A Cloud-based Plagiarism Detecting System

Arpan Pal
MSc Student,
Department of Computing.
University of Surrey,Guildford, United Kingdom.
ap00253@surrey.ac.uk

***Abstract*—A Plagiarism Detection System is a computer program that extracts and compares the text in the submissions to see if similarity exists to another previous submissions by other author/students that are not properly referenced/credited. To improve the accuracy of detection in these systems, it is a requirement to have a large data-set as corpus to find the most number of possible sources of a given document. For any suck system, there has always been a speed vs. accuracy issue to deal with such large data-sets. Cloud computing transforms the way of handling, consuming and managing BIG data, with promises of cost efficiencies, on-demand scale, and faster time-to-market. This paper proposes a way to detect plagiarism in a cloud based approach where the LARGE data-sets can be handled comparatively easily and will enhance the performance of detection.**

Keywords—cloud; computing; plagiarism; detection;

I. INTRODUCTION

Plagiarism or unattributed copying has always been a sensitive and puzzling issue, and also, not-so-very legal. However, to detect such offenses, the plagiarism detecting systems have been introduced. Two main classifications of such systems are attribute counting systems, which *(simplified*) namely counts the common attributes of the corpus and candidate to determine plagiarism, and structure metric systems which analyze and quantize the content structure of the document and detect plagiarism from the structure of the document itself. To detect plagiarism accurately, it is needed to have a big data-set(termed as *corpus*) that contains all the relevant topics and text from where plagiarism may be possible. The system is fed the plagiarized document(termed as *candidate*) and it looks for matches in its dictionary comprised of the aforementioned corpus to determine if any plagiarism has taken place, and displays them accordingly. Our system currently uses two approaches, a chunked hash based detection system to detect identical portions of the documents and a word based approach to find similar parts of the two documents. The systems require exhaustive pattern-matching algorithms scouring these file to find potential matches and so, common problem in these systems are the time and performance issues, if the data-set is too large, it takes time to read and iterate through them for matches. This is exactly where comes in the cloud-computing based approach, which utilizes the vast pool of computing resources available to minimize the time taken, and increase throughput. "*According to Amazon Web Services Chief Data Scientist Matt Wood, big data and cloud computing are nearly a match made in heaven. Limitless, on-demand and inexpensive resources open up new worlds of possibility, and a central platform makes it easy for communities to share huge datasets."*[4] The paper describes a Plagiarism Detection SaaS (Software as a Service) created utilizing the Amazon *EC2*[1] (Public Infrastructure as a Service) and *S3* data storage services, *Openstack*[2] (Private Infrastructure as a Service) cloud to store and process the documents to look for plagiarism and Google App Engine[3] (*GAE*, Platform as a Service) as an interface to the application. All of these are able to scale accordingly with the dataset size and client demand. The detection approaches of the system includes a chunked hash based system which computes overlapping hashes of a certain length from the sentences and matches them corpus against candidate, a word based system that detects identical word in the documents and their similarity of sentence structure. The results are displayed in a Google Chart[5] format, and are customized to produce PAN-style output, further customization to display results on the Google site, as a hit-per-document table are on the process.

II. CLOUD ARCHITECTURE/COMPONENTS

The architecture of the system comprises of a P*rivate Infrastructure as a Service cloud Openstack*, which is used as a data analyzer and processor for our application, a *public Infrastructure as a Service cloud Amazon EC2/S3* where the probable source documents are preprocessed to generate hash index and word dictionary and the *Public Software as a Service GAE*, which provides an graphical- user interface(GUI) to the system for the users. The goal is to combine the properties of both public and private IaaS clouds and a PaaS in our system and do efficient detection in a RESTful manner.

The cloud components/concepts used here are:

.*Openstack*

"Openstack is an Infrastructure as a Service (IaaS) cloud computing project that is free open source software released under the terms of theElastic Load Balancing Apache License. The project is managed by the Openstack Foundation, a non-profit corporate entity established in September 2012[1] to promote, protect and empower Openstack software and its community."[6]

In our application the hash-matching and word-matching analysis is done in Openstack. Although being a private cloud system *Openstack* is behind a firewall and cannot be accessed without self-initiation. Hence it is set up as a daemon process looking for data and processing and when given to it, otherwise sleeping soundly.

