Web Crawler Middleware for Search Engine Digital Libraries: A Case Study for CiteSeerX

Jian Wu[†], Pradeep Teregowda[‡], Madian Khabsa[‡], Stephen Carman[†], Douglas Jordan[‡], Jose San Pedro Wandelmer[†], Xin Lu[†], Prasenjit Mitra[†] and C. Lee Giles[†‡]
[†]Information Sciences and Technology [‡]Department of Computer Science and Engineering
University Park, PA, 16802, USA
jxw394@ist.psu.edu

ABSTRACT

Middleware is an important part of many search engine web crawling processes. We developed a middleware, the Crawl Document Importer (CDI), which selectively imports documents and the associated metadata to the digital library CiteSeerX crawl repository and database. This middleware is designed to be extensible as it provides a universal interface to the crawl database. It is designed to support input from multiple open source crawlers and archival formats, e.g., ARC, WARC. It can also import files downloaded via FTP. To use this middleware for another crawler, the user only needs to write a new log parser which returns a resource object with the standard metadata attributes and tells the middleware how to access downloaded files. When importing documents, users can specify document mime types and obtain text extracted from PDF/postscript documents. The middleware can adaptively identify academic research papers based on document context features. We developed a web user interface where the user can submit importing jobs. The middleware package can also work on supplemental jobs related to the crawl database and respository. Though designed for the CiteSeerX search engine, we feel this design would be appropriate for many search engine web crawling systems.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous ; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

search engine, information retrieval, web crawling, ingestion, middleware

1. INTRODUCTION

Crawling is a prerequisite and an essential process for operating a search engine. A focused crawler should be able to efficiently harvest designated documents from the internet. The CiteSeerX [?] digital library and search engine is designed to provide open access to academic documents in PDF and postscript formats. While a well designed vertical crawler can efficiently select documents based on their content types, it is also desirable to crawl all *potentially* useful files first and then selectively import documents of certain formats to the search engine repository.

Most available open source crawlers at present are designed for general purposes and are not customized for a particular search engine. Some web crawlers, such as *Heritrix*[?], have been well maintained and widely used by digital libraries, archives, and companies[1]. To take advantage of these crawlers to serve for a digital library which mainly indexes academic documents, it is necessary to define a clear interface to integrate these crawlers to the ingestion system of the search engine. Besides, this interface should also be able to import documents which are directly downloaded by FTP service.

Here, we develop a middleware, named Crawl Document Importer (CDI), to import documents of selected formats from files harvested by an open source crawler or from FTP downloads to the crawl database and repository before ingesting them into the CiteSeerX search engine. *Heritrix* is one of the highly rated and widely used open source crawlers, so we take it as an example application. However, the middleware is designed to be extensible, i.e., for another web crawler, the user only need to write a log parser/extractor which returns the standard metadata tuple and tells the middleware how to access the downloaded files.

The Python crawler written by Shuyi Zheng (hereafter the SYZ crawler) has been the dedicated harvester for the CiteSeerX project since 2009. This crawler is able to crawl about 5000–10000 seed URLs daily with a depth of two using a breadth-first policy. As a focused crawler, a number of filter rules are applied to selectively download free access online documents in PDF and postscript formats. As the CiteSeerX is expanding its service to other types of documents, switching to other more reliable and well maintained crawlers can be more efficient and desirable. The SYZ crawler is not able to import documents directly downloaded from FTP service. In addition, it is a necessity to define an interface to the crawler system in order to combine the crawler to the CiteSeerX code to make it more integrated. These considerations drive us to design a middleware which can work with multiple open source crawlers.

[1]`https://webarchive.jira.com/wiki/display/Heritrix/Users+of+Heritrix`

In the text below, the *files* include any types of resources downloaded from the web, while the term *documents* refers to files in user specified formats, e.g., PDF/postscript. The *repository* is a directory which stores physical academic documents and *database* refers to the records managed by a database management system. We use MySQL to manage the database.

2. FUNCTIONALITIES

The current version of CDI has the following features and functionalities:

1. **Extensibility.** The middleware defines a standard interface to the crawl repository and database, so it minimizes the amount of coding to import documents downloaded by other web crawlers.

2. **Multiple input file formats.** The CDI middleware is capable of handling input files written by Heritrix mirror writer, the ARC and WARC writers. It also supports importing documents from FTP downloads.

3. **Multiple mime type support.** The desired mime types can be specified in the configuration file. For CiteSeerX purposes, we focus on PDF and postscript formats.

