THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CREATING A NEW AND ENHANCED CONTENT AND METADATA EXTRACTION SYSTEM FOR CITESEER$^{\chi}$

JASON P KILLIAN
SPRING 2015

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

Reviewed and approved* by the following:

C. Lee Giles
David Reese Professor at the College of Information Sciences and Technology
Thesis Supervisor

Jesse Barlow
Professor of Computer Science
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

One challenge in the world of digital libraries is obtaining accurate data and metadata from sources where this information is not easily digitally accessible. In the case of CiteSeer$^{\chi}$, a digital library search engine of scholarly documents, this challenge manifests itself in extracting metadata and content from scholarly PDF files. CiteSeer$^{\chi}$'s current extraction system responsible for this is functional but could be improved in many areas. This thesis will present the architecture and design of a new extraction system that fixes the issues of CiteSeer$^{\chi}$'s current extraction system.

Table of Contents

List of Figures **iii**

List of Tables **iv**

Acknowledgements **v**

1 Introduction **1**
1.1 Background 1
1.1.1 CiteSeer$^{\chi}$ Overview 2
1.1.2 Content and Metadata Extraction 3
1.2 Objective 3
1.3 Outline 4

2 The Current CiteSeer$^{\chi}$ Extraction System **6**
2.1 Design 6
2.2 Discussion 9

3 Improving the CiteSeer$^{\chi}$ Extraction System: A First Attempt **11**
3.1 Design 11
3.2 Discussion 13

4 A New CiteSeer$^{\chi}$ Extraction System **14**
4.1 Extraction Framework 14
4.1.1 Objectives 14
4.1.2 Design 15
4.1.3 Implementation 17
4.1.4 Discussion 18
4.2 CiteSeer$^{\chi}$ Extraction System 19
4.2.1 Design 19
4.2.2 Discussion 22

5 Conclusion **23**

Bibliography **24**

List of Figures

1.1 CiteSeer$^{\chi}$ architecture overview 2

2.1 CiteSeer$^{\chi}$ extraction process 7

3.1 CiteSeer$^{\chi}$ revised extraction process 12

4.1 Black box diagram of the extraction framework 15
4.2 Levels of extraction framework parallelization 16
4.3 High-level design of the new CiteSeer$^{\chi}$ extraction system 20

List of Tables

2.1 Schema of the `main_crawl_document` table in the crawl database 8
2.2 Meanings of the `state` field in the `main_crawl_document` table 8

4.1 Time elapsed for data extraction on 100 PDF files 22

Acknowledgements

It was an honor for me to have the chance to work with so many fantastic people during my thesis research. I would like to first thank Lee Giles, my primary thesis adviser, for supporting me throughout my work. It has been an absolute pleasure to work with him and everyone on the CiteSeer$^{\chi}$ team.

I owe my deepest gratitude to Jian Wu, who took the time to help educate me on CiteSeer$^{\chi}$ and then help guide and review my work throughout. Without him, my thesis would have never reached this completed stage and I thank him for answering all my questions and always being willing to lend a hand.

I am grateful to David Yang for his help in testing out portions of my work and giving productive feedback.

I would also like to thank Kyle Williams, Sagnik Ray Choudhury, and Douglas Jordan for their help and intelligent advice about CiteSeer$^{\chi}$.

Finally, I would like to thank my honors advisor, Jesse Barlow, for guiding me throughout my four years at Penn State with regards to whatever I needed. It has been a wonderful four years!

Chapter 1

Introduction

1.1 Background

In the modern information age, it's a challenge to store, organize, and make accessible the vasts amounts of data available. Digital libraries attempt to do this, but face a variety of challenges such as dealing with huge quantities of data at reasonable speeds, dealing with unreliable data, and making data easily searchable and accessible. A key to accomplishing these tasks successfully is being able to extract content and metadata accurately from the unstructured data contained in analog or digital sources.

CiteSeer$^{\chi}$[1], a digital library of scholarly papers, deals primarily with research works published as PDF files. Papers can be searched for by their content, title, authors, keywords, abstract content, year of publication, and more. Papers are linked to each other by citation information and clustered with similar papers.

[1] http://citeseerx.ist.psu.edu

1.1.1 CiteSeer$^{\chi}$ Overview

CiteSeer$^{\chi}$'s components can be divided into two main groups, the frontend, which responds to user web requests, and the backend, which handles data acquisition and preparation. This is diagrammed in Figure 1.1 [1].

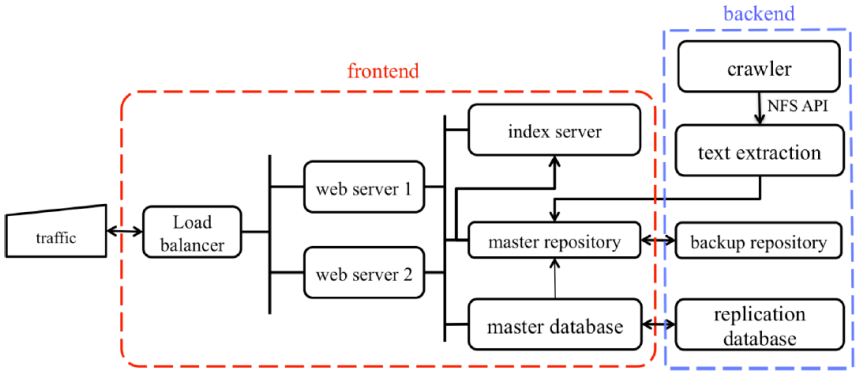

Figure 1.1: CiteSeer$^{\chi}$ architecture overview

The frontend web servers retrieve information from the master database, the master repository, and the index server and serve the content back to users as appropriate. The index server is used for textual searches of the full body text extracted from PDF files. The master repository contains the actual PDF files; the master database holds the metadata extracted from these files, such as title, author, and citation information.

