A Web Service for Scholarly Big Data Information Extraction

Kyle Williams[†], Lichi Li[†], Madian Khabsa[‡], Jian Wu[†], Patrick C. Shih[†] and C. Lee Giles[†‡]
[†]*Information Sciences and Technology,* [‡]*Computer Science and Engineering*
The Pennsylvania State University, University Park, PA 16802, USA
kwilliams@psu.edu, lzl5092@psu.edu, madian@psu.edu, jxw394@ist.psu.edu, patshih@ist.psu.edu, giles@ist.psu.edu

***Abstract*—The automatic extraction of metadata and other information from scholarly documents is a common task in academic digital libraries, search engines, and document management systems to allow for the management and categorization of documents and for search to take place. A Web-accessible API can simplify this extraction by providing a single point of operation for extraction that can be incorporated into multiple document workflows without the need for each workflow to implement and support its own extraction functionality. In this paper, we describe CiteSeerExtractor, a RESTful API for scholarly information extraction that exploits the fact that there is duplication in scholarly big data and makes use of a near duplicate matching backend. The backend stores previously extracted metadata and avoids extracting metadata from a document if it has already been extracted before. We describe the design, implementation, and functionality of CiteSeerExtractor and show how the duplicate document matching results in a difference of 8.46% in the time required to extract header and citation information from approximately 3.5 million documents compared to a baseline.**

***Keywords*-Web service, information extraction, scholarly big data, CiteSeerExtractor**

I. Introduction

Scholarly big data refers to the vast amount of data produced as the result of scholarly undertaking and includes journals, conference proceedings, theses, books, patents and experimental data. This data is not only of use to scientists and researchers, but also to decision making bodies in government and education as well as the general public. Originally, three *V's* were used to describe big data. These *V's* were volume, velocity and variety [9]. Recently however, additional concepts have been added, such as value, veracity, viscosity and vulnerability. As evidence of the volume of big data, it is estimated that Microsoft Academic contains over 50 million records for academic documents and that about 43% of the articles published between the years 2008 and 2011 are freely available online [1]. As evidence of the velocity of scholarly data, it was estimated in 2010 that the annual growth rates of several popular academic databases between 1997 and 2006 ranged from 2.7 to 13.5% [10]. The variety of scholarly big data is evident from the the different types of scholarly output that is produced.

As a result of the prevalence of scholarly big data, a number of services for managing and providing access to it have emerged, such as Google Scholar[1], Microsoft Academic[2], CiteSeer$^{\chi}$[3] and the ArXiv[4]. All of these tools make use of the metadata from scholarly documents for managing and categorizing documents as well as for search. Furthermore, the metadata can enable higher level services such as those based on named entity recognition and citation matching.

Some of the services for scholarly documents, such as the ArXiv, allow users to submit scholarly documents and provide metadata while others, such as CiteSeer$^{\chi}$, collect documents by crawling the Web and perform automatic information extraction. Automatic information extraction, while less accurate than manually supplied information, is beneficial since it is a more scalable method for collecting metadata and can be applied to big data. It is possible for each service that performs automatic metadata extraction to implement its own extraction module; however, a Web-accessible API can simplify this extraction by providing a single point of operation that can be incorporated into multiple document and scientific workflows [13] so as to allow for easier processing of data. Furthermore, a single point of operation with a standard interface allows for improvements in the extraction algorithms to be used by all without the need to distribute the improvements or rewrite code in order for it to be compatible with new changes.

An important characteristic of scholarly publications is that it is common for duplication to occur. For instance, when crawling papers from the Web it is possible that co-authors might each have a version of a paper on their websites and that there may be minor differences between these versions. Similarly, differences exist between the versions of papers at publisher sites and those that authors host on their websites, such as omitted copyright notices and page numbers. Furthermore, for a Web service that performs automatic metadata extraction it is possible that different users might submit the same paper at different times. At small scale it is sufficient to perform extraction each time a paper is submitted to the Web service; however, at big data scale that may result in inefficiencies. Thus, methods for avoiding redundant information extraction are useful since

[1]http://scholar.google.com/
[2]http://academic.research.microsoft.com/
[3]http://citeseerx.ist.psu.edu/
[4]http://arxiv.org/

978-1-4799-5054-6/14 $31.00 © 2014 IEEE
DOI 10.1109/ICWS.2014.27

IEEE computer society

they can improve extractor performance.

