The Pennsylvania State University
The Graduate School

OVERCOMING THE BOTTLENECK OF EXTRACTING AND INDEXING HUNDREDS OF MILLIONS OF ACADEMIC PAPERS TO SUPPORT A SCHOLARLY BIG DATA SERVICE: A CASE STUDY OF CITESEERX

A Thesis in
Computer Science and Engineering
by
Sai Raghav Reddy Keesara

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2021

The thesis of Sai Raghav Reddy Keesara was reviewed and approved by the following:

Clyde Lee Giles
David Reese Professor of Information Sciences and Technology
Thesis Advisor

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering

Chita R. Das
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Jian Wu
Assistant Professor of Computer Science Old Dominion University
Special Signatory

Abstract

CiteSeerX is one of the world's first academic digital libraries. After being established in 1998, CiteSeerX has been serving researchers with access to scholarly big data in various scientific domains. It serves about 2 millions hits with around 40-60 concurrent users on a typical day. The system hasn't been scaling so well with SQL database as the main bottleneck in search performance and lower ingestion throughput. While the previous works have justified the need to migrate to a NoSQL database, the current work realizes it by designing and implementing an Extraction and Ingestion system that is capable of ingesting up to a million academic documents per day and addresses shortcomings of the previous architecture in terms of scalability, modularity, and usability.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am gratefully indebted to my mentors Dr. Lee Giles and Dr. Jian Wu for all their guidance with my project for the past several semesters. This wouldn't have been possible without their inputs. They have been important contributors of many novel ideas that have been implemented in this work. They are always supportive and gave me all the resources I needed.

I would like to express my sincere gratitude to Dr. Bhuvan Urgaonkar for reviewing my work and for taking time in his busy schedule to take part in my Thesis committee.

To my fellow labmates Shaurya Rohatgi, Jason Chhay and Kevin Kuo, you have been a great help in giving me ideas whenever I was stuck, this thesis wouldn't have been possible without your contributions. Many thanks to Bharath Kandimalla, who has been my motivation even before I started working with CiteSeerX.

To my family who were patient through the entire process, thank you.

Chapter 1
Introduction

Digital Library Search Engines (DSLEs here after) started since the end of 20th century and CiteSeerX is one of the world's first academic digital library and search engine originally launched in 1998. The system currently holds about 7 million+ full text PDF Documents and 45 million citation graph nodes and 112.5 million edges records [1]. The system indexes citation graph of citing and cited relationship between academic papers and helps researchers search and navigate related articles. Other academic digital libraries include Google Scholar, Microsoft Academic Search, Allen AI's Semantic Scholar, ArXiv. While most of them are proprietary, CiteSeerX architecture and code is entirely open-source helping researchers take advantage of the knowledge of building a scalable DLSE.

CiteSeerX differs from ArXiv since the data obtained is from web crawl and hence do not necessarily contain clean metadata and unique papers. While other DLSEs do not necessarily serve full text documents by indexing metadata records, CiteSeerX always serves cached copies of PDFs for download and is unique in that aspect.

While current CiteSeerX has been pioneering open source search engines ever since it was launched, it still had it's limitations in terms of scalability and usability, The system is based on MySQL datastore and the full text is indexed by Solr. It is tedious to maintain and scale SQL servers and they do not meet the ingestion and performance requirements. There has always been scope for improving the system to meet the performance and feature goals for coming years.

Extending the works of [3] and [4] who have addressed and proposed in their thesis, we briefly discuss the concerns with the current system, requirements of the new system

and make an initial step towards building the new system almost entirely from scratch while taking inspiration from the current CiteSeerX.

The rest of the thesis is organized as follows. In Chapter-2, we give an overview of the current architecture and it's limitations, in Chapter-3 we propose, design and implement a prototype architecture that addresses and overcomes the issues mentioned in Chapter-2 and describe several novel approaches that made it possible. In Chapter-4 we evaluate the new architecture and provide a few results. In Chapter-5, we provide ourselves with future directions before concluding in Chapter-6. The code repository, documentation and a preview of prototype and other details in Appendix.

Chapter 2
System Overview & Limitations

In this chapter we give an overview of current CiteSeerX architecture, the infrastructure used to host the service and process of Ingesting new documents. We describe the limitations of the system. Finally, we mention the contributions of this work that enabled us to overcome the mentioned limitations with the new architecture.

2.1 Current Architecture of CiteSeerX

The current CiteSeerX system consists of about seven main components which are, a Web Server, SQL Database, Index severs running Apache Solr. Crawler Servers that crawls the web for PDF documents, an Extraction server (PDFMEF) and Ingestion service, a DOI service that uniquely identifies each paper with CiteSeerX ID (unique identifier for documents in the corpus used internally), a File repository based on Global File System (GFS) that serves PDF for download. The components and their interactions are shown in Figure 2.1 [9].

The crawler crawls the web and downloads PDF documents, after which the metadata is extracted and are ingested into SQL database and the file server marking with CiteSeerX ID (doi, digital object identifier) provided by DOI Service, the full text is indexed by index servers running Apache Solr [16]. The web service interacts with the database, repository and the index server to provide the search interface to the end users.

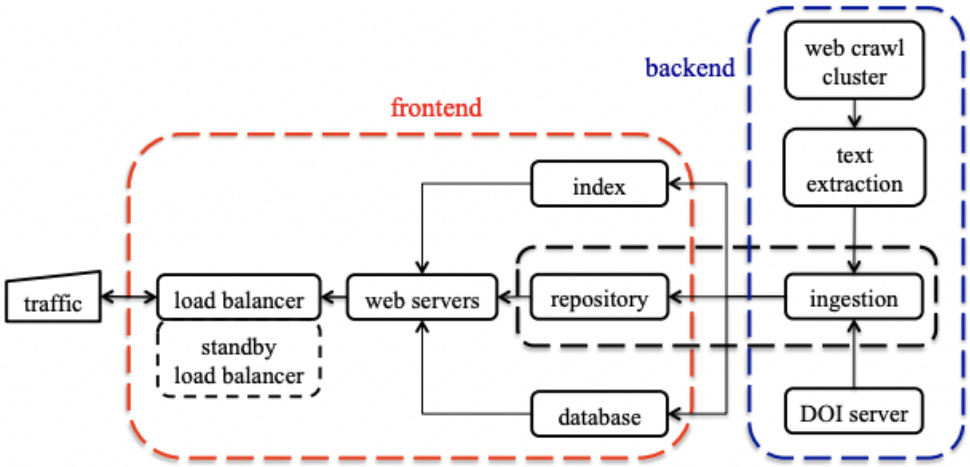

Figure 2.1. Overall architecture of CiteSeerX [9]

2.2 Current Infrastructure of CiteSeerX

The application is hosted on VMware ESXi based private cloud infrastructure in the College of IST at The Pennsylvania State University. The Virtual Machines deployed and their interactions shown in Figure 2.2. We note that there is some level of redundancy and fault-tolerance as we back up the database and use multiple servers behind the load balancer to serve the end users in the event of failure.

2.2.1 Crawler

The crawler is run on a rack server with 24 cores, 32 GB RAM, 30 TB HDD. The service is responsible for scraping the web looking for academic papers. The crawler dumps its contents to a file system as well as stores metadata about the crawl in a crawl database. The crawler follows a breadth first approach that crawls up to a specified depth on specified whitelist domains.

2.2.2 Extraction Service

Servers csxextraction01 and csxextraction02 run PDFMef framework [8], a multi entity extraction framework that filters academic documents, extracts entities like text (&