The backend is where the extraction system is located. First, the crawl system crawls the web to find potential academic PDF files. The PDF is then stored on disk along with a small amount of metadata in the crawl database, such as the date the crawler found the PDF and the URL where the PDF was found.

After PDF files are acquired via the crawl system, the extraction system extracts content and metadata from them. The results of the extraction process for each document are stored in various text and XML files.

Finally, the ingestion process is run which adds the extraction results to the master database and index servers as appropriate. This process is run on the backup repository

server. More than simply copying information from destination to source, the ingestion process has to dedupe and cluster data, link and match citations, and disambiguate authors [1].

1.1.2 Content and Metadata Extraction

The extraction system's role, to extract content and metadata, such as paper titles, authors, citations, and body text from scholarly PDF files is a critical step in making CiteSeer$^{\chi}$'s content accessible and searchable. This data could be entered manually for each document, but this would be impractical to do as CiteSeer$^{\chi}$ stores approximately five million documents. In light of this, a fast and high-quality extraction process for PDF metadata is a crucial component of CiteSeer$^{\chi}$.

Unfortunately, the PDF file format is a complex file format designed to standardize document presentation, not convey content in a machine-friendly way. As an additional issue, there is no standard way to specify metadata for PDF files which makes automatic metadata extraction difficult [2]. So, any metadata extraction system for PDF files has to handle the challenge of accurately extracting content and metadata from a file format composed of unstructured data. In order to accomplish this, an extraction system ought to be designed to be *maintainable*, *usable*, *reliable*, *accurate*, and *performant*.

1.2 Objective

The objective of this thesis is to create a new extraction system for CiteSeer$^{\chi}$ which is *maintainable*, *usable*, *reliable*, *accurate*, and *performant*.

Maintainable software is software that can easily "be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment" [3]. Although there have been many attempts to quantify maintainability, a a recent systematic review has shown that these maintainability analysis methods are likely

not effective [4]. In the case of CiteSeer$^{\chi}$, since new extraction concepts and tools are constantly being developed, it is specifically important that the extraction system is extendable, a property of maintainable software.

Usable software is software where a user can easily "learn to operate, prepare inputs for, and interpret outputs of the system" [3]. In the case of CiteSeer$^{\chi}$, the extraction system should be as automatic as possible and require as little human intervention as possible.

Reliable software is able to "maintain a specified level of performance when used under specified conditions" [3]. In the case of CiteSeer$^{\chi}$, the extraction system should be able to handle any PDF file as input and extract results (either successfully or not) without crashing.

In this paper, an *accurate* extraction system refers to the quality and accuracy of extracted content and metadata. In the case of CiteSeer$^{\chi}$, this is the content and metadata extracted from scholarly PDF documents. Without accurate extraction results, CiteSeer$^{\chi}$'s content would be much more difficult to access for end users.

In this paper, *performant* software is software that performs at an acceptable level. In the case of CiteSeer$^{\chi}$, the extraction system should extract data and metadata from PDF files quickly. The vast quantity of online scholarly documents, at least 114 million [5], makes this a necessity.

1.3 Outline

This chapter presented some background information about digital libraries, CiteSeer$^{\chi}$, and the challenges in extracting content and metadata from PDF files. The second chapter will give background information on the architecture of CiteSeer$^{\chi}$'s extraction system and discuss the positives and negatives of the current system. Chapter 3 will provide a summary of an attempt to modify and improve this extraction system that ultimately did not meet all of the goals listed above. The fourth chapter will explain the design and development of an extraction framework software library to support the goals for an improved extraction system.

It will then describe how CiteSeer$^{\chi}$ can use this framework to create a new and improved extraction system which meets its objectives. The final chapter will provide conclusions on the work as a whole and possibilities for further research.

Chapter 2

The Current CiteSeer$^{\chi}$ Extraction System

2.1 Design

The extraction system of CiteSeer$^{\chi}$ extracts content and metadata from scholarly documents in the PDF file format. The process is controlled by a Perl script and the general flow is diagrammed by the flowchart in Figure 2.1.

The extraction system extracts information from documents that the crawl system has found. The crawl system maintains a database of these documents; the primary table in this database with information relevant to the extraction system is the `main_crawl_document` table. Table 2.1 illustrates the schema of the `main_crawl_document` table. Of especial importance is the `state` field, which holds an integer value representing the extraction status of the document. Possible values and meanings are listed in Table 2.2.