In this paper we describe CiteSeerExtractor[5], a Web service for scholar information extraction that deals with the issue of big data by storing metadata after it is extracted. Whenever a new paper is submitted that matches a previously submitted document, this stored information is retrieved and thus unnecessary extraction is avoided. The document matching algorithm is able to deal with matches that are not bitwise identical and that might have minor differences. In describing this service, the rest of this paper is structured as follows. Section II presents related work followed by Section III, which describes the design and functionality of the RESTful API. Section IV describes the architecture of the service and Section V then presents how the duplicate document matching is performed in order to avoid redundant information extraction. Section VI presents a set of experiments that evaluate the service and, lastly, conclusions are discussed in Section VII.

II. Related Work

A few services currently exist for metadata extraction from scholarly documents. One such service is the API that runs on top of ParsCit [5]. This service allows users to submit the plain text of papers and then returns the parsed citations. GROBID [12] is a library for extracting metadata form scholarly documents. It is able to extract header metadata, citation metadata and parse the metadata. GROBID includes a RESTful API that can be used to access the service from other programs. GROBID also attempts to match extracted metadata with Crossref[6] and if core metadata, such as the title or first author, is matched, then the system attempts to retrieve the full publisher metadata. FreeCite[7] is another citation parsing Web service hosted at Brown University that is based on ParsCit. FreeCite allows users to submit a single citation string or list of citation strings and they are parsed and tagged.

Generally, existing services for metadata extraction have been designed to run on top of specific extraction tools. CiteSeerExtractor on the other hand provides a generic framework that can easily be extended to allow for additional extractors to be incorporated. This is discussed in more detail in Section IV. Furthermore, to our knowledge, none of these services specifically try to address the challenges of big data by making use of near duplicate matching.

Some tools make use of Web services to perform or improve metadata extraction showing how Web services can be incorporated into metadata extraction workflows. For instance, PDFMeat [2] converts PDF documents to text and then generates queries from the text that are submitted to Google Scholar. The search results are matched against the query document and the Bibtex entry for the best match is then retrieved. The Mendeley[8] tool for reference management supports a similar function whereby users can query Google Scholar to improve the metadata that is automatically extracted from documents when they are added to a collection. Gao et al [6] describe a similar system for using Web services to improve extracted citations. Their tool first parses citations by selecting an appropriate citation metadata extractor and, once the citation is extracted, Web services are queried to improve the quality of the extracted citations.

III. API Design

A. Resource Oriented Architecture

CiteSeerExtractor is a RESTful Web service based on the Resource Oriented Architecture (ROA) [17]. RESTful Web services have a number of benefits, such as being lightweight, scalable and easily accessible [20]. ROA is defined by four main concepts: resources, identifiers, representations of resources, and the links between resources. Furthermore, ROA has four main properties: addressability, statelessness, connectedness, and a uniform interface [17].

A *resource* in ROA is something that is important enough that it is worth being referenced. Each resource is *identified* by a URI that is unique for the resource and that allows for one of the *representations* of the resource to be accessed, where a representation of a resource is some view of that resources. An ROA application is *addressable* if information is exposed through URIs; it is *stateless* when HTTP requests are independent of each other and can happen in isolation; it is *connected* when when there are links between content; and HTTP provides a *uniform* interface [17].

1) Resources: Documents (PDF, PS, TXT) are resources in CiteSeerExtractor since they are worth being directly referenced in order to extract information from them. Resources are created by submitting a `POST` request to the extractor URL. This has the effect of creating a new document resource in CiteSeerExtractor. Once a document resource has been created, the text from the document is automatically extracted. PDFBox[9] is used to extract text from PDF documents and the *ps2txt* tool is used to extract text from PS documents. Text can also be extracted from additional file formats by incorporating the appropriate text extraction tools into CiteSeerExtractor. The successful creation of a new resource through the submission of a document and the extraction of the text is identified by a `HTTP 201 CREATED` status code, whereas if an error occurs a `HTTP 503 INTERNAL ERROR` status code is returned. Furthermore, CiteSeerExtractor can be configured to limit the submitted document size and return an appropriate message if the document size exceeds the limit. In addition to the HTTP status code, the successful creation of a resource also returns an XML or JSON document with

[5]http://citesseerextractor.ist.psu.edu/
[6]http://www.crossref.org/
[7]http://freecite.library.brown.edu/
[8]http://www.mendeley.com/
[9]http://pdfbox.apache.org/

```
<?xml version="1.0" encoding="UTF-8"?>
<CSXAPIMetadata>
<file>base_url/extractor/token/file</file>
<header>base_url/extractor/token/header</header>
<citations>base_url/extractor/token/citations</citations>
<body>base_url/extractor/token/body</body>
<text>base_url/extractor/token/text</text>
</CSXAPIMetadata>
```

Figure 1. XML returned after the creation of a resource

links to different representations of the document (shown in Figure 1 and discussed below).