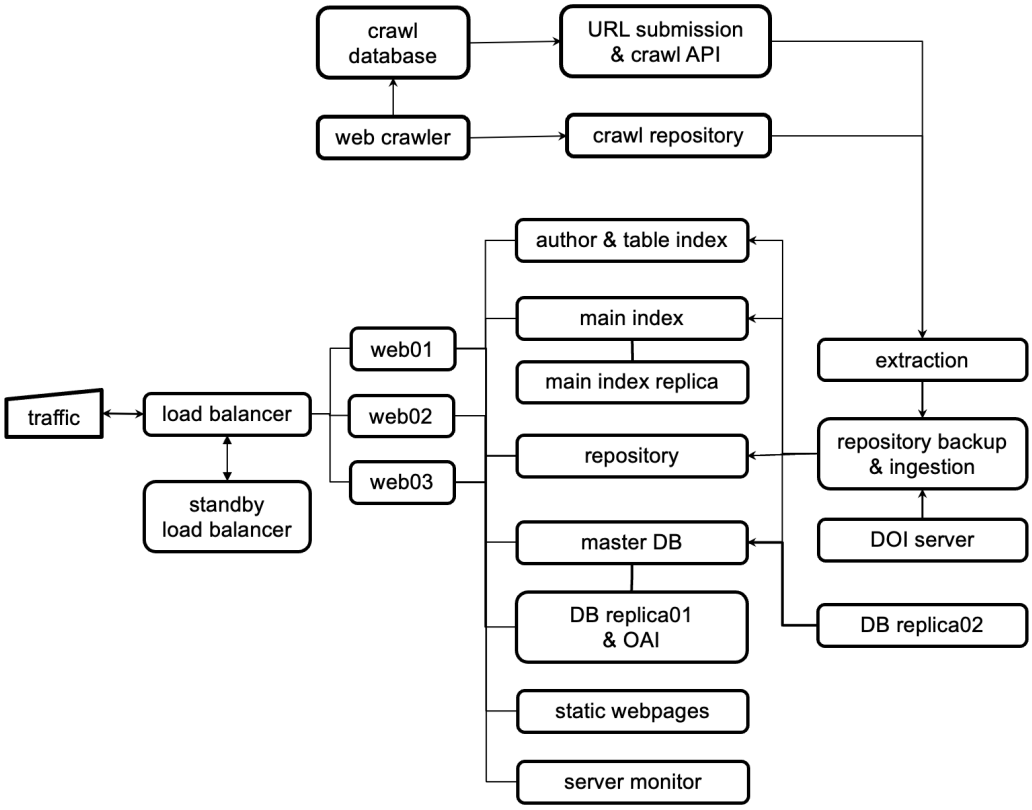

Figure 2.2. Private Cloud Infrastructure of CiteSeerX cluster

metadata), figures, algorithms, keywords from PDF Documents as files which can be used to perform analysis or to build a search engine. PDFMef is a modular and scalable framework and is built to utilize all the cores by multiprocessing. The framework provides simple abstractions that allow users to write wrappers around it to re-purpose it to their needs.

2.2.3 Data Servers & Index

The processed data is ingested into SQL based data servers istcsxdb01 and asynchronously replicated to csxdb02 (albeit, manually). The processed data includes citations, paper metadata, citation graph relating papers and citations. These data is used to fetch the related information while loading paper summary page and citing relationships. The paper data is indexed in servers csxindex01 and csxindex02 that allow searching the full text relevant to user query.

2.2.4 File Repository

The actual PDF files to be served for users to download are stored in servers csxrepository01 and replicated asynchronously to csxrepository02. The servers run XFS File System and a RESTful API that provide access to the PDF documents. The servers use a Varnish [11] cache to serve most frequently accessed PDF documents from memory.

2.2.5 DOI Server

The DOI Server is component of CiteSeerX that assigns unique CiteSeerX specific document ID consisting of 5 parts with each part being a directory. The service is offered through a SOAP API [17]. An example doi looks like 10.1.1.82.1202 with PDF and other metadata location being stored at the directory level of 10/1/1/82/1202/10.1.1.82.1202.pdf

2.2.6 Web Application

The Web application is a Spring Boot Application served by Java Server Pages and Javascript to the the end users through the web browser. The service is hosted in csxweb01 and csxweb02. The service is behind an heartbeatldirectord [10] load balancer that distributes the load and sends the traffic to live server.

2.2.7 Crawling, Extraction & Ingestion Process

CiteSeerX system crawls the web periodically to download PDF files of possible academic interest. Once the PDF are obtained the PDFMef runs a batch extraction service runs an Academic Filter to determine if it should be part of CiteSeerX and extracts metadata and full text from the document. The output is batch processed to be appropriately clustered to detect near duplicate PDF as well as match same citations across different PDF to a canonical entity. While this is a time consuming process, the current system does this in a batch processing with manual steps in between. Once the data has been processed it is ingested into MySQL database and File Repository and the metadata in SQL and full text is indexed by Solr search service. The manual steps involved are described in Figure 2.3

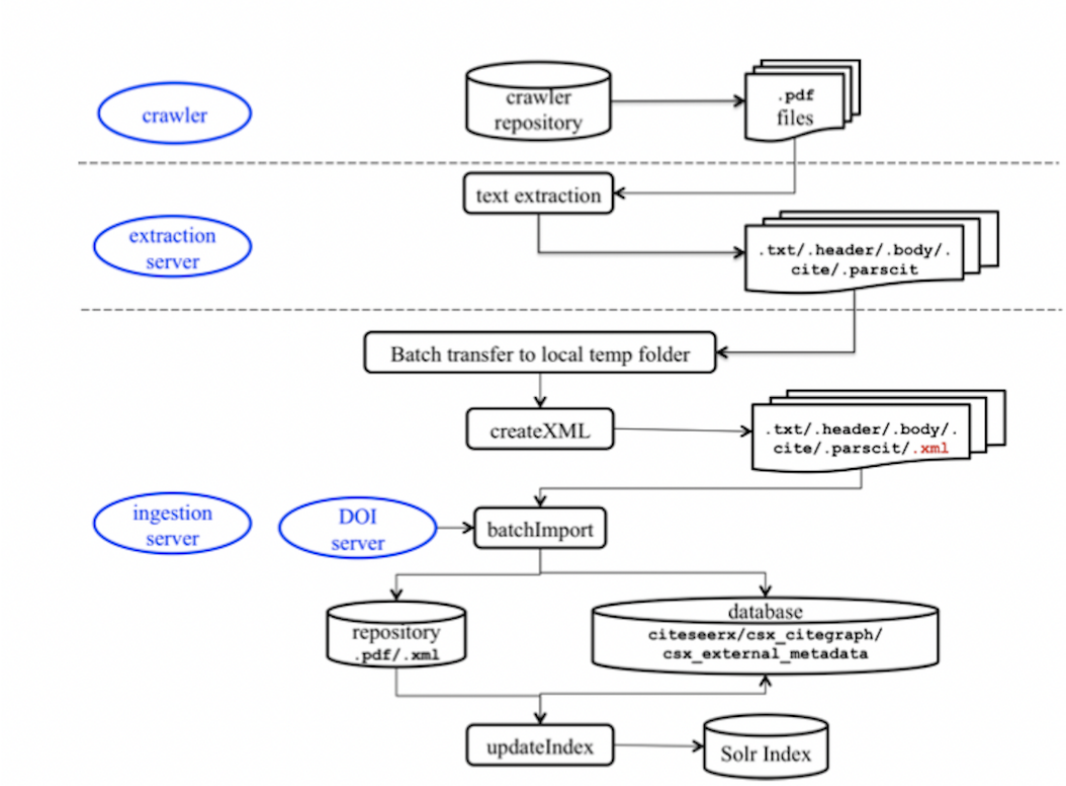

Figure 2.3. Current Extraction & Ingestion Process

2.3 Limitations

While the current version of the system is third iteration since CiteSeerX was originally implemented, the system has been improving upon it's previous versions. The current system still has it's limitations.

The current system has been scaling just well for upto 10 Million PDFs, previous analysis showed that the search response times are increasing, the extraction and ingestion process being batch proocesses aren't achieveing the necessary throughput to ingest more documents as they are available. Apart from this, the code has not been actively maintained and lot of refactoring is required to implement and support newer features including Auto complete, Faceted Search, providing API access to CiteSeerX data.

Scalability: While currently there are about 10 Million documents, ingestion of 20 Million more documents will lead to an estimate of about 250 Million nodes and a billion edges requiring practically impossible Memory per server. The MySQL impedes scalability by posing difficulty in requiring custom sharding of data. So, The system needs

to be inherently be horizontally scalable. With Elasticsearch as a NoSQL data-store one could optimize for both storage and search. The choice of Elasticsearch is justified by [4]

Ingestion Throughput: Figure 2.3 describes the steps involved in current ingestion system, the throughput for which is drastically low with around 5,000 papers a day because of the manual steps involved in crawling, extraction, and ingestion. The clustering process is a sequential and single threaded process which ingests document one after the other to prevent near-duplicate clusters, this is a main factor responsible for slow ingestion pipeline. We address this problem by developing a new parallel ingestion time clustering that clusters near duplicates and matching citations before the documents are ingested into Elasticsearch.

Modularity: The code in web service is tightly coupled with implementation detail of the data servers by directly making SQL queries making it infeasible to reuse components when building a different search engine. We address this issue by creating modular components with specifiable configuration that could point to different component according to the use case. Moreover, the components can be developed independently as long as the API contract remains the same, thus reducing developer communication and driving independent development.

Extensible: The code for the current system is convoluted and requires a lot of refactoring to support newer features. We rewrite most of the code in a modular fashion with newer feature demonstrating the extensibility of the new system.

Fault-Tolerance: The current system needs to be made fault tolerant by frequently taking backup of the SQL data and File Repository data, The web services are tightly coupled with the index and data servers making it difficult to debug and recover in the event of downtime. We address this issue with deploying multiple web servers and creating multiple replica shards for Elasticsearch index, which backs up the data asynchronously and automatically while also improving read and search performance.