4. **Control options.** The users have options to turn on/off the document content filter (DCF), and the database writer. The DCF checks if the document is academic or not. If the database writer is turned off, the documents that pass all filters are saved to a separate folder.

5. **Standard MySQL table.** The middleware creates standard database tables automatically. All database operations are implemented using the `cursor` object, so the user do not need to edit the `models.py`.

6. **Web user interface.** A web console is developed on top of the Django project[2] [**?**] to provide a user friendly interface to the middleware.

7. **Command line mode.** The user can also run the importing jobs from the linux console using a single command.

8. **Multiple functionalities on database query.** Besides document importing, the CDI middleware can also transfer documents which meet a certain conditions out of the crawl repository and generate the seed list (whitelist) for future web crawling.

9. **On-screen information prints.** The middleware prints processing status for each file in color on the linux console. It automatically counts the numbers of files with different flags and prints them in a table at exit.

3. DESIGN AND CODING

This middleware is designed to provide a universal interface for multiple crawlers and input file mime types to minimize the extra coding when a new crawler is used. The role of this middleware in the crawler architecture is illustrated in Fig. 1. The CDI plays as a "bridge" between the crawler and the crawl repository/database. In this figure, Heritrix crawler can be replaced with another crawler or FTP downloads, which may have a different log formats. The SYZ crawler was designed specifically for the CiteSeerX project, so it does not require the CDI middleware.

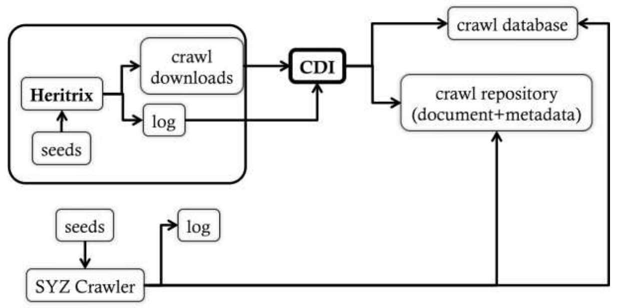


Figure 1: The role of the CDI middleware with the crawler, the repository and the database.

3.1 Work Flow

The work flow of the CDI middleware in form of pseudo code is illustrated in Algorithm 1. The middleware first checks the user configurations, including the accessibility of the input and output directory. If the crawl files were saved in ARC/WARC formats, they are decompressed and files/folders are saved in hierarchical order as a site mirror. The crawl log file is then loaded and each line is parsed according to its specific format. The goal is to extract the resource URL (u_i), its parent URL (p_i), the time it was crawled (t_i), the file format (content-type) as identified from the HTTP response header (if available), and the crawl depth (h_i). Some log records are skipped because they do not provide actual file downloads, e.g., the DNS checking request. After parsing all log records, the middleware creates database tables if they do not exist. Then it loops over and processes all the files it identifies in the log file.

First, all the resource attributes of File i are encapsulated into a single variable r_i. The mime type filter is then applied and only files that are among the mime types specified in the configuration file can pass this filter. The program then checks if these files physically exist in the local drive and decompresses gzipped files if necessary. If the DCF is turned on, the text information must be extracted from the original files. Once the document content is accepted, it is saved to the repository and/or the database.

The middleware has several counters which count the file numbers that pass and fail at each filter. After finishing all files, it prints the values of all counters (see Table 1). The middleware also prints the execution time at exit.

In Fig. 2, we present the architecture of the CDI middleware. Each part of the middleware is described below in detail.

3.2 Log Parser

The log file and the crawl downloads provide the input. The log parser (or log/metadata extractor) parses the crawl log and is the only component which need to be replaced for a new crawler. This component is responsible for parsing

[2]`https://www.djangoproject.com`

Algorithm 1 The pseudo-code of the CDI program.

```
Require: Crawl log and documents
Ensure: Crawl database and repository
 1: check configurations
 2: decompress ARC/WARC files (if necessary)
 3: parse the log file to get $u_i$,$p_i$,$t_i$,$f_i$,$h_i$
 4: create database tables
 5: for File $i$ from log do
 6:    $r_i$=resource($u_i, p_i, t_i, f_i, h_i$)
 7:    if not mimetypeFilter.check($f_i$) then
 8:       continue
 9:    end if
10:    if not $i$.fileExist then
11:       continue
12:    end if
13:    if $r_i$.mimetype == 'x-gzip' then
14:       unzip file $i$
15:       find the document path
16:    end if
17:    if document content filter == 'ON' then
18:       if not $i$.extractText then
19:          continue
20:       end if
21:       if not documentcontentFilter.check($i$.text) then
22:          continue
23:       end if
24:    end if
25:    if save to database == 'OFF' then
26:       save_to_repository($i$)
27:       continue
28:    else
29:       save_to_database($r_i$)
30:       save_to_repository($i$)
31:    end if
32: end for
33: save_verification()
34: counters.print()
35: print(processing time)
```

and extracting metadata from crawl logs. The output of this extractor should at least provide a resource object with the following attributes:

1. The **document URL**, i.e., the original URL where a document is downloaded. For example, `http://www.example.edu/tom/pub/paper1.pdf`. This attribute is required.

2. The **parent URL** of the document URL, i.e., the page where the paper is linked to. This is usually an HTML or a text page, e.g., `http://www.example.edu/tom/home.htm`. This attribute is also required.

3. A boolean variable indicating whether the document URL is a **seed or not**. This attribute is optional.

4. The **depth** of the document URL with respect to its seed URL. For Heritrix, the depth information can be obtained by counting the letters in the *discovery path* field[3]. This attribute is optional. The depths of seed URLs are defined to be zero.

[3]`http://crawler.archive.org/articles/user_manual/glossary.html`

Table 1: Counters printed when program exists.

Counter Name	Description
all	All files processed
saved_New	New documents saved[a]
saved_Duplicate	Duplicate URLs[a]
filtered	All documents filtered out
filtered_MTF[c]	Documents filtered out by MTF
filtered_DCF[d]	Documents filtered out by DCF
failed_TextExtract	Failed to extract text
failed_FileNotFound	Files not found
failed_PDFFilenotFound	Zipped files not containing PDFs[b]

[a] This counter is set to zero if the user choose not to write into the database.
[b] The gzipped files may exist, but no PDF/postscript documents are found after they are decompressed.
[c] MTF=mime type filter.
[d] DCF=document content filter.

5. The **crawl date and time**, which is a Python `datetime` object. This attribute is required.

6. The document **mime type**, e.g., application/pdf. This attribute is required.

Both of document and parent URLs are normalized so that the relative paths are collapsed and the URL reserved characters are quoted and converted to utf-8 encoding if needed. The SYZ crawler itself can extract these attributes from the HTTP response headers. Heritrix crawler (version 1.14.1 or 3.0.0 and above) log contains all the information above. The FTP downloads may not generate a log file, but it is easy to generate a text file which lists the file paths. The attributes above can then be customized as constants (e.g., the download date and time) or depending on file paths (e.g., parent URL). If a new crawler is used, its log extractor should be able to extract and aggregate these attributes into a resource object.

3.3 Mime Type Filter

The metadata object is passed to the mime type filter which filters out any files which are not among the mime types specified in the configuration file. Mime types are identified from the HTTP header by the crawler or by using the 'file -mime' command. File extensions are not used for mime type detection as they are not reliable. Documents that pass the type filter will be renamed based on their mime type before saving to the repository. Documents which pass the mime type filter are processed by the DCF if it is toggled on. Otherwise, they can be directly saved to the repository.

3.4 Document Content Filter (DCF)

The DCF attempts to classify a document based on text contents extracted from PDF or postscript files. We use *PDFBox*[4] (version 1.5.0 and above) for PDF file text extraction and *ps2ascii* (GPL Ghostscript 8.70) for postscript file text extraction. At this time, we have not integrate the optical character recognition (OCR) into this middleware, so scanned PDF documents cannot pass this filter. A small fraction of postscript files cannot be parsed most likely because they are converted from images.

The content filter module performs a binary classification. The acceptance decision is made not just based on the content, but also by presence of sections such as *References*. The

[4]`http://incubator.apache.org/pdfbox/`

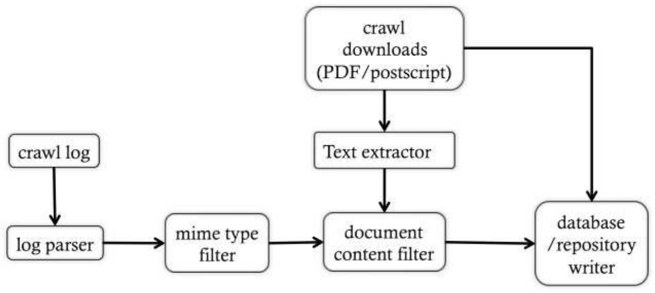


Figure 2: The architecture of the CDI middleware. Arrows indicate data flow directions. The log parser is the only component that needs to be replaced if a new crawler is used.