2) Identifiers: Once a resource has successfully been created, it is assigned a unique and random identifier. This approach violates the ROA practice of having well-named resources; however, it simplifies the resource naming procedure and, since the resources are for the most part temporary, was considered a reasonable approach. Figure 1 shows an example of the identifier for a new resource (the string of characters trailing `extractor/` in the URL). This identifier uniquely identifies the new resource for as long as it exists and allows for representations to be extracted.

3) Representations: Representations of a resource are different views of a resource and in CiteSeerExtractor represent different types of information extracted from the original document as well as the document itself. To access a resource in CiteSeerExtractor, an HTTP `GET` request is made to http://$url/extractor/resource_id/representation, where the representations currently supported are:

- **file:** The original document that was submitted.
- **header:** The header of the document, including the title, authors, abstract, venue and any other information that may be extracted.
- **citations:** The citations extracted from the document.
- **body:** The main body text of the document, excluding the citations.
- **text:** The full text of the document as extracted by an appropriate text extraction tool.

A successful `GET` request for the representation of a resource returns an `HTTP 200 OK` status code, while an error is returned if the request fails.

4) Addressability, Statelessness, Connectedness, and Uniformity: Representations of resources in CiteSeerExtractor are addressable through their URIs based on the identifiers assigned to each resource. Resources remain addressable for as long as the system retains the stored document and extracted text file. CiteSeerExtractor is stateless as each HTTP request happens independently and is not dependent on any preceding requests. Connectedness is provided through links to representations of resources when a new resource is created and all access is provided through the uniform HTTP interface.

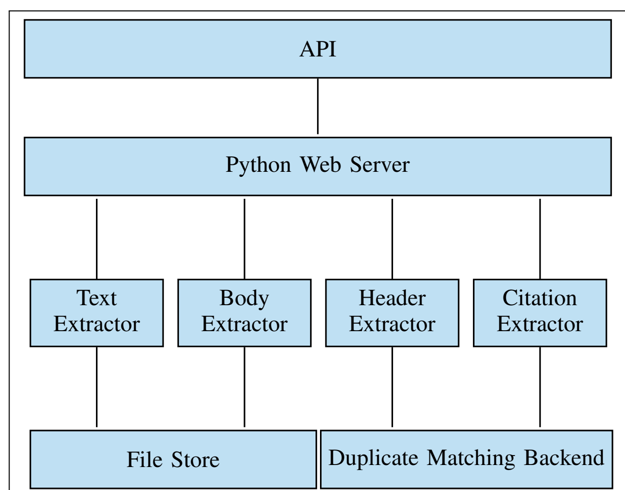

Figure 2. CiteSeerExtractor architecture

B. HTTP Methods

Table I summarizes the HTTP methods supported by CiteSeerExtractor. As can be seen from the table, the first method involves using a `HTTP POST` to create a resource. Different representations of a resource can then be retrieved with a `HTTP GET` on the representation URI. Lastly, resources can be deleted with a `HTTP DELETE` on the representation URI. These methods capture most of the functionality that one would expect when extracting information from scholarly documents. Furthermore, the API also supports different output formats, i.e., XML and JSON, for the returned data which may be useful from a processing perspective.

IV. Architecture

The overall architecture of CiteSeerExtractor was designed so as to be stand-alone, able to run in isolation, and able to be integrated with a number of services. Figure 2 shows the overarching architecture of CiteSeerExtractor. As can be seen from the figure, the RESTful API is the entry point and communicates directly with the Python Web Server, which is responsible for handling the creation of resources and for serving various representations of those resources. Security and permissions can also be controlled and implemented at the Python Web Server level. In the rest of this section, we briefly describe the technology and implementation details for each level of the architecture.

A. RESTful API

The RESTful API provides the functionality as described in Section III.