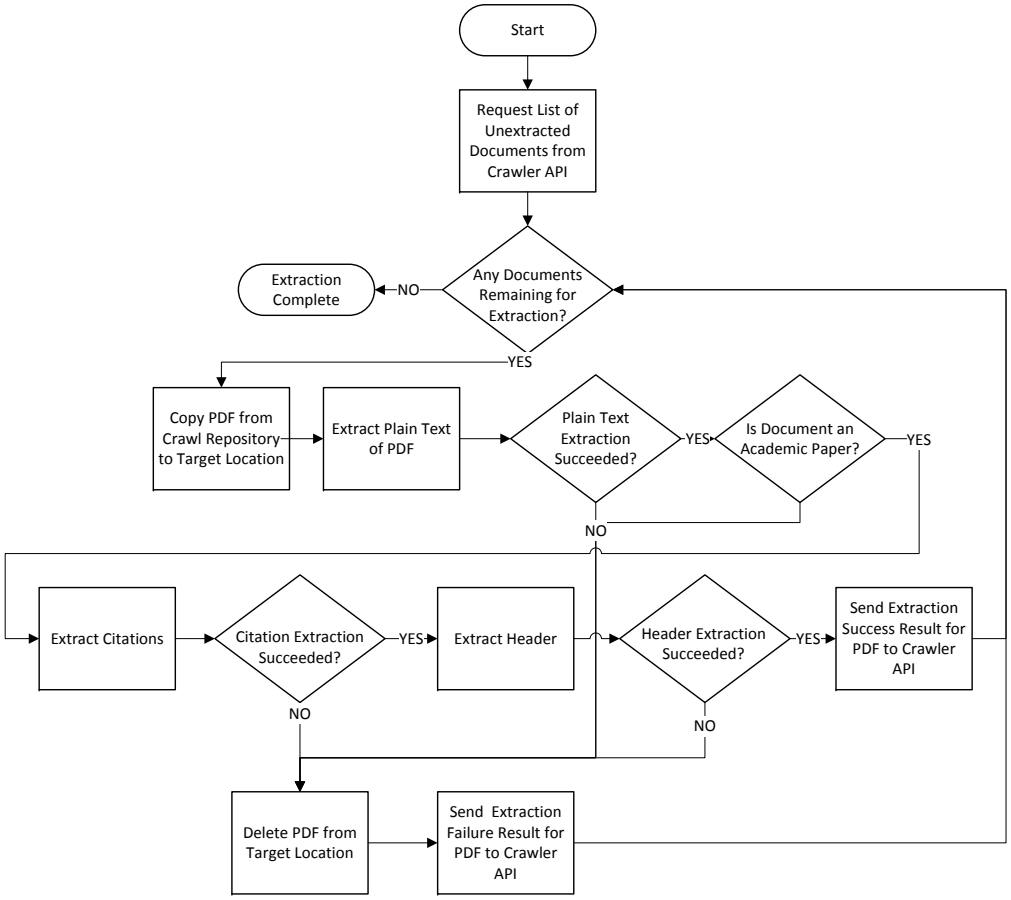

Figure 2.1: CiteSeer$^{\chi}$ extraction process

The Perl script makes an HTTP request to the crawl API server, a server that provides a layer of abstraction over the crawl database, for a list of documents on which to perform extraction. The number of documents requested can be set as a parameter of the HTTP request. In response to the request, the crawl API server retrieves a batch of documents from the crawl database that have not had extraction attempted on them yet, that is, documents with a state of 0. The API server then updates their state to 2 and sends this list of documents back to the Perl script in an XML format.

The Perl script then iteratively runs through the extraction process on each document. First, it copies the PDF document from the crawl repository to the target location, accessible by the ingestor. It next tries to convert the PDF to a plain text file, using either Apache

Field	Type	Description
id	int(11)	Document ID
url	varchar(255)	Document URL
md5	varchar(32)	MD5 hash of url
host	varchar(255)	Host name of url
content_sha1	varchar(40)	SHA1 hash of PDF
discover_date	datetime	Time document was first crawled
update_date	datetime	Time document was last crawled
parent_id	int(11)	Parent URL ID
state	int(11)	Status of ingestion

Table 2.1: Schema of the `main_crawl_document` in the crawl database

Value	Meaning
0	Data not extracted
1	Extraction completed successfully
-1	Extraction failed
2	Extraction ongoing

Table 2.2: Meanings of the `state` field in the `main_crawl_document` table

PDFBox [1] or PDFlib TET (Text Extraction Toolkit) [2]. If this succeeds, the script tries to determine if the paper is an academic paper based off of the contents of the extracted plain text. Currently, this is done through a simple regular expression that looks for the presence of the term 'References' or 'Bibliography.' If the paper is academic, then citations are extracted from the plain text file using the ParsCit library [6]. If this succeeds, header information, such as the title and authors, is extracted using SVMHeaderParse [7]. If the header extraction is successful, an HTTP request is made to the crawl API server indicating that content and metadata extraction was successful for the current document. If at any step along the way a problem occurs, such as PDFBox failing or the paper failing the academic filter, the extraction process ends for the document and an HTTP request is made to the crawl API server indicating that extraction failed. The crawl API server updates the status

[1] http://pdfbox.apache.org/
[2] http://www.pdflib.com/products/tet/

of the document in the `main_crawl_document` table depending on the parameters in the request it receives.

2.2 Discussion

While this design is functional, it is not ideal and there are a variety of areas in which it could be improved.

First, the implementation of the current design is not maintainable. The Perl script that controls the extraction process is put together in an ad-hoc fashion in which the code is quite brittle. While difficult to quantify, examples of flaws in the code include hard-coded integers with qualitative meaning, a sequential interspersing of subroutine definitions and imperative code, unused constant-like variables, and lines of code commented out with no clarifying explanations. In addition to these issues, the script's overall structure makes it difficult to extend or to add features. CiteSeer$^{\chi}$ has other extractors, such as table, figure, algorithm, and acknowledgment extractors in development which could be a part of the extraction process alongside of header and citation extraction. A better extraction system would be more modular and make it easy to add or remove extractors like these.

