
URL https://doi.org/10.1007/s10209-016-0475-y

[16] "Apache Solr -," .
URL https://lucene.apache.org/solr/

[17] Kononenko, O., O. Baysal, R. Holmes, and M. W. Godfrey (2014) "Mining modern repositories with elasticsearch," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, Association for Computing Machinery, Hyderabad, India, pp. 328–331.
URL https://doi.org/10.1145/2597073.2597091

[18] Nurseitov, N., M. Paulson, R. Reynolds, and C. Izurieta (2009) "Comparison of JSON and XML Data Interchange Formats: A Case Study," in *CAINE*.

[19] Gao, S., J. J. Li, and J. P. Schmeiser (2015), "Generating XML schema from JSON data," Library Catalog: Google Patents.
URL https://patents.google.com/patent/US9075833/en

[20] "Spring," Library Catalog: spring.io.
URL https://www.spring.io

[21] "Apache Tomcat® - Welcome!" .
URL http://tomcat.apache.org/

[22] "MySQL," .
URL https://www.mysql.com/

[23] Wu, J., J. Killian, H. Yang, K. Williams, S. R. Choudhury, S. Tuarob, C. Caragea, and C. L. Giles (2015) "PDFMEF: A Multi-Entity Knowledge Extraction Framework for Scholarly Documents and Semantic Search," in *Proceedings of the 8th International Conference on Knowledge Capture*, K-CAP 2015, Association for Computing Machinery, Palisades, NY, USA, pp. 1–8.
URL https://doi.org/10.1145/2815833.2815834

[24] (2020), "Docker Documentation," Library Catalog: docs.docker.com.
URL https://docs.docker.com/

[25] Greca, S., A. Kosta, and S. Maxhelaku "Optimizing data retrieval by using MongoDb with Elasticsearch," , p. 6.

[26] "Use Cases · Elastic Stack Success Stories | Elastic," Library Catalog: www.elastic.co.
URL https://www.elastic.co/customers/

[27] "Elasticsearch Reference [7.x] | Elastic," Library Catalog: www.elastic.co.
URL https://www.elastic.co/guide/en/elasticsearch/reference/7.x/index.html

[28] "Python Elasticsearch Client — Elasticsearch 8.0.0 documentation," .
URL https://elasticsearch-py.readthedocs.io/en/master/

[29] "Stack Overflow Developer Survey 2018," Library Catalog: insights.stackoverflow.com.
URL https://insights.stackoverflow.com/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share

[30] "PyMongo 3.9.0 Documentation — PyMongo 3.9.0 documentation," .
URL https://api.mongodb.com/python/current/

[31] Akca, M. A., T. Aydoğan, and M. İlkuçar (2016) "An Analysis on the Comparison of the Performance and Configuration Features of Big Data Tools Solr and Elasticsearch," *International Journal of Intelligent Systems and Applications in Engineering*, pp. 8–12.
URL https://www.ijisae.org/IJISAE/article/view/912

[32] "psutil documentation — psutil 5.7.0 documentation," .
URL https://psutil.readthedocs.io/en/latest/

[33] "Bulk Write Operations — MongoDB Manual," Library Catalog: docs.mongodb.com.
URL https://docs.mongodb.com/manual/core/bulk-write-operations

[34] "Nested datatype | Elasticsearch Reference [7.6] | Elastic," .
URL https://www.elastic.co/guide/en/elasticsearch/reference/current/nested.html

[35] "Join datatype | Elasticsearch Reference [7.6] | Elastic," .
URL https://www.elastic.co/guide/en/elasticsearch/reference/current/parent-join.html

[36] Lo, K., L. L. Wang, M. Neumann, R. Kinney, and D. S. Weld (2019), "GORC: A large contextual citation graph of academic papers," 1911.02782.

[37] Chen, H.-H., P. Treeratpituk, P. Mitra, and C. L. Giles (2013) "CSSeer: an expert recommendation system based on CiteseerX," in *Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '13, Association for Computing Machinery, Indianapolis, Indiana, USA, pp. 381–382.
URL https://doi.org/10.1145/2467696.2467750

[38] siddontang (2020), "siddontang/go-mysql-elasticsearch," Original-date: 2015-01-15T09:54:18Z.
URL https://github.com/siddontang/go-mysql-elasticsearch

[39] "Logstash Reference [7.6] | Elastic," Library Catalog: www.elastic.co.
URL https://www.elastic.co/guide/en/logstash/7.6/index.html

[40] "Painless scripting language | Elasticsearch Reference [master] | Elastic," .
URL https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-scripting-painless.html

[41] Jordan, D. W. (2016) "Lessons In Scaling A Large Digital Library: A Case Study For Citeseerx," .
URL https://etda.libraries.psu.edu/catalog/29174

[42] "Jdbc input plugin | Logstash Reference [7.6] | Elastic," Library Catalog: www.elastic.co.
URL https://www.elastic.co/guide/en/logstash/current/plugins-inputs-jdbc.html

[43] "Apache Lucene - Welcome to Apache Lucene," .
URL https://lucene.apache.org/

[44] "MongoDB Documentation," Library Catalog: docs.mongodb.com.
URL https://docs.mongodb.com/