2.4 Contribution of This work

Novel index time clustering and near duplicate detection.

- **Elasticsearch Index Design** a novel approach of designining a elasticsearch schema that scales and provides desired functionalities.

- **Improved Extraction & Ingestion System** an Improved extraction and ingestion system that is able to process multiple entities in parallel and ingest into data stores appropriately, the system does index-time clustering to cluster near-duplicates and match citations. The approach scales to extraction and ingestion of about a million documents a day.
- **A Web Application** a web application framework that is extensible, modular for maintaining an academic search engine for foreseeable future.
- **A Load Test** a load test proving the scalability of current system as well as guidance on how to configure the system in production.

Chapter 3

New Architecture Design & Implementation

3.1 Introduction

The new system is designed keeping in mind the limitations of previous system. The idea is to have a system that is modular, horizontally scalable, extensible and reliable.

3.2 Overall Architecture

The new system is backed by multi-threaded scrapy based crawler that downloads about a million documents in half-a-day with 24 processes from reliable academic sources like Microsoft Academic Graph, Semantic Scholar, PubMed etc., and other websites. The PDF file location along with other metadata including source URL, checksums, are stored in a NoSQL index (Elasticsearch in our case) This would be later batch processed by Extraction and Ingestion System to extract and Ingest into search Index and is immediately usable by CiteSeerX users.

The Extraction and Ingestion uses the containerized version of PDFMEF (PDF Multi-entity extraction framework) to extract the text (or metadata), figures, math, key-phrases in parallel. The system updates the success status back to metadata index. The web application interfaces the search index and file repository to provide a clean user interface to end users. The figure 3.1 represents overall architecture

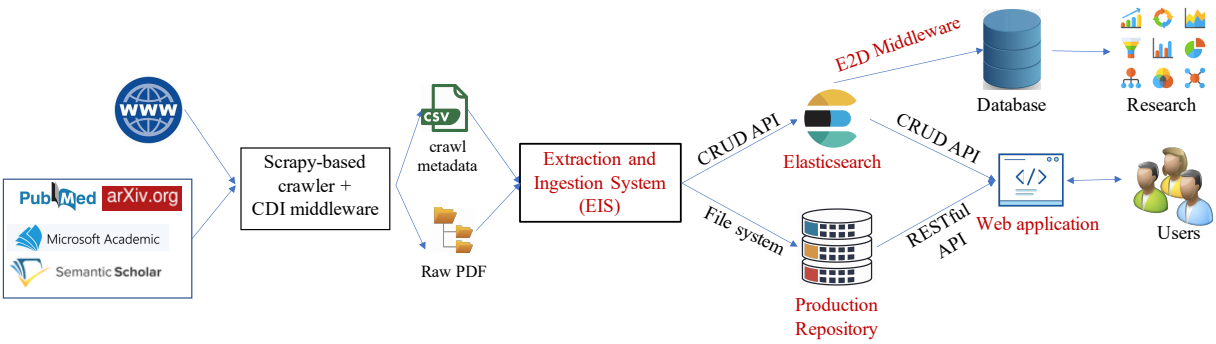

Figure 3.1. The new architecture diagram at a glance

3.3 Elasticsearch Schema

As previous works [3, 4] have established the need for migration to more scalable NoSQL database and Search Engine. The schema has to be designed to meet the goals of CiteSeerX interms of functional and non-functional requirements.

```
{
 "paper_id": ["csxid1", "csxid2"]
 "title": "Title goes here",
 "abstract": "Abstract goes here",
 "has_pdf": true
 "cited_by": ["csxid3", "csxid4"]
 "authors": [
     {"name": "author1", "affiliation": "affiliation1"},
     {"name": "author2", "affiliation": "affiliation2"},
 ]
 "source_urls": ["url1", "url2"],
 "venue": "venue goes here",
 "year": 2015
 "text": "Full text of article if present, goes here"
}
```

Listing 3.1. Single Index Schema of Elasticsearch

The single schema approach gives an advantage in clustering process as well as allowing to index both citations metadata records and full-text papers to be searchable.

The decision to use a single index schema and how the current schema compared to the previous SQL schema is further discussed in section 3.7

3.4 Extraction and Ingestion System

The Extraction and Ingestion system abstracts out the data extraction from PDF and ingesting it to appropriate data stores that servers the web application.

3.4.1 Academic Filter

Since the data downloaded from the web may contain non-academic PDFs, for example legal documents, presentations, privacy policy documents, they need to be filtered out with an Academic Filter. The previous academic filter was implemented as a binary classifier implemented by [2] with features such as file specific features (file size, page count), text specific features (document length, number of words etc.,), section specific features (abstract, introduction, conclusion) and containment specific features with text such as "This work", "This chapter" etc.

We also modified PDFMEF Extraction system to be able to use alternate filters that suits the needs of the data being ingested. When the data is already known to be academic, we could implement a simpler version of the academic filter. Therefore, we implemented a PyPDF based simple academic filter that is fast and could work well if the documents being ingested are mostly academic, unlike previous filter which needs to run a classifer to determine a PDF as academic or not. While this not only improves the ingestion throughput by being efficient, it improves the recall.

PyPDF filter is inspired from [5] The filter checks for number of pages and discards those that produce error while processing with PyPDF or that have less than 2 or more than 50 pages and those that have greater page width than height (PPTs).

3.4.2 PDFMEF

We made changes to existing PDFMEF [8] to port it to Python 3, containerize it to ease the setup. We added an additional step of ingestion after extracting metadata. Currently we are using just GROBID but the system could be easily extended to run Math, Text, Figure extraction in separate containers to ingest to Elasticsearch in parallel.

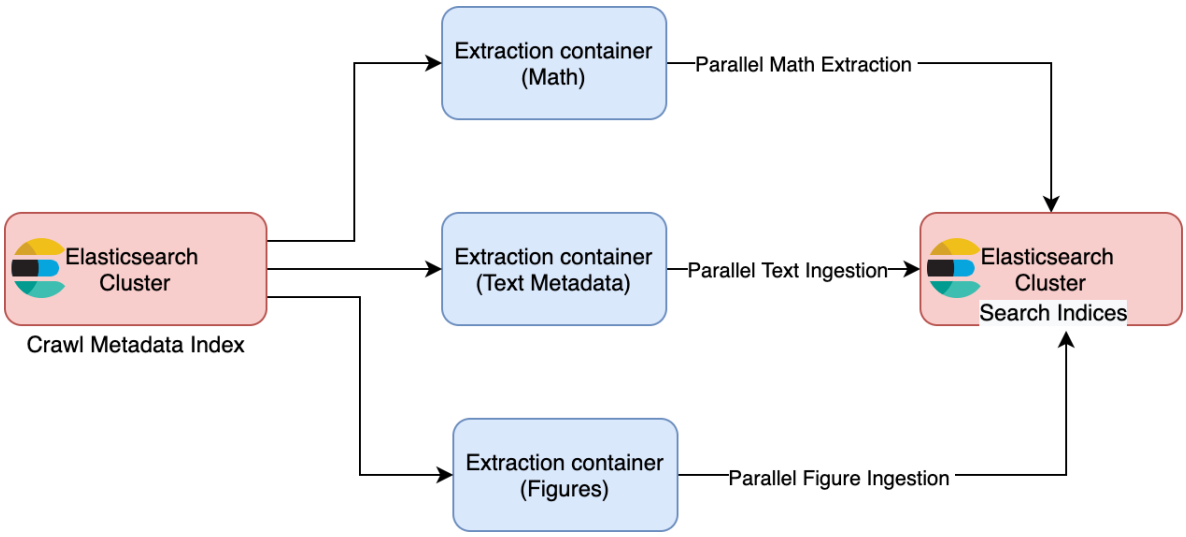

Figure 3.2. Containerized version of PDFMEF that could ingest in parallel

The figure 3.2 depicts how a version of PDFMEF could be deployed to run on separate machines to parallely ingest paper related metadata

3.4.3 Clustering & Metadata Ingestion

Near duplicate detection in academic papers is well studied problem [5–7, 9], while they are domain specific [7] and are based on minhash or other approaches that are not built for online processing [6]. Our system introduces a database-backed aproach, allowing us to evaluate incoming documents against keys of of millions of previously-processed documents. While our algorithm largely implements [6], we make it work online and add an additional step at the end that calculates actual overlap between two potential near-duplicate documents, thus reducing false positives and improving clustering quality.