Second, the program terminates after extracting data from the list of documents it received from the crawl API. Instead, to make the program more usable, it should continue polling the crawl API after a run so it does not need to be manually restarted.

Third, the implementation of the current system is not reliable. For example, if PDFBox or TET hangs, the whole extraction process will hang. A more reliable design and implementation would handle this.

Fourth, the current extraction system could extract data and metadata more accurately. A recent evaluation of different metadata extraction tools compared their performance for extracting the title of a paper, its authors, its abstract, and its year. SVMHeaderParse, the tool that CiteSeer$^{\chi}$ currently uses for header metadata extraction, produces less accurate

results in every category than Grobid[3], a competing tool.

Finally, this design only processes documents sequentially. A more performant design would process documents in parallel in order to take advantage of multi-core CPUs found on most machines. With the current design, in order to achieve any sort of parallel speedup, the Perl program must be started multiple times and run side by side with itself. However, this is not ideal from a user standpoint and could cause issues as the Perl script isn't specifically designed for this kind of parallel execution. For example, in the past, running the Perl script multiple times at once caused data consistency issues because there was no 'Extraction ongoing' state in the crawl database. Multiple processes would attempt to extract data for the same document at the same time which lead to race conditions and conflicts. The addition of the 'Extraction ongoing' state fixed this issue and allowed multiple side-by-side executions of the Perl script.

[3]https://github.com/kermitt2/grobid

Chapter 3

Improving the CiteSeer$^{\chi}$ Extraction System: A First Attempt

The first attempt to improve the extraction system focused specifically on improving usability and performance by automatically parallelizing the execution of the Perl extraction script. This chapter will detail its design and the issues with the design.

3.1 Design

As described in Chapter 2, the current extraction system requires human action to start the Perl script every time extraction is to happen. The focus of this attempt to improve the extraction system was the development of a 'wrapper' program which would automatically run the Perl extraction script (and run it in parallel with itself) thus removing the need for manual interaction. Because Java is a widely-used language (and widely-used in CiteSeer$^{\chi}$), and it is generally easier to work with and read than Perl, it was decided that this wrapper

would be written in Java. Figure 3.1 diagrams the relationship between the Java wrapper and the Perl extraction script.

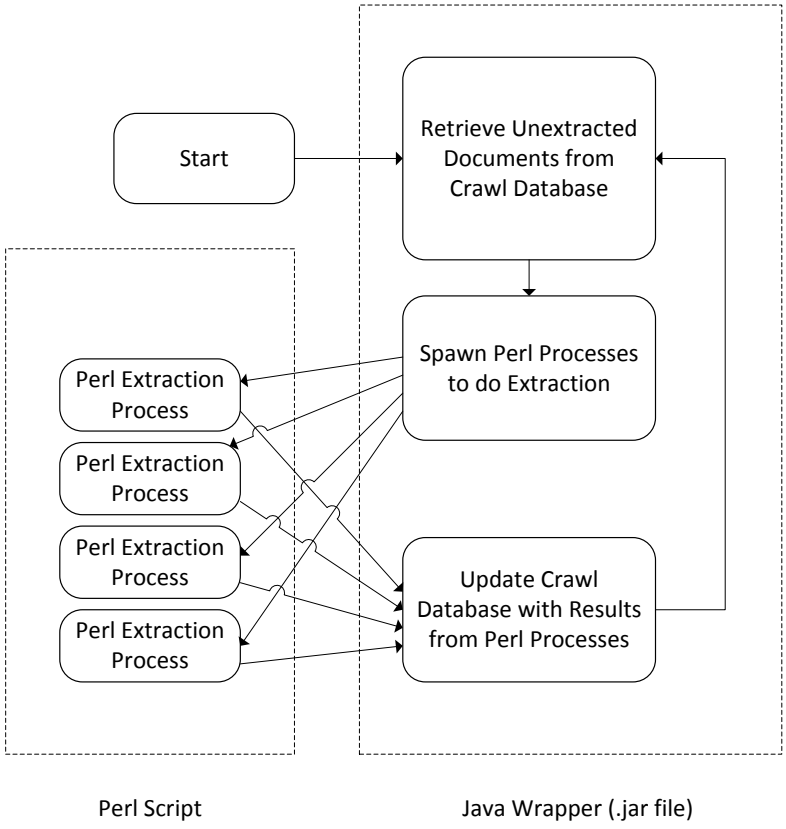

Figure 3.1: CiteSeer$^{\chi}$ revised extraction process

Specifically, the Java wrapper would get the IDs and locations of unextracted documents directly from the crawl database, thus bypassing the crawl API. It would then start the Perl extraction script multiple times as separate processes and pass a portion of the unextracted documents to each process. Each Perl process would iterate over its list of documents and extract information as in the previous design. Once a Perl script completed, it would send a success or failure status for each document back to the Java wrapper, which would then update the crawl database with the results. Any time a Perl script finishes, the Java wrapper would also retrieve more unextracted documents from the crawl database and start a new

Perl process to work on the extraction of data from these documents.

3.2 Discussion

This attempt at improving the extraction system had some positive results: it improved usability and it improved performance by making it easier to take advantage of multiple CPU cores. Multiple instances of the Perl script were now automatically run at once, removing the need for human action to start them. Also, data consistency issues were eliminated since the wrapper handled all reads and writes to the crawl database.