B. Python Web Server

CiteSeerExtractor is run as a stand-alone Web server and is implemented using the web.py framework[10]. The

[10]http://webpy.org/

Table I
HTTP METHODS SUPPORTED BY CITESEEREXTRACTOR

Method	URL	Description	Returns	Options
POST	/	Uploads a new PDF document either via a form or via bytestream	XML document with URIs to resource, resource_id	myfile=@filename (required for form POST)
GET	/resource_id/file	Used to download original document for resource	Document for resource	N/A
GET	/resource_id/header	Extracts the header information (authors, title, etc) from the resource	Representation of header information	output=xml (default) | json
GET	/resource_id/citations	Extracts the citations from the resource	Representation of the citations	output=xml (default) | json
GET	/resource_id/body	Extracts the body (text excluding header and citations) from the resource	Representation of the body	output=xml (default) | json
GET	/resource_id/text	Extracts the full text from the resource	Full text of resource	N/A
DELETE	/resource_id	Deletes the resource	Confirmation	N/A

Web server is responsible for the creation and removal of resources and handling all HTTP requests. The actual extraction functionality is provided by a series of independent tools that are executed by the Web server. This design approach makes it trivial to add additional representations of resources since all that is required is that the URL for the new representation is handled and the appropriate system call is made.

C. Extractors

1) Text Extractor: When a document is uploaded, a new resource is securely created on the Web server using the Python `mkstemp` command, which creates a temporary file in the most secure manner possible. Once the document resource has been created, an appropriate tool is used to automatically extract the text from the document and store the extracted text alongside the original document. We chose to use PDFBox for PDF files and *ps2txt* for PS files; however, it is possible to integrate any text extraction tool by modifying the appropriate system call in the Web server method that handles the text extraction.

2) Citation Extractor: ParsCit [5] is used for citation extraction. To extract citations, the section of text in a document containing the citations is first identified using a regular expression. Once the citation text has been identified, the citations are segmented and various aspects of each citation are tagged, such as title, authors, venue, etc. Furthermore, the context of a citation in the text is also identified. For full details of the citation fields that are classified by ParsCit and returned by CiteSeerExtractor see [5].

3) Header Extractor: The header extraction in CiteSeerExtractor is based on a tool that classifies various aspects of a header using a support vector machine [7]. As with citation extraction, the section of text containing the header is identified using a regular expression and then each line of the header is classified and various aspect of the header, such as authors, title, etc., are classified and returned. For full details on the header fields that are classified and returned by CiteSeerExtractor see [7].

4) Body Extractor: The body extraction code extracts the body of text, excluding the citations. This representation is particularly useful for text analysis where users may not be interested in the citations. The body is extracted by removing the citations from the full text.

D. File Store

Documents and their associated text representations are stored in the file store. Access to the file store is provided via the Web server and the permissions for files are configurable and set by the Web server when files are created. Resources are generally removed from the file store via an `HTTP DELETE` request on the resource ID; however, it is also possible to make use of a cron job that is run at a regular interval and removes files for which the access time exceeds some threshold. In doing this, it is possible to limit the number of resources stored on the server.

E. Duplicate Matching Backend

A NoSQL backend exists for matching near duplicates in order to avoid repetitive extraction. Since this is a major component of CiteSeerExtractor, it is discussed in detail in the next section.

V. DUPLICATE MATCHING BACKEND

As previously mentioned, it is common for multiple versions of papers to exist of the Web and possible that different users will submit the same document to the API for metadata extraction. In these cases, it is not desirable to extract information that has already been extracted and thus CiteSeerExtractor includes a near duplicate matching backend. The purpose of this backend is to store metadata that has already been extracted and retrieve the metadata

if a document submitted is a near duplicate of a document that has previously been submitted. In order for this to be successful, it is important that the overhead of performing duplicate matching does not have a detrimental effect on the performance of the system. Thus, we make use of a fast near duplicate detection algorithm for matching documents and store extracted data in an in-memory NoSQL database. The remainder of this section describes this process.

A. Near Duplicate Matching Algorithm

The simhash algorithm [3] is a state of the art algorithm for duplicate detection that maps a high dimensional feature space to a fixed-size fingerprint [14]. Our implementation of the simhash algorithm is based on that used by Manku et al [14]. The process involves calculating a hash that represents each document and then detecting near duplicates by identifying documents that have similar hashes.