Academic documents with PDF files contain full text but citations do not. Therefore, we define a cluster to be a distinct bibliographic unit that contains either or both of them as long as they refer to the same paper. Clustering in the context of a DLSE repository is the process of identifying and grouping near-duplicate paper records and its citations in other papers. We implement the key mapping algorithm [6] as an online and parallel process to make ingestion near real-time. The idea is to index the keys generated by concatenating title snippet and author names and use these key to match documents. When ingesting a paper its keys are compared with existing keys to retrieve candidate clusters using Elasticsearch's GET-by-id API. Doing so enables parallelization

of the algorithm while also significantly reducing the search space during matching. This gives an enormous boost in extraction and ingestion throughput of about one million documents a day. The candidate clusters are further matched by using similarity metrics from [5] before merging the entity into a cluster. The similarity metric S_{title} is obtained as harmonic mean of Jaccard index (J) and containment measure (C):

$$S_{\text{title}} = \frac{2JC}{J+C}, \quad J = \left|\frac{N_1 \cup N_2}{N_1 \cap N_2}\right|, \quad C = \frac{|N_1 \cap N_2|}{\min(|\ N_1\ |, |\ N_2\ |)}$$

where N_1, N_2 are normalized n-grams ($n = 3$) of the current title and matched cluster title respectively. The similarity threshold is empirically set to 0.6.

The clustering process is described in Figure 3.3

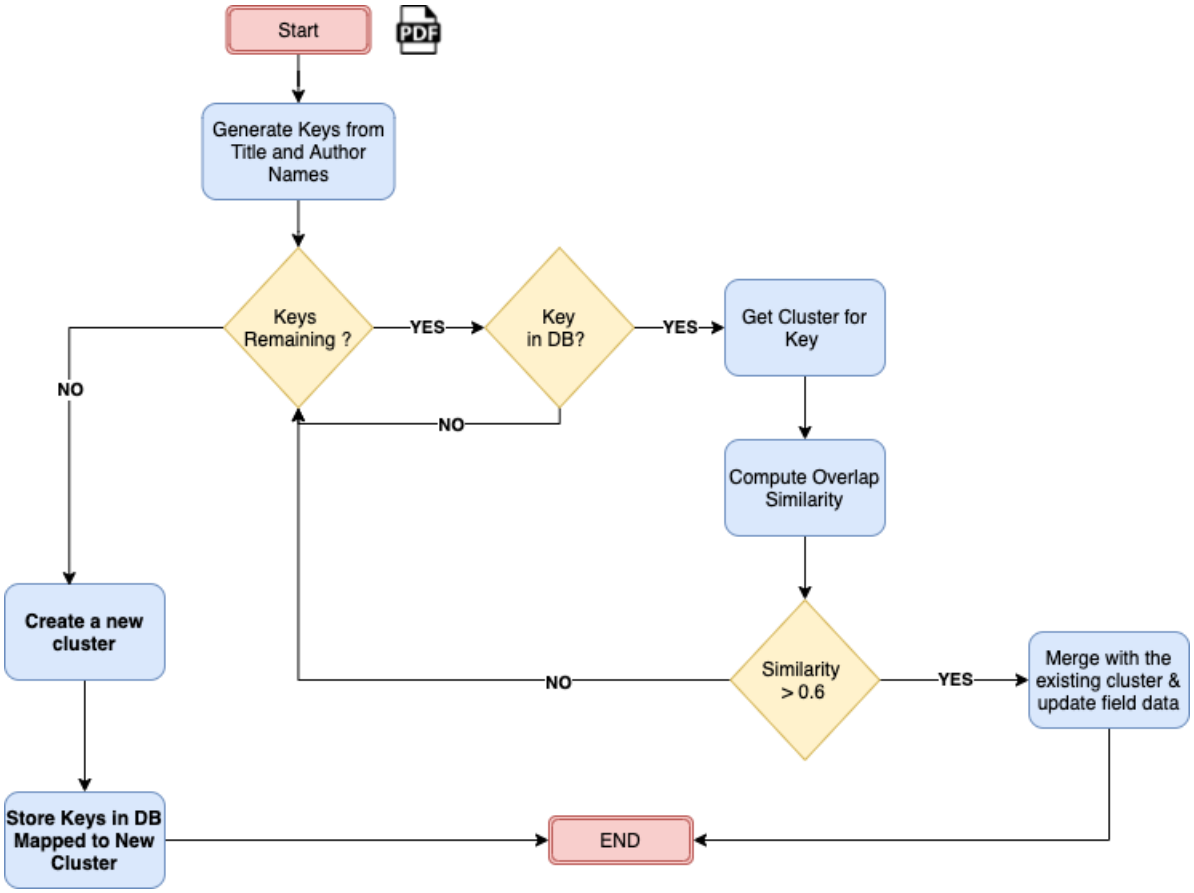

Figure 3.3. Parallel Clustering Algorithm described as a flowchart

3.4.4 PDF Document Ingestion

The paper data and citation data are converted into entities consumable by the Elasticsearch API. We use the single index approach, where each indexed document is a document cluster that holds both metadata and citation relationships, so we can easily find out its citing and cited clusters. The PDFs are then renamed by their SHA1 values. We store them in the repository server by nesting them by name. For example, a PDF with SHA1 `a7b98ea0f94b920920524cdeee142232d7ccc488` is stored at location `/data/a7/b9/8e/a0/f9/4b/92/a7b98ea0f94b920920524cdeee142232d7ccc488/a7b98ea0f94b920920524cdeee142232d7ccc488.pdf`. These PDF resources are accessible through a REST API that is served for end users via the search interface

3.5 Web Server

The web server provides REST API to systematically access the Elasticsearch data consisting of papers, citations, and clusters. Additionally, it provides REST API that is consumed by the Front-end User Interface to help user navigate through academic papers via the search Engine as well as functionality that enhances user experience like Auto Complete, Faceted search etc.

The following table lists the API provided by the webserver

Table 3.1. REST API of Web Server interacting with Elasticsearch

API Name	API Path	API Parameters	API Response
search	/api/search	Search query	Paginated List of Papers for SRP[a]
paper	/api/paper	csxId	Paper details with given id
citations	/api/citations	csxId	Citations for paper with given csxId
cluster	/api/cluster	clusterId	get cluster details
showCiting	/api/showCiting	clusterId	papers citing the given cluster
suggest	/api/suggest	search string	auto-complete paper suggestion
similar	/api/similar	csxid	papers similar to current paper

[a]Search Results Page

We chose to separate front-end and web server that provides back-end so that the system is modular with each component can be swapped out with a different implementation as long as the contract remains intact.

3.6 Repository Server

Repository Server is responsible to provide API that helps ingestion of a PDF document as well as viewing and download of the same. While the current implementation only provides the view/download functionality, it can be easily extended to provide an interface a file upload API for ingestion into the file repository. Having a REST API would allow the service to be independently modified as long as the contract remains same.

The REST API provided by Repository server has the path `http://<baseUrl>/api/document?doi=xxxx&type=pdf` where doi is the paper id of the corresponding paper to which the pdf is required. We used SHA1 of the paper as doi.

3.7 Discussion on Special Issues

3.7.1 Single Index Schema

In the current system's database, papers, citations, and authors are stored in separate tables. Although it is intuitive to make separate indices when transfer data from MySQL to Elasticsearch, it is not the most efficient way because the system still needs to jointly query different indices to obtain certain results. To take advantage of the fast searching performance of Elasticsearch, we organize all data into a single index depicted in listing 3.1. The advantage of this design enables us to store both full text papers and citations together in form of clusters. Since the search is performed on clusters, the Search Results Page (SERP) will not include near-duplicates.

The citation relation information is stored in the `cited_by` field that contains list of paper ids cited by the current cluster. To retrieve citations for a given cluster we retrieve all clusters that contain the current `paper_id` in its `cited_by` field.

The advantage with the new schema is that, we could index and search metadata records of the papers that are in the form of citations and do not necessarily have full text or PDF in the corpus.

3.7.2 Modularity

The Web Application and Ingestion modules are modular, meaning that both can be easily configured to work with a different index as long as the metadata schema remains the same. This is made possible by hosting the web service separately and allowing administrators to configure which index to fetch data from. Similarly, the Ingestion system can be configured to ingest data into different Elasticsearch index and File System repository. This enables creating multiple search applications of different corpora with same code repository.

Since the repository server is modular too, unlike the current system, the new system with search functionality would work even when repository server is down.

Moreover, The Web Application itself has modular components for Frontend and Backend, with Backend being written in FastAPI [12] interacts with Frontend written in Vue.js [13] via REST API.

3.7.3 Parallel & Concurrent Ingestion

The new Extraction & Ingestion System (EIS) improves upon the previous one by making extraction and ingestion concurrent. The latencies of extracting entities such as Math, Figures are much higher than text metadata and would take considerably huge amount of time. To overcome this, we could run multiple containers of the extraction service in different servers, thus, using resources effectively and ingesting data in parallel up to the limits of elasticsearch cluster.