However, overall, this design was not a significant enough improvement and, in fact, created some new problems and worsened some prior problems. Getting documents from the crawl database directly instead of going through the crawl API was a mistake that decreased maintainability. The crawl API provided a nice level of abstraction over top of making raw SQL queries to the crawl database. Removing this layer of abstraction coupled the extraction system more tightly to the crawl system and it also added unnecessary complexity to the extraction system code.

As the extraction system was primarily in Perl, it may have been cleaner to implement the parallelization in Perl. Adding a layer of Java code added more complexity that decreased maintainability and increased the chance of bugs.

Finally, this design did not address some of the core problems with the extraction system: its reliability, maintainability, and accuracy. The extraction system still suffered from the same issues where it would hang while processing some PDFs, the system was still hard to extend, and the system was still using extraction tools, like SVMHeaderParse, that could be replaced with better alternatives. A larger reworking of the system would be needed.

Chapter 4

A New CiteSeer$^{\chi}$ Extraction System

To effectively accomplish the goals of making the CiteSeer$^{\chi}$ extraction system maintainable, usable, reliable, accurate, and performant, a broader approach than small changes to the existing codebase was needed.

4.1 Extraction Framework

To facilitate this large redesign and rewrite of the extraction system codebase, a small general-purpose extraction framework was developed.

4.1.1 Objectives

There were multiple goals behind the creation of the extraction framework. The most important, perhaps, is that the framework should facilitate maintainable code in the CiteSeer$^{\chi}$extraction system. The framework's main purpose is to abstract away the logic of coordinating and running the extraction process and let the user focus solely on defining the actual logic for

extracting data. Also to this end, the framework should meet the needs that CiteSeer$^{\chi}$'s extraction system requires, but it should not be coupled directly to CiteSeer$^{\chi}$. In other words, it should be possible to use for applications beyond just CiteSeer$^{\chi}$'s use case.

The framework should be performant by easily supporting data extraction from documents in parallel.

Like any well-designed library, the extraction framework should have a usable and easy to understand API. As poor documentation is one of the most common API usability issues [8], the framework should be clearly documented with up-to-date documentation.

The framework should be reliable. This in part entails having a solid set of tests that cover the framework internally.

4.1.2 Design

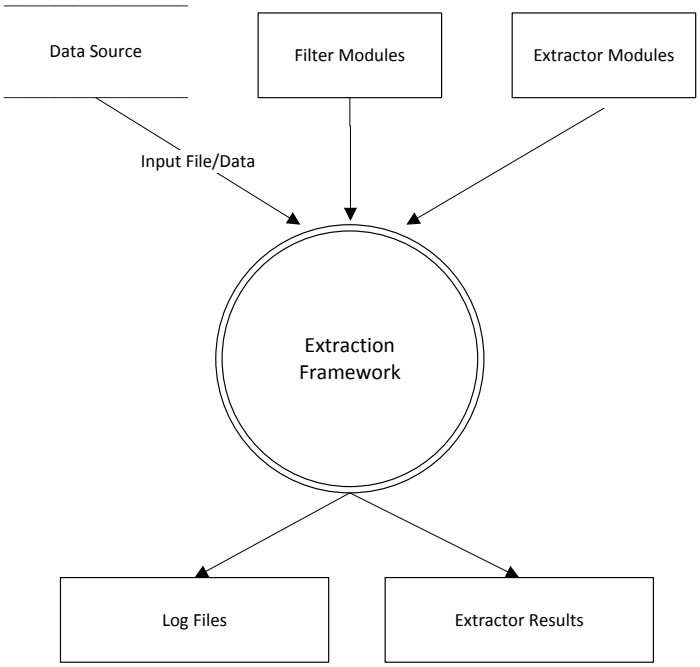

Figure 4.1: Black box diagram of the extraction framework

The overall concept of the framework is fairly simple. A user of the framework creates their own extraction and filtering classes (collectively referred to as runnables). These

runnables have three main properties: the needed input data (the runnable's dependencies), the actual extraction or filtering logic, and the type of the output. For example, in CiteSeer$^{\chi}$'s case, the user would define an extraction class that takes a PDF file as an input and outputs a plain text file representing the textual contents of the PDF. The user might also define a filter class that takes the textual contents of the PDF and determines whether it is academic or not. These runnables are provided to the framework, as well as a starting input source(s). The extraction framework then runs the source through the filters and extractors, handling any exceptions along the way and outputting the results. This is illustrated in Figure 4.1.