For each document submitted to CiteSeerExtractor, a hash is calculated as follows. Each document is represented by a fixed fingerprint V of size f. For each token (word) t that appears in a document, an f bit hash F_t is calculated. If the i-th bit of F_t is 1, then the i-th bit of V is increased by the weight of that token. Conversely, if the i-th bit of F_t is 0, then the i-th bit of V is decreased by the weight of that token. In this study, all tokens are assigned a weight of 1. Once all tokens have been processed, V contains both positive and negative numbers that are the result of the sums of the weights of all of the tokens. V is then thresholded to create the final bit-hash and the distance between document bit-hashes can then be calculated using the Hamming distance [14].

Two documents are considered as being near duplicates if the Hamming distance between their two hashes is less than or equal to k [14]. The set of documents in a collection whose Hamming distance differs from a query document by at most k-bits can then be efficiently found as follows. For a collection for which the hash of each document has been computed, each hash is partitioned into $k+1$ sub-hashes and these sub-hashes are stored in $k+1$ tables that maintain a list of each sub-hash and the ids of documents that have that sub-hash. At query time, the hash for the query document can be partitioned into $k+1$ sub-hashes, which can be looked up in the hash tables and the matching hashes returned. This method guarantees that all hashes that differ from the query hash by k bits will be found since, in the worst case, the differing bits can only occur in k of the sub-hashes and thus one of the $k+1$ sub-hashes is guaranteed to match.

B. Implementation in CiteSeerExtractor

The algorithm described above is implemented in CiteSeerExtractor as follows. Redis[11] is used as the database that stores the generated hashes and sub-hashes as well as already extracted metadata. Redis is a key-value store NoSQL database that operates in memory and thus allows for fast access to data. For each document submitted to CiteSeerExtractor, the text is extracted and then stop words are removed and the text is stemmed using Porter's stemming algorithm [16]. Algorithm 1 shows the implementation of the algorithm for matching near duplicates.

Algorithm 1 Duplicate matching algorithm

```
1: procedure MATCHDUPLICATES(doc, metadata)
2:     simhash ← CALCULATESIMHASH(doc)
3:     data ← LOOKUP(simhash, metadata)
4:     if data ≠ NULL then
5:         return data
6:     end if
7:     subhashes ← GETSUBHASHES(subhashes)
8:     dupes ← GETMATCHES(simhash, subhashes, k)
9:     if dupes ≠ NULL then
10:        data ← LOOKUP(dupe[0], metadata)
11:        if data ≠ NULL then
12:            return data
13:        end if
14:    else
15:        ADDSUBHASHES(subhashes, simhash)
16:        data ← EXTRACT(metadata)
17:        SAVEMETADATA(simhash, metadata)
18:        return data
19:    end if
20: end procedure
```

First, the simhash of the document is calculated (line 2) and the Redis database is immediately queried to check if the requested metadata exists for that simhash (line 3). This would occur if, for instance, a document with the same simhash already had metadata extracted from it. If a match is found, the metadata is returned and no further processing or extraction need take place. If no exact match is found then the simhash is split into sub-hashes (line 7) and the Redis database is queried with the sub-hashes (line 8) to check if any full hashes exist in the database that have a Hamming distance H of at most k using the process described in Section V-A. k is set to 3 since this has previously been found to work well for academic documents [19], though when we split a simhash into sub-hashes, we split it into 3 sub-hashes (rather than $k+1=4$) so as to reduce the number of unnecessary comparisons. This results in faster processing at the cost of fewer matches. If matches are found, then the simhash of the most similar match (as measured by the Hamming distance) is used to query the Redis database for the requested metadata (line 10). If the metadata is found then it is returned and no further processing or extraction need take place. No match will be found either because **(a)** no near duplicates as measured by the Hamming distance

[11] http://redis.io/

exist, or **(b)** a duplicate does exist but the requested metadata does not, i.e., the header metadata may exist for the duplicate document but not the citation metadata. In both of these cases, the sub-hashes are then added to the Redis database (so that sub-hash matching can be performed later with this hash) and the data is then extracted (lines 15 & 16). Lastly, the metadata is saved to the Redis database (lines 17) and the data is returned.

VI. Experiments

Experiments were conducted to evaluate the effect that the duplicate matching backend has on the performance of the Web service and the quality of the duplicate matching.