Table 2: Top features for content filtering.

Feature Name
Title case words
Page one section count
Figure keyword
Sentences with title case beginnings
Self descriptors (this paper)

filter utilizes a cross section of features including keywords, keyword positions, and typography extracted from the document text. Top features identified by the information gain metric are provided in Table 2. By iterating the process of feature selection, we were able to reduce the number of features from a few thousand keywords to 45. We trained a simple logistic classifier with over 400 documents and a cross validation accuracy of 91.3%. If a document is determined to be an academic paper, it is passed for further processing. Otherwise, it is dropped.

By default, the metadata is output in XML format, the extracted texts and the documents are saved to the same directory in the crawl repository.

3.5 Database Design

In Table 3 and 4, we present the database design. There are two entities in this design: document and parent URL. The encryption algorithms MD5 and SHA1 are used for duplicate detection. If a document passes all filters, its parent URL MD5 value is calculated. The document itself is downloaded to the cache (but not saved yet, see Section 3.6) and its SHA1 value is also calculated. If this document URL is new (its MD5 is new), a new database record is created. Otherwise, the SHA1 and the *update date* are both refreshed. If a document is saved into the crawl database, its parent URL must be saved or the *last crawl datetime* of the existing parent URL is updated. Similar to the document table, the parent URL MD5 value is also calculated to ensure that there is no URL duplication.

The *state* field indicates whether a document is ingested or not. This field is reset to the default value if an updated document version is identified. Ingestion is a post-crawl process to examine, parse, classify, and index the documents and make them searchable. There are three possible states for a given document: ingested, failed to be ingested, or not ingested (default).

Table 3: The `document` table design.

Field	Type	Description
`id`	int(11)	Document ID
`url`	varchar(255)	Document URL
`md5`	varchar(32)	MD5 value of document URL
`host`	varchar(255)	host name of document URL
`content_sha1`	varchar(40)	SHA1 value of crawled document
`discover_date`	datetime	First time to crawl this document
`update_date`	datetime	Last time to crawl this document
`parent_id`	int(11)	parent URL ID
`state`	int(11)	Ingested or not

Table 4: The `parent URL` table design.

Field	Type	Description
`id`	int(11)	parent URL ID
`url`	varchar(255)	parent URL
`md5`	varchar(32)	MD5 value of parent URL
`first_crawl_date`	datetime	Time of the first visit
`last_crawl_date`	datetime	Time of the last visit

The database design is implemented using MySQL (version 5.1 and above). The Django framework (version 1.3 and above) is applied for database interface. The database can be located on the localhost or a remote server.

3.6 Repository Design

The repository is a directory to save filtered crawled documents. Subfolders are automatically generated in a hierarchical order. Each subfolder is named using three digital numbers corresponding to a piece of document ID (see Table 3). For example, a PDF document with ID 1234567 is saved to 001/234/567/001.234.567.pdf. The associated metadata and text files (optional) are named as 001.234.567.pdf.met and 001.234.567.txt, respectively and saved to the same folder. If an updated version of a document is identified from an existing URL, the old document files in the repository is updated to ensure freshness.

3.7 Web Crawling from Heritrix

By default, Heritrix writes all its crawled contents to disk using its *ARCWriterProcesser*. The ARC file format has long been used by the Internet Archive[5]. Each ARC file consists of a *version block* and a *document block*. The version block identifies the metadata such as the original file name, version and URL records, while the document block records the body of the archive file. Heritrix can be configured to output files in WARC format. The WARC format, which has an ISO standard[6], is a revision of the ARC file format that has been traditionally used to store web crawls.

Although some ARC/WARC readers are developed, we found that the most reliable ARC/WARC reader is the *ARCreader* and *WARCreader* developed by the Internet Archive in Java[7]. Such readers are adopted by the *YouSeer* project[8]. The *YouSeer* project provides software to ingest documents harvested by Heritrix into the apache Solr[9]. Instead of indexing and exporting the crawled content to Solr, we save

[5]`http://archive.org/web/researcher/ArcFileFormat.php`

[6]`http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44717`

[7]`http://crawler.archive.org/apidocs/org/archive/io/arc/ARCReader.html`

[8]`http://youseer.sourceforge.net/`

[9]`http://lucene.apache.org/solr/`

extracted files to a temporary directory before they are accessed by the middleware document writer. The ARC/WARC files are compressed and can take a significant overhead of running time. It also require a large amount of temporary storage for extracted files.