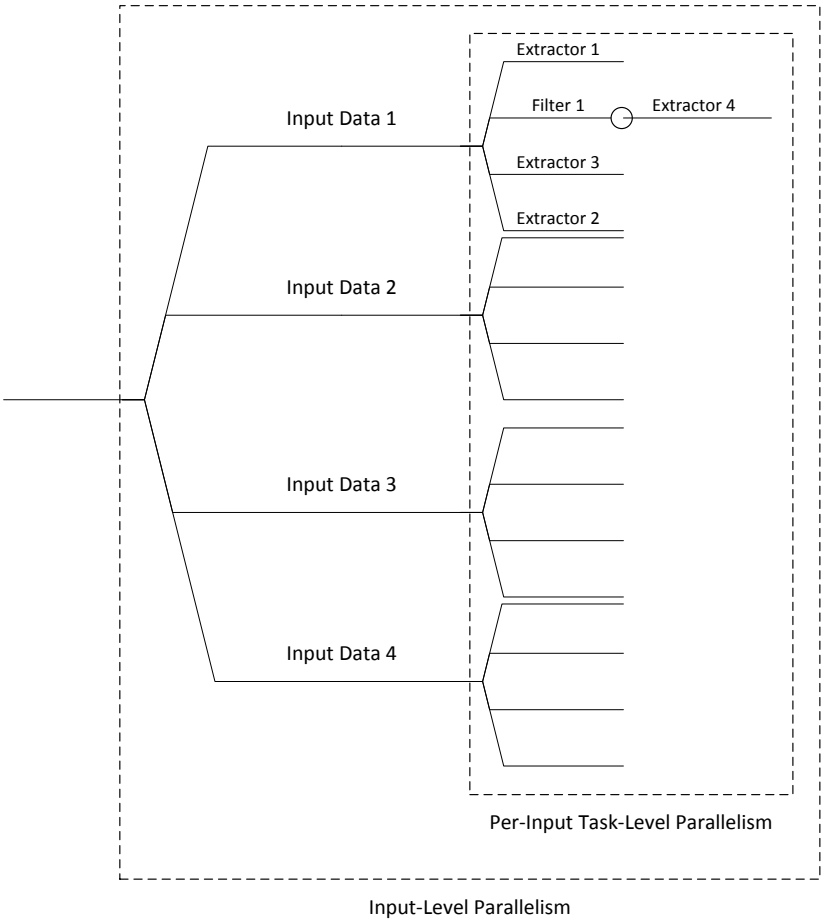

Figure 4.2: Levels of extraction framework parallelization

Internally, the extraction framework creates a directed acyclic graph (DAG) of the pro-

vided runnables, based on each runnable's specified dependencies. When provided with input data, the framework then executes the runnables on the data in the appropriate order based on the DAG, executing runnables that do not depend on each other in parallel.

In addition, the framework can process a batch of initial input data in parallel. This task also is quite conducive to parallelization as there is no shared data or messages that need to be passed between threads: each input can have information extracted from it in isolation. These two levels of parallelization, input-level parallelization and per-input task-level parallelization are diagrammed in Figure 4.2. Figure 4.2 is just meant to represent the parallelization possibilities: there could be a different number of threads handling input data, and the layout of extractors and filters has no specific meaning other than to demonstrate parallelization potential.

4.1.3 Implementation

The framework was implemented in Python. The code is split up into four main modules. The `core` module is responsible for the main functionality of the framework; it runs the overall extraction process. It contains the `ExtractionRunner` class, which is how users of the software library configure and execute extractions. The `core` module internally manages all parallelization referenced in Figure 4.2[1].

The `runnables` module contains the code which lets users define their own runnables. It contains the base `Runnable` class and the `Extractor` and `Filter` class which inherit from `Runnable`. To define extractors and filters, a user extends either the `Extractor` or `Filter` class and overrides its `extract` or `filter` method, respectively.

The `utils` module, contains various utilities and useful tools. For example, `utils` contains a function called `external_process`, which provides users with an easy way to start, pass data to, and get the result from an external process while specifying a time-limit for the process.

[1] At the time of this writing, the framework is still being actively developed. The DAG construction and task-level parallelization is not yet implemented, while the input-level parallelization is implemented.

Finally, the `log` module contains some additional logging handlers that the extraction framework uses by default.

The full source code for the extraction framework is available online on Github.[2] At the time of writing, the framework is still being actively developed.

4.1.4 Discussion

Implementing the framework in Python has positives and negatives. Python has been shown to be more usable than Perl [9], and is widely used in academics. This is a positive with regards to a project like CiteSeer$^{\chi}$, where many researchers may be working with the code over time who will probably be more familiar with Python than Perl. On the other hand, there are some negatives, for example, Python lacks static typing. Static typing can help programmers to write more clear and correct code [10].

Another difficulty with Python is the Global Interpreter Lock (GIL), present in Python's standard implementation, cPython. The GIL prevents the Python interpreter from executing more than one thread at a time. (Threads may release the GIL when waiting on I/O, when doing computations in a C extension, etc.) So, the GIL essentially prevents CPU-bound pure Python code from being able to gain performance benefits through multi-threading. However, multi-processing does not suffer from the GIL limitations that multi-threading does because multiple instances of the Python interpreter are executing. As Python has a `multiprocessing` module in its standard library parallel, speedup can still be easily achieved in Python. The GIL issue arose in the development of this framework, and multi-processing had to be used instead of multi-threading in order to meet the performance goals for the framework.

The framework's public API meets the usability goal for the framework. It is easy to use with smart defaults, but still provides flexibility through optional parameters and methods.

The framework has a set of tests that help meet the reliability goal. The framework also

[2] https://github.com/SeerLabs/extractor-framework

facilitates reliability in end-user code by handling exceptions in user extractors and filters cleanly. It also provides useful utility functions, such as the `external_process` function mentioned above.

Finally, the framework's API, which allows users to define their extractors and filters in a modular, decoupled way, meets the goal of facilitating maintainable projects. The framework though be somewhat specialized to CiteSeer$^{\chi}$'s specific needs: the decision to include the `Filter` and `Extractor` classes as the only two types of runnables is fine for CiteSeer$^{\chi}$but might not generalize as well to other scenarios. This is fine for the CiteSeer$^{\chi}$ project, but it might make the framework less useful in other scenarios.