A. Experiment Setup

The following experimental setup was used. All experiments were run on a machine with the following hardware and software specifications: **CPU:** 24 x Intel(R) Xeon(R) CPU X5650 @ 2.67GHz; **RAM:** 48GB; **OS:** Red Hat Enterprise Linux (RHEL) Server 5.9; **Python:** 2.7; **Redis:** 2.4.10 with a 44GB memory limit. A snapshot of the CiteSeer$^{\chi}$ collection from November 2013 was taken, which included a total of 3,577,543 million documents all of which were used for experimentation. For all documents, the text had previously been extracted using PDFLib TET[12] meaning that the Web service did not need to re-extract the text, which has the benefit of reducing the time taken to run the experiments. Documents were submitted to the API in parallel using the GNU Parallel tool [18] with 24 threads, which essentially resulted in the Web server processing 24 requests in parallel. For each document, both the header and citation metadata were extracted meaning that two calls were made per submitted document. Lastly, a filter was used that excluded documents with fewer than 100 words.

B. Duplicate Matching Overhead

The duplicate matching backend should not have a detrimental effect on the service and thus an experiment was conducted to evaluate how it affects the performance. One hundred documents were submitted to the service and the time taken to process each of the 100 documents was measured. Furthermore, when duplicate matches were found they were ignored and the metadata was still extracted so as to allow for a fair comparison. The mean time taken to extract the headers and citations from 100 files was 4.26 seconds (standard deviation=1.24) and 4.35 (standard deviation=1.25) for the baseline method with no duplicate matching and the method with duplicate matching, respectively. As can be seen from these numbers, the average time difference is about 0.1 seconds per file, which shows that the use of the duplicate matching backend does not result in a large overhead.

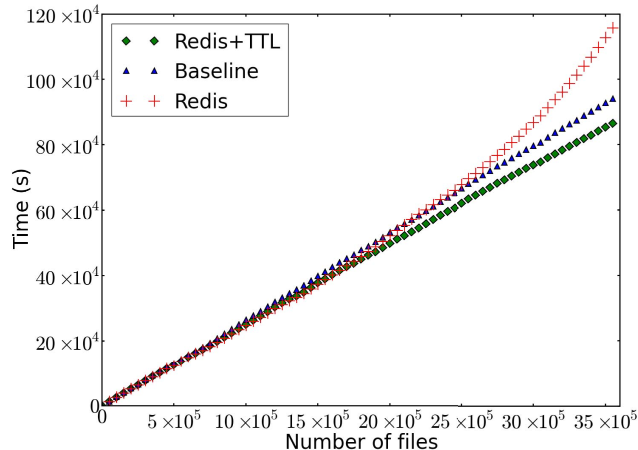

Figure 3. Time taken for extraction as the number of files increases. Times are shown for the baseline, Redis backend and Redis backend with specified TTL

C. API Extraction Performance

Figure 3 shows the incremental extraction time for the Web service as files are submitted. The baseline method refers to extracting all of the files without making use of the duplicate matching backend. As can be seen from the figure, the baseline method appears to scale linearly. The Redis method makes use of the duplicate matching backend as described in Section V. For this method, both the extracted citations and header metadata are stored for all submitted documents. As can be seen from the figure, for the first approximately 750,000 documents, the performance of the baseline and the method that makes use of the duplicate matching backend are approximately the same; however, beyond this, the addition of the duplicate matching backend leads to an improvement in performance that can be attributed to the fact that matching metadata has been found and thus extraction need not take place. The difference in performance between the baseline method and the duplicate matching backend continues to increase until about 1.5 million files at which point the difference begins to decrease with the baseline method performing better than the duplicate matching method at about 2.25 million files. This change in performance when using the duplicate matching backend can be attributed to the memory allocated to Redis becoming full. Redis was initially configured to use a maximum of 44GB, which appears to become full at about 1.5 million files. At this point, the Redis database begins to randomly select keys and delete those keys and their corresponding data using an approximate LRU algorithm. However, as this continues to happen there is an exponential decrease in performance as Redis has to continue moving data in and out of memory.