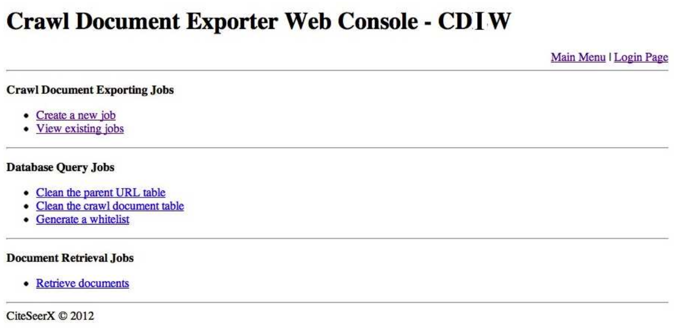
Heritrix can save crawled files in a *MirrorWriterProcessor*. The files are arranged in a directory hierarchy based on the URL, in which sense they mirror the file hierarchy that might exist on the servers. Because of this, in most cases, the location of a crawled file can be easily generated from the resource URL. Although this writer may take more space than the ARC/WARC writer, it saves a significant amount of time used to decompress ARC/WARC files.

3.8 Coding

This middleware follows an agile development model. The body is written in Python. Part of the package are adopted from the SYZ crawler. Java commands are invoked and executed within the Python code to handle ARC/WARC files and extract text content from PDF files.

4. WEB USER INTERFACE

We developed a web user interface for the CDI middleware, so that the user can create, modify and submit document importing jobs quickly. This interface needs to be deployed on the same server as the crawler and requires login credentials, but the web pages are accessible from remote computers. The web interface is based on the Django framework which provides a light weighted server. A snapshot of the dashboard (Main Menu) is presented in Fig. 3.


Crawl Document Exporter Web Console - CDIW

Main Menu | Login Page

Crawl Document Exporting Jobs

- Create a new job
- View existing jobs

Database Query Jobs

- Clean the parent URL table
- Clean the crawl document table
- Generate a whitelist

Document Retrieval Jobs

- Retrieve documents

CiteSeerX © 2012

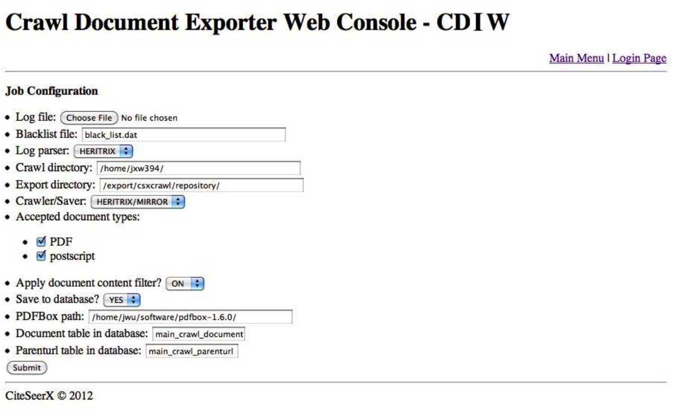
Figure 3: The dashboard (Main Menu) page of the CDI web console.

The dashboard displays three groups of jobs.

4.1 Crawl document importing jobs

These are the main jobs of the CDI middleware. Two options are provided. The user can either create a new job or view existing jobs. In Fig.4, we present the "Job Configuration" page after clicking "Create a new job" button. Most entries are in the configuration file. The "Blacklist file" contains a list of host names which the user does not want to import documents from. If the user choose `HERITRIX/ARC` or `HERITRIX/WARC` for the "Crawler/Saver", the middleware first calls the ARC/WARD reader module to extract the crawled documents before importing them.

The mime type filter is always applied. If the DCF is turned off, all documents in the selected document types are imported. If the "Save to database" is toggled off, documents


Crawl Document Exporter Web Console - CDIW

Main Menu | Login Page

Job Configuration

- Log file: Choose File No file chosen
- Blacklist file: black_list.dat
- Log parser: HERITRIX
- Crawl directory: /home/jxw394/
- Export directory: /export/csxcrawl/repository/
- Crawler/Saver: HERITRIX/MIRROR
- Accepted document types:
  - ✓ PDF
  - ✓ postscript
- Apply document content filter? ON
- Save to database? YES
- PDFBox path: /home/jwu/software/pdfbox-1.6.0/
- Document table in database: main_crawl_document
- Parenturl table in database: main_crawl_parenturl

Submit

CiteSeerX © 2012

Figure 4: The job configuration page of the CDI web console.

metadata are not saved to database and only imported to a separate directory. After pressing the submit button, the job configurations are saved to the database and the job starts to run. The users can also view existing jobs and re-run them.