4.2 CiteSeer$^{\chi}$ Extraction System

The final step in the CiteSeer$^{\chi}$ extraction system redesign was to rewrite the extraction system code using the extraction framework described in the previous section.

4.2.1 Design

Overall Design

With the new extraction framework developed, the design of CiteSeer$^{\chi}$'s extraction system is greatly simplified. Essentially all the implementation work is accomplished by writing individual extractors and filters.

The overall design has a slightly different model than before: instead of extracting a plain text version of the PDF and then using ParsCit and SVMHeaderParse to extract metadata, the new design uses Grobid to do the heavy-lifting of extraction. Grobid takes a PDF file and extracts its contents into a TEI-formatted XML file[3]. This is based on recent research that shows that Grobid performs best at extracting metadata from scholarly articles [2]. A

[3]The Text Encoding Initiative (TEI) Consortium provides guidelines for representing textual content in a structured, digital form. More information is available at their website: http://www.tei-c.org

filter, using the information from the TEI-formatted XML file, determines if the document is academic. If it is, header, citation, and other information is extracted using the TEI XML file or the original PDF and outputted in a format friendly to the rest of CiteSeer$^{\chi}$.

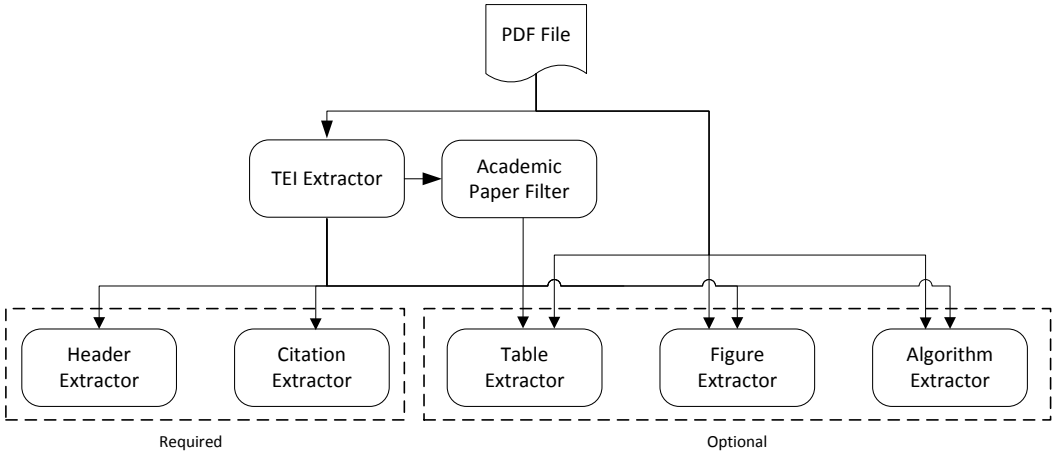

Figure 4.3: High-level design of the new CiteSeer$^{\chi}$ extraction system

Figure 4.3 diagrams the relationship between the different extractors and filters. An arrow indicates a dependency, that is, an extractor or filter at the tail of an arrow must finish before the extractor or filter at the head of the arrow can start.

Design of Filters and Extractors

TEI Extractor The TEI extractor takes the PDF file as input and makes a local HTTP request to a Grobid service running internally in the CiteSeer$^{\chi}$ network. The extractor handles any errors appropriately, such as the Grobid service not responding. Finally, the extractor returns the TEI-formatted XML file.

Academic Paper Filter The academic paper filter determines if a PDF found when crawling the web is actually a scholarly document. It uses a machine learning approach identified to be more accurate than the old naive filtering method [11].

Header Extractor The header extractor takes in the TEI file as input. It traverses the file and finds header information relevant to CiteSeer$^{\chi}$, such as the paper title, author names, author affiliations, and the abstract. It then outputs this information in an XML file which fits CiteSeer$^{\chi}$'s expected schema.

Citation Extractor The citation extractor takes in the TEI file as input. It traverses the file and finds citation information relevant to CiteSeer$^{\chi}$. For example, for each citation found, data such as the paper title, paper year, journal, and author names might be extracted. Also, the location in the paper of the citation reference and surrounding context is extracted. Finally, the extractor outputs all this information in an XML file which fits CiteSeer$^{\chi}$'s expected schema.

Integration with CiteSeer$^{\chi}$ as a Whole

Integration with CiteSeer$^{\chi}$ is similar to before. A batch of unextracted PDF files is retrieved via the crawl API. Content and metadata is then extracted from these documents in parallel using the new extraction framework as explained above. Parameters like extraction file output location can be set on a file by file basis for each file in the batch. In the case of CiteSeer$^{\chi}$, the output files are written to the location expected by the ingestor. Finally, the extraction status for each document (either success or failure) is updated via the crawl API when extraction finishes. The process repeats while there are still documents on which extraction has not been attempted.

The full source code for the new extraction system is available online on Github.[4] At the time of writing, the system is still being actively developed.

[4]https://github.com/SeerLabs/new-csx-extractor

4.2.2 Discussion

The use of the new extraction framework described in Chapter 4 greatly improves the whole extraction system for CiteSeer$^{\chi}$. The extraction system is much more maintainable as a whole. If new extractors are needed, they can be easily added into the system. For example, it would be trivial to add the optional table extractor to the system (assuming the table extraction logic has been developed). It's also simple to change the extraction system: for example, if a new program was found that more accurately extracted paper metadata than Grobid, it would be easy to replace the Grobid extractor class with a different one which used the new program.