Also, the clustering (& near-duplicate detection) happens during ingestion and in parallel helping improve throughput, as mentioned earlier this was achieved by key map algorithm and elasticsearch real-time GET by Id API to retrieve stored key to cluster mapping.

3.7.4 Paper Identification & Duplicate Prevention

Since CiteSeerX is a crawl-based digital library unlike other submission-based search engines, the system does duplicate and near-duplicate docuemnts as mentioned in [9] CiteSeerX uses SHA1, a cryptographic hash value, to discard direct duplicates of the PDF documents, the idea is to use them to uniquely identify papers as well, this way we could avoid an additional DOI Server and also consistent digital identifier as that of Semantic Scholar and other search engine.

As mentioned previously, the new identifier not only helps in uniquely identify a document, but also encodes the file's location path in the repository as well. Moreover, we realized that Internet Archive's Fatcat Scholar [15] and Semantic Scholar [14] Semantic Scholar's S2 Corpus also use the same identifier, i.e., SHA1 to identify the documents in their corpus, thus we could easily compare the extracted metadata with that of theirs to get any missing metadata and thus producing cleaner metadata.

Chapter 4
Evaluation & Results

4.1 Introduction

We have built a prototype system that meets the desired requirements mentioned elsewhere in this Thesis. The alpha version of the system is deployed in the local servers and deployment of pre-production version is in works with about a million documents. At the time of this writing, The production servers have been ordered and are yet to be setup.

We perform a few load tests to show that the system meets the desired performance goals with infrastructure that is already available.

4.2 Ingestion Throughput

The current system uses batch extraction followed by a batch ingestion, the new system improves this by making them concurrent. Also, the system is made to extract, cluster and ingest data to elasticsearch in parallel to improve the ingestion throughput.

Compared to current system that takes roughly about a week to extract and ingest 100,000 documents a day, we improve this enormously to extracting and ingesting about a million documents per day.

As mentioned earlier, the extraction is a CPU intensive process and could benefit from multi-core processor, and the ingestion is a network intensive operation boosted by threading, we measure ingestion time on a machine with AMD EPYC 7702P 64-Core Processor with 2 threads per core CPU clocked at 2000 MHz to find time taken for various

configurations of number of processes, threads and documents per batch to ingest 22,817 documents from ACL corpus. The table 4.1 summarizes the indicating that the system benefits from more number of documents, threads and processor until Elasticsearch cluster becomes a bottleneck throwing connection refusal exceptions, i.e., beyond 128 processes when significant number of requests fail and have to be retried causing time to complete increase, for example a case shown for the case when number of processes is 256.

Table 4.1. Ingestion throughput for various configurations

Number of Processes	Number of Threads	Batch size	Time in seconds
128	1000	1000	2228.74s (37.15 min)
128	1000	500	2385.90s (39.76 min)
128	10	1000	3040.27s (50.67 min)
128	10	100	3995.45s (66.59 min)
128	20	1000	2357.28s (39.29 min)
50	1000	1000	2278.83s (37.98 min)
30	1000	1000	2482.30s (41.37 min)
128	128	1000	2262.89s (37.71 min)
256	1000	1000	2462.52s (40.44 min)

4.3 Load Testing

Elasticsearch is configured to run on a cluster of two nodes, namely, csxindex05 and csxindex06, each with 8 cores Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz. While the data is ingested via the machine lrs-giles05 that is equipped with AMD EPYC 7702P 64-Core Processor with 2 threads per core and CPU clocked at 2000 MHz.

The data is full text academic papers and their citation clusters that amounts to a size of 45.8 GB with one primary shard, and a total of 1 Million full text documents on machines csxindex05 and csxindex06.

To gauge the scalability of our system we perform a load test using Locust framework. Locust simulates desired number of concurrent users and their system usage by making API calls to the back-end while the number of failures as well as response times. This way we get an estimate of number of servers, and configuration (number of shards and replicas) required in production. Elasticsearch scales horizontally. The system's configuration that

includes number of nodes and number of shards and replicas should be set according to the desired scale according to usage pattern and number of concurrent users at anytime. We prove that our system scales sufficiently with even the existing infrastructure.

We perform 2 experiments, one by simulating 2000 users and other 100 concurrent users by incrementally adding 10 users per second, while the former allows us to gauge the maximum tolerable scale of the new system until it breaks, the latter allows us to benchmark the number of shards required to scale the system to desired latencies and performance.

The figure 4.1 shows the results of simulating 2000 concurrent users. Failures start at around 785 concurrent users with all requests failing for the experiment with 2000 users. The median response time stays within 1 ms, however the 95%ile and failed request suffer large latencies of more than 3 seconds when the number of users exceed 785.

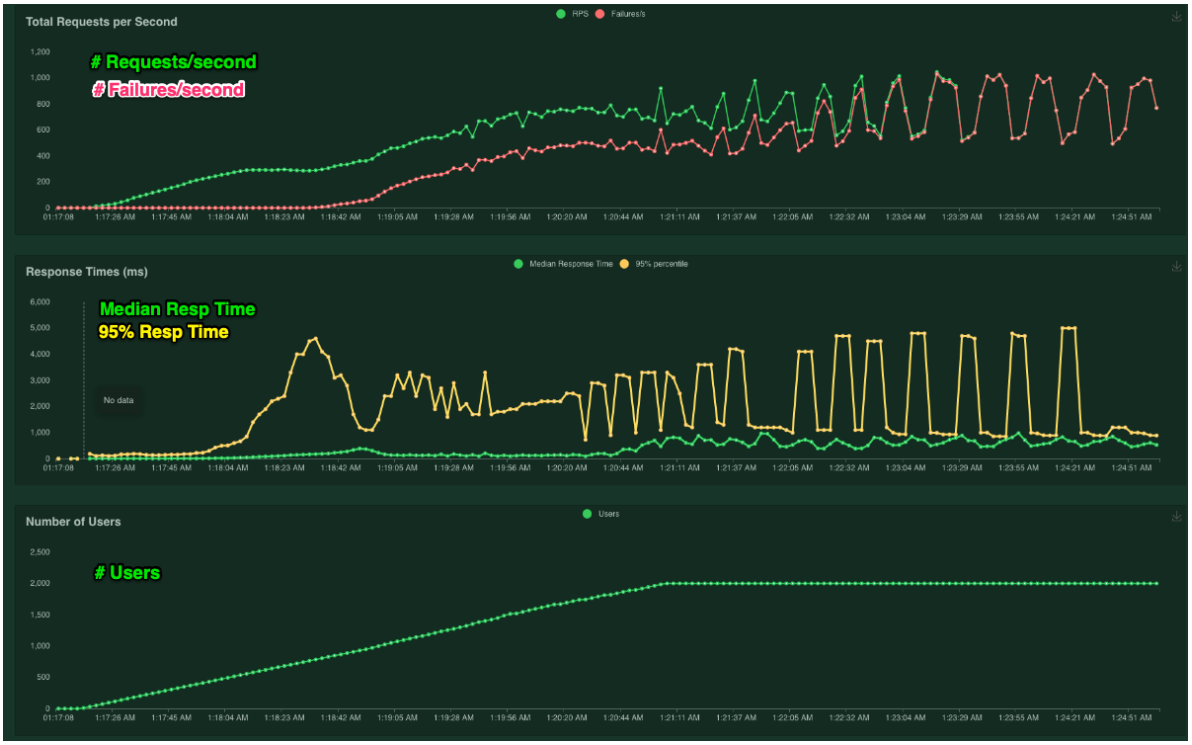

Figure 4.1. Load Test results when simulating 2000 users at the rate of 10 users spawned per second, x-axis denotes clock time in minutes