4.2 Database query jobs

The "Clean the crawl document table" job cleans the document URL table by removing documents that meet *both* of these conditions: (1) the URL hosts are in the blacklist; (2) the documents fail to be ingested. The goal of this job is to remove crawl documents that are definitely not useful to save disk space and speed up database queries.

The "Clean the parent URL table" job generates a *parenturl.revised.csv* file which contains the cleaned parent URL list by removing low quality parent URLs that meet *any* of the following conditions: (1) the URL host names match any host names in the blacklist; (2) no document is found in the document URL table corresponding to the parent URLs; (3) the URL links are dead. The goal of this job is to obtain a list of high quality seed URLs for whitelist generation.

The "Generate a whitelist" job generates a text file that contains the URLs selected from the *parenturl.revised.csv* file and ranked based on the number of ingested documents and citation rates [**?**]. This whitelist represents the highest quality seeds selected and ranked based on the crawl history and can be re-crawled in the future.

4.3 Document retrieval jobs

The "Retrieve documents" job can retrieve a set of documents from the crawl repository or database which satisfies certain conditions. For example, the user can randomly retrieve 1000 PDF/postscript documents crawled between January, 2011 and June, 2011 as a sample for their research purpose.

5. RUNNING CDI

It is easy to run the CDI code on a Linux terminal. There is one configuration file for the CDI main program (`runconfig.py`) and one for the Django module (`settings.py`). The configuration items in `runconfig.py` are listed in Table 5. In the `settings.py`, the user only needs to configure the database access. All the statements follow the standard

Python syntax. To run the program, just make `cdi.py` executable and type

```
$ ./cdi.py
```

Table 5: Configuration terms.

Configuration Term	Description
`django_settings_module`	Django setting module
`logfile`	Full path of crawl log file
`logparser`	Crawler log parser
`inputdir`	Crawler download directory
`outputdir`	Crawl repository
`tempdir`	Temporary directory
`crawler`	Crawler name
`saver`	Crawler saver
`allow_doc_type`	Accepted mime types
`toggle_doc_content_filter`	Turn on/off DCF
`toggle_save_to_db`	Turn on/off database saver
`toggle_save_doc_separate`	Save documents to a separate folder
`pdfboxpath`	PDFBox .jar file path
`ps2asciipath`	ps2ascii command
`dbt_document`	document table name
`dbt_parenturl`	parent URL table name

If there is not any problem in the configuration file, the program outputs runtime information on the screen, including the progress (number of files processes vs. total number of files), the URL which is being processed, the full path of the file, whether the file passes the mime type filter, whether the text is successfully extracted, whether it passes the DCF, whether it is a new document (or duplicate), and finally whether this document is successfully saved.

Although Python 2.4 is supported, we recommend to run the most recent version of CDI using Python 2.6+ and Django 1.3+. To extract the text from PDF files, JDK 1.6+ and PDFBox 1.5+ need to be installed. The MySQL 5.1+ is required in the database server. The user needs to configure the iptables if the database server is not the localhost. The `SELECT`, `UPDATE`, `CREATE` and `DELETE` privileges should be granted to the MySQL user account.

The middleware has been tested on a Redhat Enterprise Linux (RHEL) 5.8 server with dual Intel Xeon E5440 @2.83GHz, and 32GB memory. The MySQL database server is on another server with AMD Opteron Dual-Core Processor 2216 @1.00GHz, and 8GB memory. The database engine is InnoDB and the main access table contains about 5 million rows with all columns indexed.

The middleware was able to process 912 crawl records from the SYZ crawler in about 22 minutes, all of which are academic papers in PDF formats with the DCF turned on. It was able to process 594,260 crawl records from Heritrix 1.14.4 mirror writer for about 16 hours with the DCF turned on. About 34,390 (5%) are academic papers. It was also able to process 1.6 million crawl records from Heritrix 3.1.0 mirror writer for about 8.35 hours.

The CDI middleware does not have a high requirement on the processor. The most two time consuming units are the text extractor and the DCF, but if the DCF toggle is turned off, the text extractor is not used either. However, the CDI middleware may have a high demand on the memory if the crawl log file is large. With our server, we are able to import a crawl log up to 10 million records ($\sim$ 3GB) with the crawling running in idle state.