[12]http://www.pdflib.com/products/tet

Table II
EXTRACTION TIMES AND DATA SIZE FOR CITATIONS AND HEADERS EXTRACTED FROM 100 DOCUMENTS

	Citations	**Header**
Mean Time (std. dev.)	01.11 (0.29) seconds	2.86 (1.18) seconds
Total time	111.31 seconds	286.40 seconds
Size	1.4 MB	152 KB

To improve the performance of the service, the extraction process was analyzed. The time taken to extract headers and citations from 100 was measured. The documents had their metadata extracted using the standalone extraction scripts that are called by the Web service. Table II shows the mean time and standard deviation when extracting headers and citations from the 100 documents, the total extraction time for all 100 documents as well as a measure of disk usage.

Two differences between header and citation extraction can be observed from Table II. The first is that citation extraction is relatively fast compared to header extraction and the second is that the amount of memory required to store citations in the Redis database greatly exceeds the amount of memory required to store header metadata. This second observation is intuitive since an academic document only has one header whereas it usually has multiple citations. Based on these observations, the decision was made to limit the number of citations stored in the Redis database since this should result in less memory consumption while not increasing processing time as much as if header metadata was not stored. Limiting the number of citations was implemented by setting a TTL (time to live) of 6 hours on all citation metadata. When setting the TTL there is a tradeoff between performance and memory consumption since higher TTLs result in higher memory consumption but better performance due to citation metadata being retained for longer by Redis. A TTL of 6 hours was chosen in this study since it resulted in good utilization of the memory allocated to Redis; however, different values would need to be investigated for different configurations. Compression can also be used to reduce the size of the data when it is stored in the Redis database and thus the data was compressed for this experiment using the *zlib* compression library[13] with a compression level of 3. Figure 3 shows the performance of the extractor with a TTL set on citations. The standard Redis storage performs better initially since it benefits from both citations and headers being stored; however, beyond 1.5 million files the Redis+TTL storage begins to outperform the standard Redis storage. Furthermore, as the number of documents continues to increase the difference between the performance of the Redis+TTL duplicate matching backend and the baseline continues to increase with the percentage difference between the two methods being 8.46% after 3,577,000 files were processed. This translates into about 21.36 hours saved with the total running time being 10.08 days.

[13]http://www.zlib.net/

D. Verifying Results

1) Number of Documents Processed: We verify that the number of documents processed by each method is approximately the same to show that the use of the duplicate matching backend does not lead to additional failures. Possible reasons for a document not being successfully processed are: the document is too short (fewer than 100 words); the document mimetype is not *text*, *application/pdf* or *application/postscript*; the document does not pass the academic document filter; or the citation or header extraction fails. The number of documents processed for the baseline method, standard duplicate matching method and duplicate matching method with TTL for citations were 3,490,791, 3,484,213 and 3,490,799 respectively. The number of documents successfully processed with the standard duplicate matching method was about 6,000 fewer than the other methods, which can most likely be attributed to the fact that the Redis database became full leading to service failures. On the other hand, the difference between the baseline method and duplicate matching method with TTL for citations was 9 documents, with the duplicate matching method successfully processing more documents. This demonstrates that the use of the duplicate matching backend with TTL did not lead to more failures than the baseline.

2) Near Duplicates: To evaluate the extent to which near duplicate matches really are near duplicates, documents were submitted to the service for header extraction and the first 100 matches identified by the near duplicate matching backend were inspected. Of these 100 matches, 37 were exact simhash matches, i.e., no Hamming distance calculation needed to be done, and 63 matches had Hamming distances between their simhashes of at most 3. Inspecting the first 10 lines of these 100 files and comparing the titles (with minor differences allowed), it was found that 92 were true positives. Of the 8 that were false positives, it was found that in 7 of the 8 cases at least one of the documents had either large amounts of mathematical notation or large tables of numbers with the same document being falsely identified as a near duplicate 4 of the 8 times. If this is in fact the reason, then this is a weakness in the algorithm that can easily be corrected by filtering numbers from the text when calculating the simhash. The last false positive appears to have been an extended version of an existing paper. Lastly, each pair of near duplicates was found to have different SHA1 hash [15] values demonstrating that standard hashing functions are not appropriate for detecting near duplicates.

VII. CONCLUSIONS

We described a RESTful Web service for scholarly information extraction. To deal with big data, the service exploits the fact that near duplicates are a common occurrence in

academic documents on the Web and thus incorporates a duplicate matching backend, which is shown to reduce the processing time for a large collection of documents.