4.4 Benchmarking

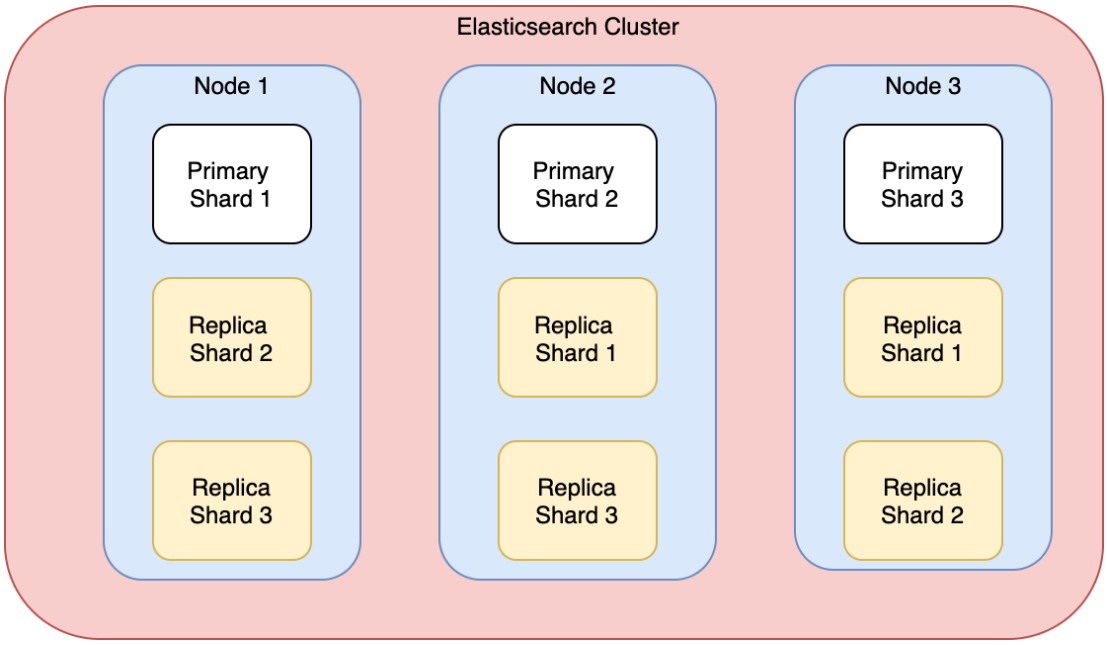

Figure 4.2. An overview of how shards and replicas are distributed across nodes in Elasticsearch

Data is organized in indices and each index which in-turn points to one or more shards and each shard is a single instance of Lucene. Shards are automatically relocated by Elasticsearch to spread the load. planning ahead by over-allocating number of primary shards helps in scaling by taking advantage of additional computing power when new nodes are added while not having any downtime since the migration is done by simple copy operation by Elasticsearch.

An Elasticsearch shard can be a primary shard or a replica of a primary shard. To allow scaling the cluster horizontally and appropriately we need to configure the number of primary shards before ingesting documents as it is cannot be changed afterwards. Also, having more replica shards which can be configured live improves search throughput provided the number of nodes are increased too. The number of primary shards dictate maximum amount of data that can be stored in an index

To build the system for the desired scale into the future, we need to determine the number of primary shards that are appropriate for the specific use case well in advance of deploying the application in production. Elasticsearch community recommends benchmarking, we perform the following experiment for the same

- We create an Elasticsearch cluster consisting of a single server with the hardware similar to the one we intend to use in production.
- We create a test index with same settings as that of in production with only single primary shard and no replica shards.
- We fill it with real documents and run real queries and simulate the user behaviour via load testing with about 100 concurrent users.
- We try to ingest as many full text documents (and corresponding clusters) as possible until the response times or tend to increase significantly (say >200ms)
- The above step determines the capacity of a single shard, we then extrapolate that number to your whole index to get the total number of primary shards we need.

The results for the case of 100 concurrent users are shown in the Table 4.2 It is observed that the system scales well for the load

Table 4.2. Summary of load test results for 100 concurrent users and no failures

Name	Request Count	Median Response (ms)	Average Response (ms)	Min Response (ms)	Max Response (ms)	Requests /s	50%	66%	99.99%
Download	948	17	25.96	3.89	254.15	5.42	17	23	250
Paper Summary	994	6	8.38	3.28	326.53	5.68	6	7	330
Search	917	47	152.21	3.32	946.28	5.24	47	140	950
Citing Papers	999	11	18.01	3.21	748.52	5.71	11	12	750
Similar Papers	1004	5	5.68	2.86	19.30	5.74	5	6	19
Suggest	949	8	27.70	3.47	442.27	5.42	8	9	440
Citations	966	7	11.96	2.73	295.61	5.52	7	7	300
Aggregated	6777	8	34.54	2.73	946.28	38.72	8	11	950

Table 4.2 summarizes the average, median, minimum, and several percentile responses for the slowest queries, it is observed then when a shard is configured with about a million full text documents (around 15 Million clusters including citations) the system scales well before the latency increases further, the data in the single primary shard is around 40GB, while this experiment is an extreme benchmarking test that do not use replica shards, the results prove that the system scales even better by planning in advance with replica shards that reduce response time and improve search throughput even further.

Chapter 5
Future Work

While we were able to achieve the goals we initially set out by improving the current system to build a performant and scalable system, the path forward is endless. Future goals of the system is to further improve upon the prototype we built and get it deployed on production to be on par with latest DLSEs while still retaining the principles of the existing CiteSeerX system. We briefly describe our planned future work that extends the current work.

5.1 Author Disambiguation and Search

As of now, we do not have an Author search functionality that allows users to search for details or papers published by a certain Author, we need to do a clustering or a disambiguation process either as part of ingestion or as a post-ingestion step to assign unique author identifiers for authors.

5.2 Application Layer Caching

Currently, we don't have any cache at the application layer to store most frequently and recently queried results. We can easily deploy Varnish [11] for the File System Repository similar to the current system. However, this is something that can be easily integrated with existing implementation based on the user traffic and latency requirement.

5.3 Metadata Correction

While we do not plan on providing direct edit access for user to metadata, we intend to input user's feedback and store it in an index to later either approve manually or to run in batch mode after making sure the edits are legitimate.

5.4 Deletion of Paper

Publisher's might want their papers to be removed from the Index, such requests have been common in the past and we would like to implement the same in the next generation CiteSeerX as well. For this we intend to provide a REST API for administrator to provide a paper id of the paper to be deleted which internally clears of the cluster, citations, and key map associated with that paper. This can be further extended by writing a wrapper API that could be called it in bulk.

5.5 Asynchronous processing for EIS

The new Extraction & Ingestion System (EIS) improves upon the previous one by making extraction and ingestion concurrent. However, the new system is still synchronous per PDF, meaning that the extraction thread is blocked until ingestion is complete thus having some inefficiently in using resources, and there is no way to efficiently retry failed ingestion. Having a Kafka queue would help in load balancing ingestion to elasticsearch, enables multiple retries for data that isn't ingested due to network or concurrency issues.

The Figure 5.1 depicts a possible architecture where document ids are pushed into Kafka queue that could be processed asynchronously.‘

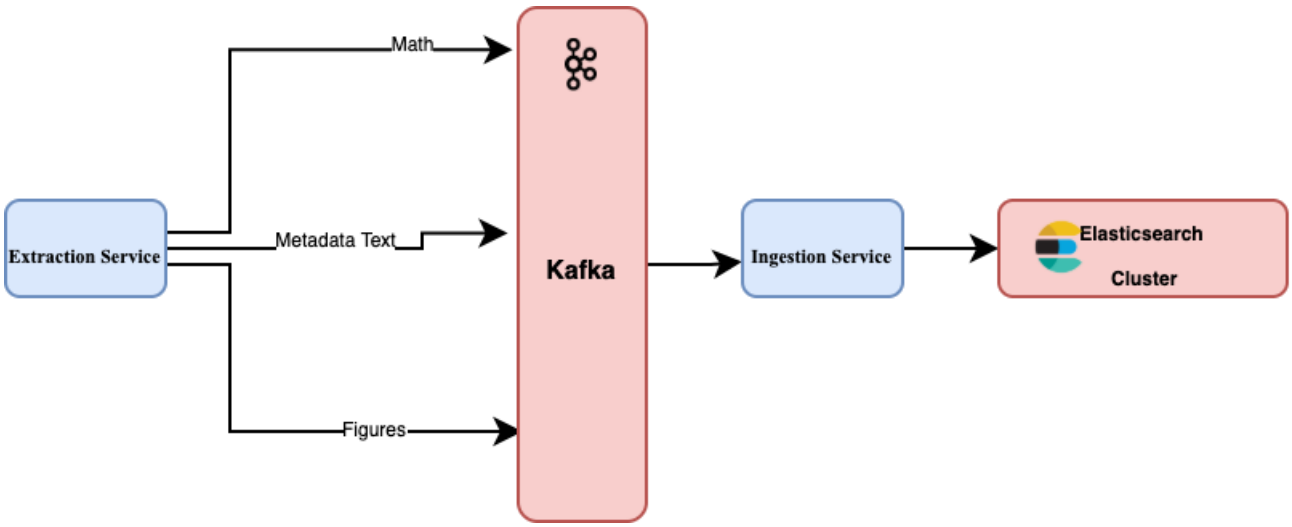

Figure 5.1. Plan depicting an Asynchronous EIS pipeline

5.6 MyCiteSeer

The system can be extended to authenticate users by hooking the application with third party login service like Google Sign-in. Features such as bookmarking and rating articles can be easily integrated with the system making it more futuristic and user-friendly.

Chapter 6
Conclusion

We have made a first step towards next generation version of Extraction and Ingestion system and a DLSE prototype. We could improve upon the previous version in terms of scalability and performance by allowing system to scale horizontally with NoSQL based Elasticsearch. We improved the document Ingestion throughput with a novel clustering and ingestion approach to about a million full-text PDF documents a day along with its citations. Also, we developed a modular system with separate components for web server (backend & frontend), a repository server, a new way of identifying and storing documents with their SHA1 ids.