6. TESTING

6.1 Scalability

To test the scalability of the CDI middleware, We performed two series of experiments in which we import open access medical and biological publications. In the first series of experiment, we turn off the DCF so that all documents are imported into the crawl database and repository. The time T it spends as a function of documents (N(DCF off))) imported are presented in Fig. 5, and the data are presented in Table 6.

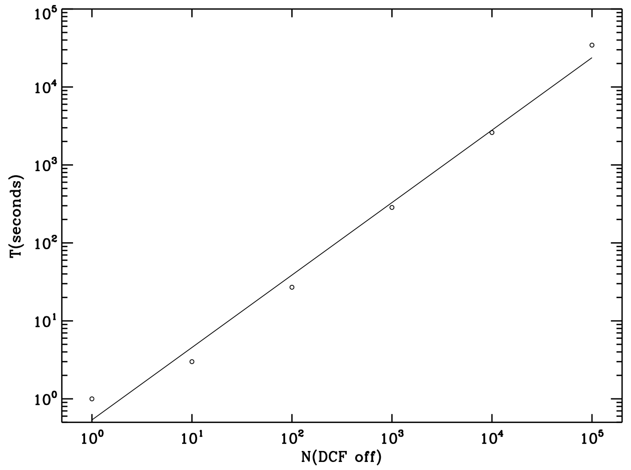


Figure 5: Processing time as a function of document number with the DCF turned off. See data in Table 6.

Table 6: Processing time for running each set of documents after turning off the DCF.

N(DCF off)	T/seconds
1	< 1
10	3
100	27
1000	285
10000	2605
100000	34413

A linear fitting in the logarithmic space yields a relation of

$$\log T = (0.93 \pm 0.05) \log N(\text{DCF off}) + (-0.27 \pm 0.15).$$

The slope is very close to one, which indicates a linear relation between the T and N(DCF off), and the average time to add a document without applying the DCF is about 0.24 seconds (calculated by averaging the T/N(DCF off) of the last five data points).

In the second series of experiments we turn on the DCF, and import a different set of documents (so there is no duplication). In Fig. 6, we plot the processing time as a function of document number. Turning on the DCF requires extracting text from PDF files which significantly increases the running time, so we only import up to 100 documents.

Fig. 6 demonstrate that the running time is linearly correlated with the number of documents. The ordinary lease-square fit yields a linear regression of

$$T = (2.37 \pm 0.08)N(\text{DCF on}) + (-2.86 \pm 4.98),$$

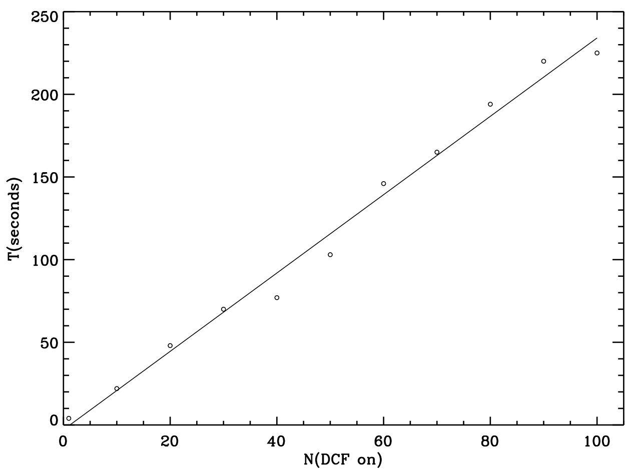


Figure 6: Processing time as a function of document number with the DCF turned on. Data are listed in Table 7.

Table 7: Processing time for running each set of documents after turning on the DCF.

N(DCF off)	T/seconds
1	4
10	22
20	48
30	70
40	77
50	103
60	146
70	165
80	194
90	220
100	225

which is consistent with a zero interception. It takes about 2.37 seconds to import a document by turning on the DCF.

This middleware was also able to import about 500 powerpoint files from a Heritrix crawl in only less than a minute without using the DCF. Actually, most of the time was spent on text extraction process.