The system is also more reliable because of the extraction framework, it handles errors more rigorously and should not hang.

Number of Processes	**Time (seconds)**
1	210.03
2	109.90
3	79.55
4	67.75

Table 4.1: Time elapsed for data extraction on 100 PDF files

While it's hard to compare performance directly between the old and new extraction systems because of the switch from PDFBox/TET to Grobid, the new system provides reliable parallelization by default which is important for performance. Table 4.1 illustrates the parallel speedup using one to four processes on a sample of 100 PDF files on a four-core 2.53GHz CiteSeer$^{\chi}$machine. The system achieves a very reasonable 77% parallel efficiency in the four-process case.

Finally, this internal support for parallelization removes hassle for the administrator of the extraction system and enhances usability. The new extraction system also does not need to be restarted by hand as before.

Chapter 5

Conclusion

The new extraction system is a major improvement compared to the previous extraction system. It is more maintainable, usable, reliable, accurate, and performant than the current extraction system.

However, there are still multiple areas where more research could be done and the extraction system could be improved. One possible expansion would be to design a distributed extraction system that worked across multiple machines instead of just one machine.

While this paper didn't specifically focus on methods to improve the quality of the data extracted, improving the quality of extracted content and metadata or extracting new types of metadata would also lead to a better extraction system.

Overall, the new extraction system should help CiteSeer$^{\chi}$ address data extraction challenges that it faces and result in a better digital library for everyone.

Bibliography

[1] K. Williams, Jian Wu, S.R. Choudhury, M. Khabsa, and C.L. Giles. Scholarly big data information extraction and integration in the CiteSeer$^{\chi}$ digital library. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 68–73, March 2014.

[2] Mario Lipinski, Kevin Yao, Corinna Breitinger, Joeran Beel, and Bela Gipp. Evaluation of header metadata extraction approaches and tools for scientific pdf documents. In *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '13, pages 385–386, New York, NY, USA, 2013. ACM.

[3] Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[4] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 367–377, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Madian Khabsa and C. Lee Giles. The number of scholarly documents on the public web. *PLoS ONE*, 9(5):e93949, 05 2014.

[6] Isaac G Councill, C Lee Giles, and Min-Yen Kan. Parscit: an open-source crf reference string parsing package. In *LREC*, 2008.

[7] Hui Han, C Lee Giles, Eren Manavoglu, Hongyuan Zha, Zhenyue Zhang, and Edward A Fox. Automatic document metadata extraction using support vector machines. In *Digital Libraries, 2003. Proceedings. 2003 Joint Conference on*, pages 37–48. IEEE, 2003.

[8] M.F. Zibran, F.Z. Eishita, and C.K. Roy. Useful, but usable? factors affecting the usability of apis. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 151–155, Oct 2011.

[9] Lingyun Wang and Phil Pfeiffer. A qualitative analysis of the usability of perl, python, and tcl. *East Tennessee State University-http://www. python10. com/p10-papers/14/index. htm-geprüft*, 7, 2002.

[10] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer, 2004.

[11] Cornelia Caragea, Jian Wu, Kyle Williams, Sujatha Das, Madian Khabsa, Pradeep Teregowda, and C Lee Giles. Automatic identification of research articles from crawled documents. *Web-Scale Classification: Classifying Big Data from the Web, co-located with WSDM*, 2014.

Academic Vita

Jason P. Killian

EDUCATION

The Pennsylvania State University
Schreyer Honors College Scholar
B.S. in Computer Science (*May 2015*)

Hempfield High School
High School Diploma (*June 2011*)

SKILLS

Languages
- Ruby, Javascript, Python, Java, HTML/CSS, C, ActionScript, SQL, PHP, Objective-C, C++, C#, Perl

Frameworks/Libraries
- Ruby on Rails, jQuery, AngularJS, Bootstrap, Box2D, OpenMP, MPI, Yii

Developer Tools
- Vim, Git, Linux tools, Sublime Text, Eclipse, Visual Studio, Ant, Adobe Flash, Photoshop, Illustrator

Cloud Platforms
- Bluemix, Heroku, AWS, Softlayer, Cloudant

EMPLOYMENT

Software Developer Intern with IBM *(Summer 2014 to Present)*
- Developed applications and services on Bluemix, IBM's new cloud computing platform

Research Assistant at Pennsylvania State University *(Spring 2014 to Present)*
- Improved information extraction system module of CiteSeerX, a leading research search engine

Self-Employed Software Developer *(Fall 2011 to Present*)
A selection of a few recent jobs includes:

World in Conversation Project (Summer 2012 to Present)
Website to Coordinate and Hold Online Conversations
- Developed Ruby on Rails backend and collaborated with designer on HTML/CSS/JavaScript frontend
- Integrated OpenTok video chat platform

HighUp Studio (Fall 2012 to Fall 2013)
Browser-Based Asynchronous Multiplayer Game
- Developed client application in ActionScript 3 and server-side backend in C#

Envato (Spring 2012)
Programming Tutorials with Interactive Demos for Envato
- Published tutorials on number systems, bitwise manipulation, and ActionScript 3 techniques

HONORS/AWARDS

Code PSU Programming Contest: First Place (*Spring 2012, Spring 2015)*, **Second Place** *(Fall 2013)*
- Use of Java and Python to solve standard programming contest style problems

Sixty-seventh Place (over 5000 Entrants) in Spotify CodeQuest (*Spring 2012)*
- Use of Java to solve challenging programming contest style problems