A possible improvement to the service would be to allow clients to set their own thresholds for the Hamming distance for two documents to be considered near duplicates. This would allow for better control of matches on the client side. The Hamming distance could also be returned in the HTTP response, which would allow a client to decide whether or not they want to keep the matched metadata or request that it is re-extracted.

It should be noted that the metadata of near duplicates identified by the near duplicate matching backend may not be exactly the same. For instance, a preprint and published version of a paper may have slightly different titles or citations. Thus, when returning extracted metadata it is possible that incorrect or partially correct metadata is returned. This is a big data tradeoff that allows for possible errors in metadata so as to achieve better performance. Once again, the extent to which this happens can possibly be controlled by allowing the client to control the Hamming distance threshold.

The design of CiteSeerExtractor is modular and easily extendable. Thus it would be trivial to extend the Web service by adding additional types of information extractors. For instance, recent work has developed methods for extracting images [4], acknowledgments [8] and tables [11] from scholarly documents and integrating this into CiteSeerExtractor could enhance the ways in which it could be used.

Acknowledgments

We gratefully acknowledge partial support by the National Science Foundation.

References

[1] E. Archambault, D. Amyot, P. Deschamps, A. Nicol, L. Rebout, and G. Roberge. Proportion of Open Access Peer-Reviewed Papers at the European and World Levels - 2004-2011. Technical Report August, European Commission DG Research & Innovation, 2013.

[2] D. Aumüller and E. Rahm. PDFMeat: managing publications on the semantic desktop. *International Conference on Information and knowledge management*, pages 10–13, 2011.

[3] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[4] S. R. Choudhury, P. Mitra, A. Kirk, S. Szep, D. Pellegrino, S. Jones, and C. L. Giles. Figure Metadata Extraction from Digital Documents. *2013 12th International Conference on Document Analysis and Recognition*, pages 135–139, Aug. 2013.

[5] I. Councill, C. Giles, and M. Kan. ParsCit: an Open-source CRF Reference String Parsing Package. In *Proceedings of the Language Resources and Evaluation Conference*, 2008.

[6] L. Gao, X. Qi, Z. Tang, X. Lin, and Y. Liu. Web-based citation parsing, correction and augmentation. In *Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries - JCDL '12*, page 295, June 2012.

[7] H. Han, C. Giles, E. Manavoglu, H. Zha, Z. Zhang, and E. Fox. Automatic document metadata extraction using support vector machines. In *Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries*, pages 37–48, 2003.

[8] M. Khabsa, P. Treeratpituk, and C. L. Giles. AckSeer. In *Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries - JCDL '12*, page 185, New York, New York, USA, June 2012. ACM Press.

[9] D. Laney. 3D Data management: Controlling data volume, velocity and variety. Meta Group. *Application Delivery Strategies*, (February 2001), 2013.

[10] P. O. Larsen and M. von Ins. The rate of growth in scientific publication and the decline in coverage provided by Science Citation Index. *Scientometrics*, 84(3):575–603, Sept. 2010.

[11] Y. Liu, K. Bai, P. Mitra, and C. Giles. Tableseer: automatic table metadata extraction and searching in digital libraries. *Proceeding of the 7thth annual international ACM/IEEE joint conference on Digital libraries - JCDL '07*, pages 91–10, 2007.

[12] P. Lopez. GROBID: Combining automatic bibliographic data recognition and term extraction for scholarship publications. *Research and Advanced Technology for Digital Libraries*, pages 473–474, 2009.

[13] S. Lu and J. Zhang. Collaborative Scientific Workflows. In *2009 IEEE International Conference on Web Services*, pages 527–534. Ieee, July 2009.

[14] G. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. *Proceedings of the 16th international conference on World Wide Web*, pages 141–149, 2007.

[15] National Institute of Standards and Technology. FIPS 180-1 Secure Hash Standard. 1995.

[16] M. F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., Dec. 1997.

[17] L. Richardson and S. Ruby. *Restful Web Services*. O'Reilly Media, 2007.

[18] O. Tange. GNU parallel-the command-line power tool. *login: The USENIX Magazine*, 3(1):42–47, 2011.

[19] K. Williams and C. L. Giles. Near duplicate detection in an academic digital library. *Proceedings of the 2013 ACM symposium on Document engineering - DocEng '13*, pages 91–94, 2013.

[20] H. Zhao and P. Doshi. Towards Automated RESTful Web Service Composition. *2009 IEEE International Conference on Web Services*, pages 189–196, July 2009.