We believe the current work and the unique approaches we used could help researchers in DLSEs to apply and advance techniques in Information Retrieval specifically for Scholarly Big Data.

The future goal of the Next Generation CiteSeerX project is to document and make the system available in production for millions of users. We estimate to scale the system up to about 40 Million full text academic documents and their citation records that are concurrently accessed by about 100 users at any time.

We also deploy a prototype with a subset of academic documents in internal servers for research and development purposes with goal of building a Math equation search system.

Appendix A
Code & Documentation

We include further resources and useful links for future us to take the project forward.

- The documentation (still in construction as of this writing) is provided in Penn State's Confluence service `https://wikispaces.psu.edu/display/CT/CiteSeerX+Internal+Documentation`
- The web service (with frontend and backend) is committed to dev branch of private repository `https://github.com/SeerLabs/next-gen-citeseer`
- Extraction and Ingestion System is integrated with existing PDFMef and is available in dev branch of the repository `https://github.com/SeerLabs/pdfmef`

Appendix B
User-Interface Preview

We show a preview of front-end User Interface of the prototype which uses the Web server APIs described in Web Service section. The front-end is developed by fellow teammates Jason Chhay and Kevin Kuo.

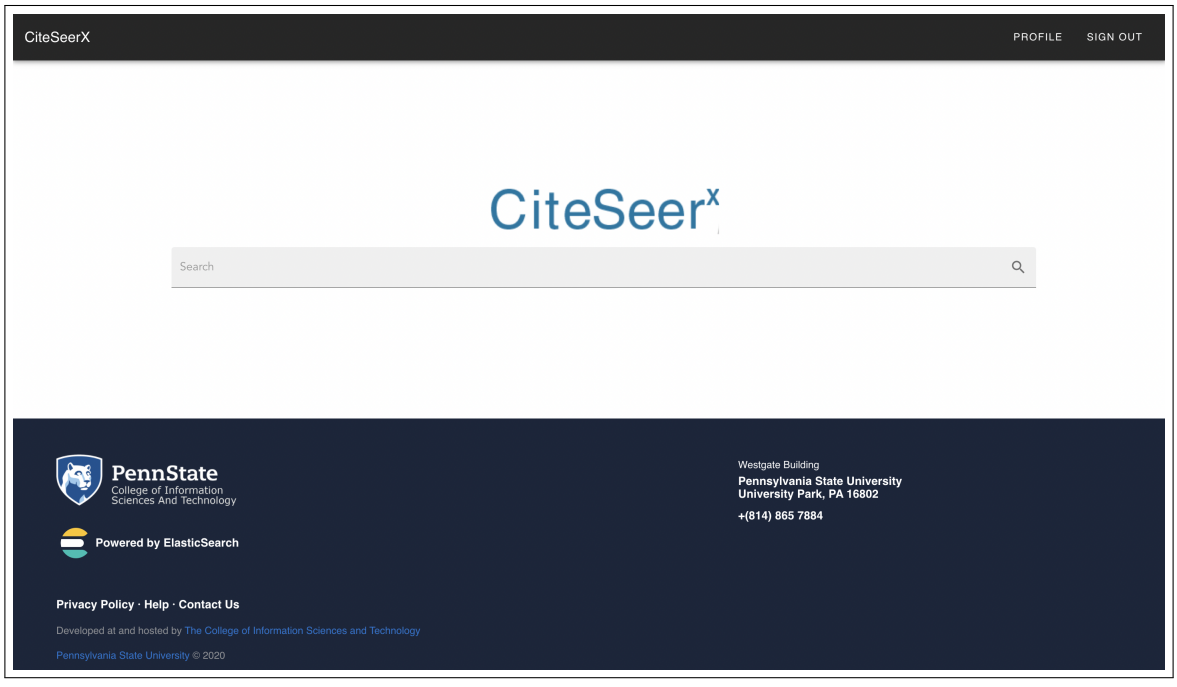

Figure B.1. Home Page

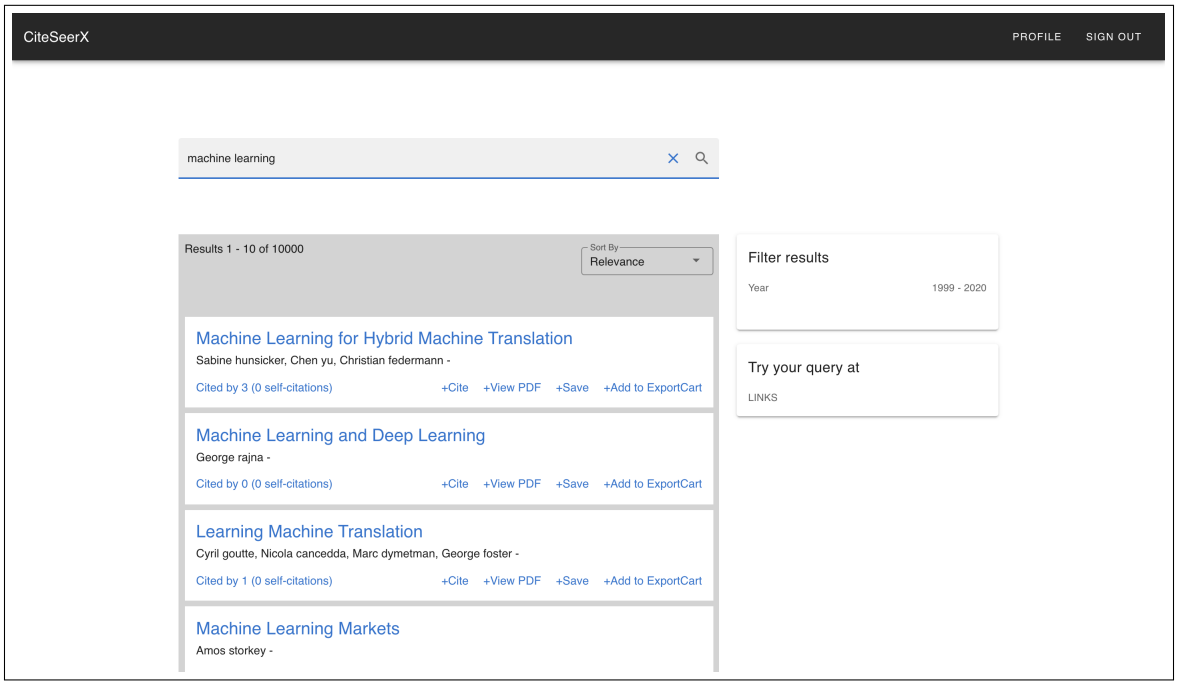

Figure B.2. Search Results Page

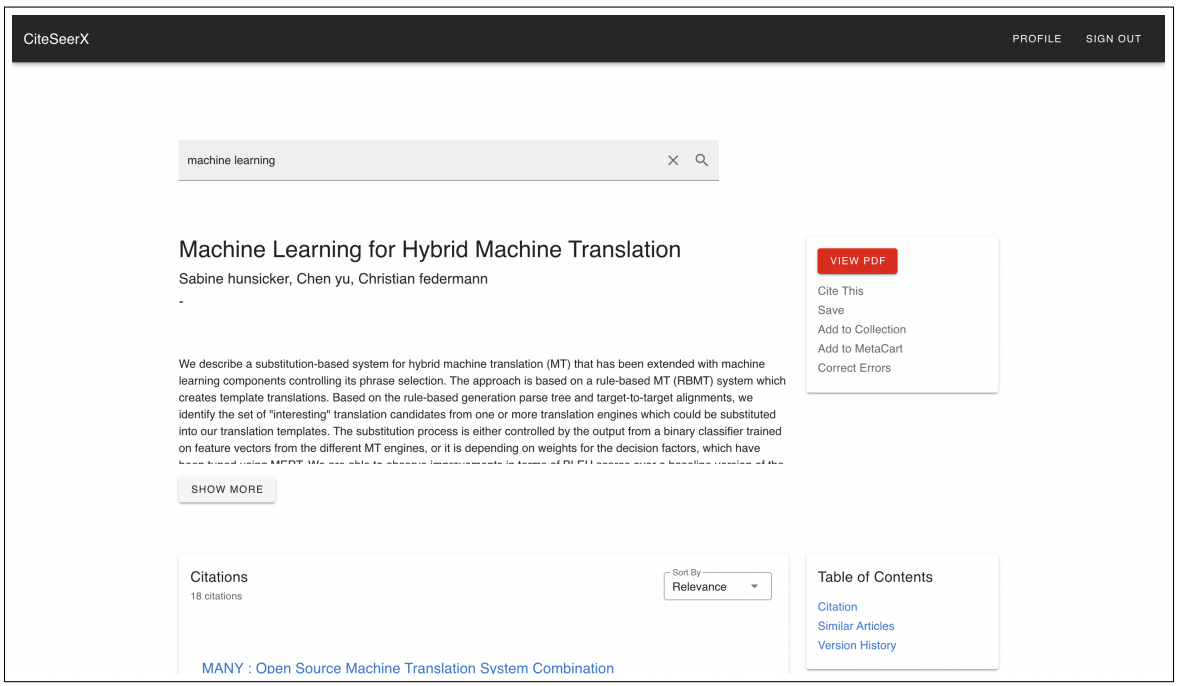

Figure B.3. Document Summary Page

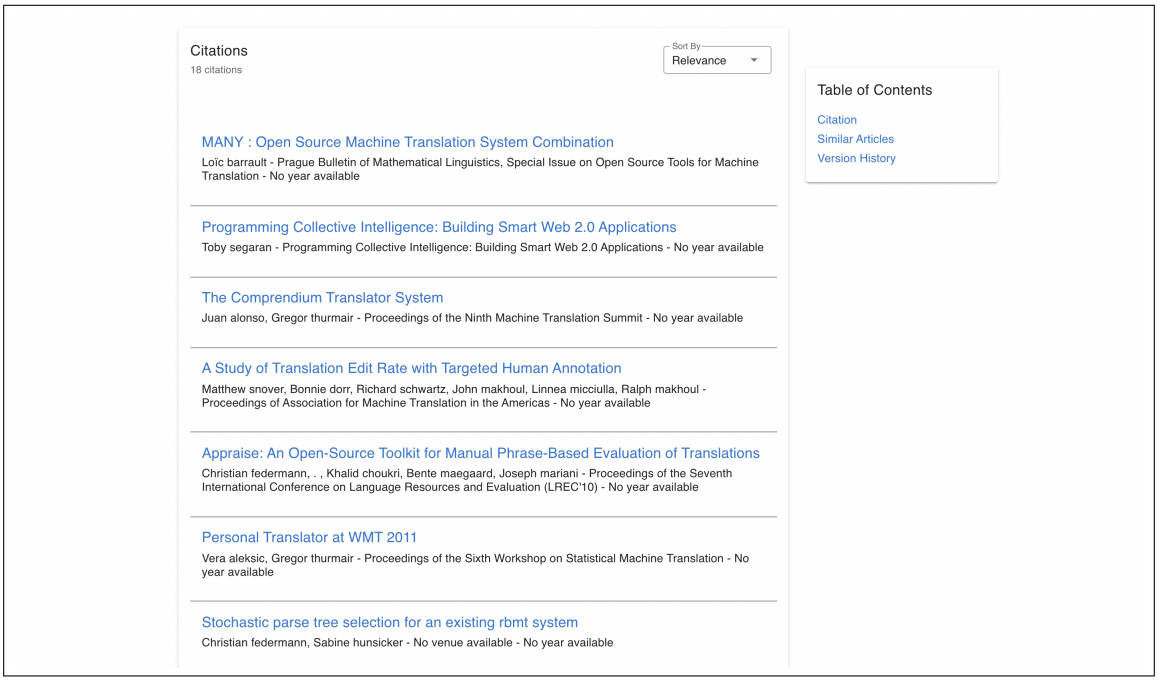

Figure B.4. Citations section of Summary Page

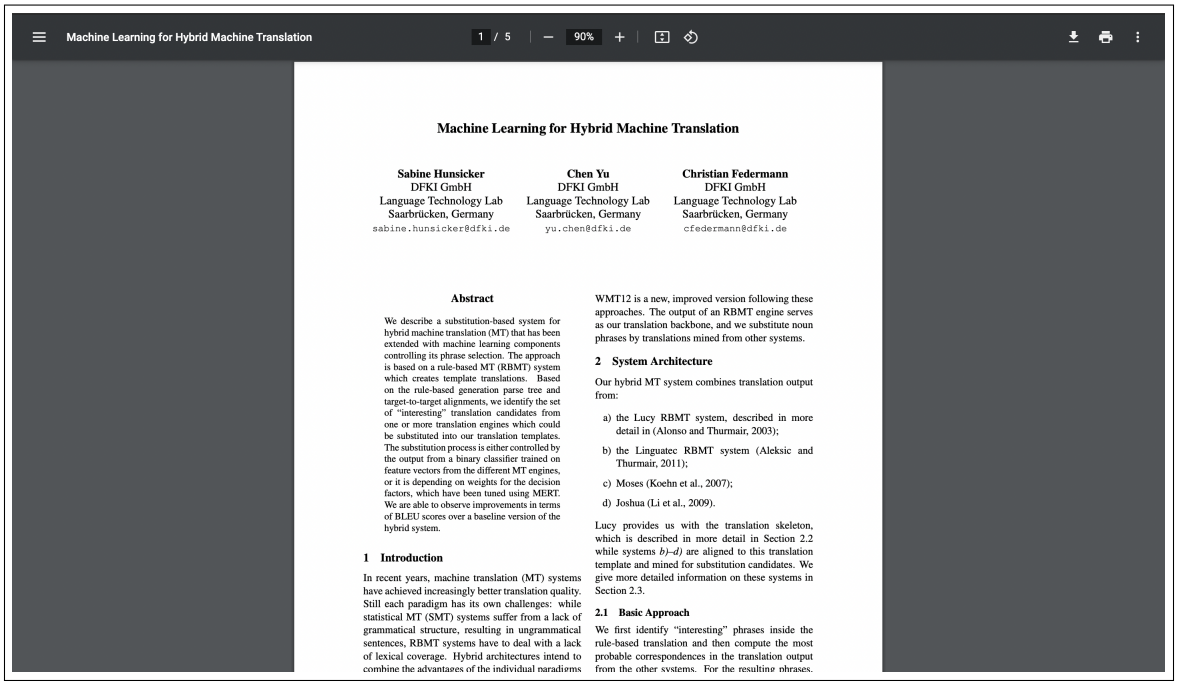

Figure B.5. PDF View Page

Bibliography

[1] WU, J and LIAN, C. and YANG, H. and GILES C. L. (2016) *CiteSeerX data: semanticizing scholarly papers*, SBD'16.