6.2 Document Content Filter

The Document Content Filter (DCF) is designed to identify academic papers from all documents downloaded. To increase precision this filter applies a conservative algorithm on document selection. Because of this, a small fraction of academic documents may be misidentified as nonacademic. In addition, the keywords and features used in DCF are optimized for identifying computer science and information technology papers. As a result, this filter may have a systematic different identification rate depending on the scientific and academic domains. To test this, we run the CDI middleware on a collection of documents provided by the Cornell's arXiv project. The data in arXiv is separated into many disciplines of science ranging from various physics topics to computer science and statistics. We test our filter on three categories: computer science (CS), mathematical physics (Math-Ph) and a mixed collection of arXiv papers (arXiv). The arXiv category represents a general set of papers under no specific topic. It gives us a good spread of papers in different domains. In addition, we supplement our test data with our PubMed dataset, which represents a set of medical science papers. For each category, we randomly select 100 papers for ten times and count the number of papers that pass the DCF. Our result is presented in Table 8.

Table 8: Number of documents that pass the DCF. In each run, 100 document files are input.

Run	CS	Math-Ph	arXiv	PubMed
1	90	80	83	36
2	90	82	83	50
3	87	79	84	51
4	90	77	86	33
5	93	75	80	32
6	93	75	89	38
7	94	71	86	44
8	88	71	87	40
9	92	73	79	50
10	94	74	88	39
Average	91.1	75.7	84.5	41.3

The testing results indicate that our DCF can identify more than 90% of computer science papers. It can only identify about 75% of mathematical physics papers and 40% of medical science papers. The overall identification rate to the arXiv paper is about 85%. Table 8 indicates that using our DCF, the ingested document collection contains more documents in computer science and engineering. Revision of text features are needed to increase the relative proportion of identificate rate in other domains.

7. DISCUSSION

7.1 Generalization

While the CDI middleware is designed for CiteSeerX web crawling, the design approach is to define an interface that takes advantage of general open source crawlers and that can be generalized to other digital library projects which require focused crawling to feed their collections. The middleware developed is an implementation that has shown to be feasible and efficient. This middleware can be readily used by other research or development groups in digital libraries which are interested in implementing CiteSeerX or similar systems to increase their number of crawled documents. For other search engine efforts, both commercial and research, the infrastructure of CDI basically outlines any web crawling middleware which performs similar jobs.

As an example, one of the advantages of the CDI middleware is that the users do not need to re-crawl URL seeds if they just want to obtain files of different mime-types. This is especially useful for digital libraries and other focused crawling scenarios in which multiple media types are stored in separate databases and repositories. Another side effect is that content may have been updated since the last crawl and may not be chronologically consistent with documents obtained in previous crawls.

7.2 Improvements

The CDI middleware can certainly be improved. Here, we discuss three possible improvements: importing from a remote crawler repository, multiple thread processing, and adding the OCR.

The current version of CDI must be run on the crawler machine. This design requires the middleware to share hardware resource with the the crawler (both of them are big memory consumers). As a result, launching the middleware while running the web crawler may slow down the crawling progress. A direct solution is to increase the memory size, but this is bonded by some hardware restrictions such as the number of available memory sockets. A better solution is to run the CDI crawler on a separate machine, only retrieving the document files from the crawling machine. This can be implemented by adding a module to transfer remote files via the `scp` command. or using a light weight API running on the crawling server.

Another improvement is to use multiple threading to parallelize the the entire job since each document is processed independent of the others. Python supports multi-threading and offers an interface to the MySQL database to perform table locking. Applying multiple threading can significantly increase the throughput of the CDI middleware.

An additional improvement is to add the OCR plug-in to make this middleware to handle scanned PDF and postscript files. Although most academic documents created nowadays are text extractible, a significant fraction of old papers before 1990s only have scanned versions but may have high citation rates. It is then important to ingest these papers to the search engine repository.

8. SUMMARY

We introduce a CDI middleware which provides an interface between multiple crawlers and the CiteSeerX crawl repository and database. The Heritrix web crawler, FTP downloads, the Mirror, ARC and WARC input files are supported. It uses a mime type filter to select documents with desired mime types and a DCF to identify academic papers. The middleware has a web interface powered by the Django framework where the user can create, modify or submit importing jobs. The middleware also has supplemental functions such as cleaning the crawled document URL table and the parent URL table, and generating the whitelist for future web crawls. This middleware can import about 250 and 25 documents per minute with and without turning on the DCF, respectively. Although this middleware design has been motivated by the CiteSeerX digital library and search engine, such a crawler can also be used by other academic search engines.

9. ACKNOWLEDGMENTS

We thank previous CiteSeerX group members Shuyi Zheng, Juan Pablo Fernández Ramírez and Saurabh Kataria and acknowledge partial support from NSF.