[2] CARAGEA, C.; Wu, J.; WILLIAMS, K.; GOLLAPALLI S. D.; KHABSA, M.; and GILES, C. L, (2016) *Document Type Classification in Online Digital Libraries.* AAAI

[3] JORDAN, D. W (2016), "Lessons In Scaling A Large Digital Library: A Case Study For Citeseerx," *Master's Thesis*,
URL `https://etda.libraries.psu.edu/catalog/29174`

[4] PARSONS, SEAN (2020), "The Migration of Data and Refactoring of Large Scale Digital Libraries: A Case Study For CiteSeerX," *Master's Thesis*,
URL `https://etda.libraries.psu.edu/catalog/17831swp5504`

[5] KYLE L. and LU L. WANG and M. NEUMANN and R. M. KINNEY and D. S. WELD, (2020) *The Semantic Scholar Open Research Corpus*, ACL.

[6] WILLIAMS K. and GILES C.L (2013) *Near Duplicate Detection in an Academic Digital Library* Proceedings of the 2013 ACM symposium on Document engineering

[7] RODIER, S. and CARTER, D. (2020) *Online Near-Duplicate Detection of News Articles* LREC, Proceedings of the 12th Language Resources and Evaluation Conference.

[8] WU, J and KILLIAN, J and YANG H and WILLIAMS K and CHOUDHURY, S. R. and TUAROB, S and CARAGEA, C and GILES C. L., (2015) *PDFMEF: A Multi-Entity Knowledge Extraction Framework for Scholarly Documents and Semantic Search*, Proceedings of the 8th International Conference on Knowledge Capture.

[9] WU, J and WILLIAM, K. and CHEN H and KHABSA, M and CARAGEA, C. and TUAROB, S. and ORORBIA A. G. and JORDAN, D. and MITRA, P. and GILES C. L. (2014) *CiteSeerX: AI in a Digital Library Search Engine*, AAAI.

[10] Software "ldirectord, Heartbeat," .
URL `https://www.suse.com/c/load-balancing-howto-lvs-ldirectord-heartbeat-2/`

[11] Software, V., "Varnish Cache," .
URL https://www.varnish-cache.org/

[12] Software, "FastAPI," .
URL https://fastapi.tiangolo.com/

[13] Software, "VueJS," .
URL https://vuejs.org/

[14] Semantic Scholar .
URL https://www.semanticscholar.org/

[15] Fatcat Scholar.
URL https://fatcat.wiki/

[16] Apache Solr
URL https://solr.apache.org/

[17] Soap UI
URL https://www.soapui.org/learn/api/soap-vs-rest-api/