.*Amazon EC2*

"Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides re-sizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers."[7]

The application uses scalable Amazon EC2 instances to generate the hash dictionary and the word-dictionarys of corpus documents (namely hashlist.xml and wordlist.xml). It is also used to compute the interprocess documents for hash and word matching for the candidate documents. The instances are launched through GAE and Boto, the Python interface for the *Amazon API.*

.*Google App Engine*

"Google App Engine (often referred to as GAE or simply App Engine, and also used by the acronym GAE) is a platform as a service (PaaS) cloud computing platform for developing and hosting web applications in Google-managed data centers. Applications are sandboxed and run across multiple servers. App Engine also offers automatic scaling for web applications."[8]

The Application uses GAE to provide a graphical user interface to our system, the GAE implementation is also used as the topmost layer of our system containing most if not all of the controls and the display information such as ability to select documents to check, the level of plagiarism in the document etc. It also has the ability fire up ec2 instance to automate the process of generating the analysis documents (hashes and wordlists) from its web-page itself.

.*Elastic Load Balancing*

"Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances."[9]

It allows to achieve even greater fault tolerance in the applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic. Elastic Load Balancing detects unhealthy instances within a pool and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored.

.*Amazon Elastic MapReduce*

"Amazon Elastic MapReduce (Amazon EMR) is a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data. It utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3)."[11]

.*Representational State Transfer Protocol (REST)*

The REST architectural style was developed by W3C Technical Architecture Group (TAG) in parallel with HTTP/1.1, based on the existing design of HTTP/1.0 The World Wide Web represents the largest implementation of a system conforming to the REST architectural style.

Key goals of REST include *Scalability of component interactions*, *Generality of interfaces*, *Independent deployment of component*, *Intermediary components to reduce latency, enforce security and encapsulate legacy systems* .

The Architecture of the application:

The documents, both corpus and candidate are stored in S3.

For generating the dictionaries of hashes and words, these documents are downloaded into a number of **EC2** instance(s)determined by the EMR function,After this process completes, the generated files (hash dictionary and word dictionary or any of the candidate documents) are exported into respective S3 buckets.

The first time, and every time the dictionaries are generated and put into S3, the **Openstack** instances downloads them through a daemon process that keeps requesting the files. When a client wishes to check a document, Openstack gets the information by contacting GAE and it downloads the processed interpretations of the file from S3, matches them and creates the result document, which afterwards is sent back to an S3 bucket for to be displayed in GAE.

GAE displays the option to either process all of the corpus and candidate text documents (Generate Index) to get the hash and word dictionaries, or to analyze (Check) any one of the candidate documents to the processed corpus dictionaries. If the client chooses to regenerate the indexes , the GAE initiates the process by notifying EC2 instances through Boto. If t he client chooses to check, the list of the candidate documents are displayed, any one of them can be chosen to check, when done so, the GAE looks for the file in an S3 bucket, if found, then downloads it and displays the information in a Google Graph, and if not, then prepares a list for Openstack to download and process the next time it contacts GAE. The figure below is the architecture and DFD of the application.

Reasons for choosing EMR over normal EC2 instances were :

1) Easy and more intuitive to implement
2) Less time to setup custom bootstrap code
3) More efficient although a bit restricted
4) No need to administer clusters, but can resize on-the fly
5) Better support from Amazon

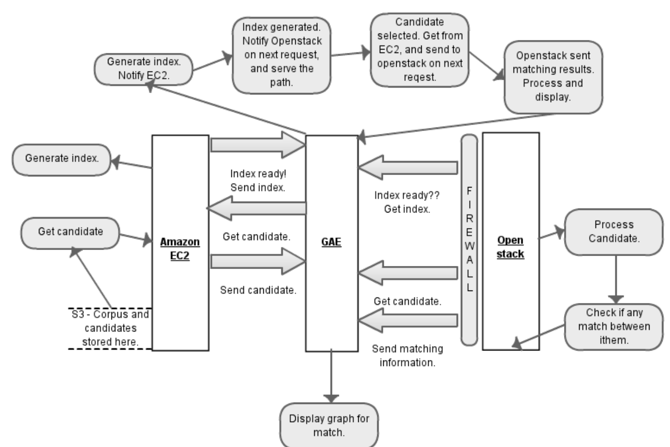

Fig. 1. *Architecture, Data Flow & Process description (basic) of the system.*[12]

III. System Components Analysis/Implementation

The application written in python detects plagiarism in given documents [a subset of clean-texts and plagiarised texts taken from PAN'11 data-sets] and displays them. The architectural description of the application is quite simple, the main application is divided into modules that would be put in each place respectively EC2, Openstack and GAE. The python, data, and dictionary files for each module in their respective folders and bash scripts to automate them.

The first working logic of the application is quite simply to match the MD5 hashes of the word groups from the clean-text(corpus) subset to the plagiarised text subset to determine if any plagiarism has taken place or not. From the text we are given as source files, we generate a index file which contains list of file-names, and MD5 hashes of the words in that file, separated into 8 word overlapping groups. The snippet of hash generation may be as follows:
[taken from */ec2/py/genhash.py*]

```
h=hashlib.md5() #--hash object
for word in wordlist:
    string="" #--set null every iteration
        while i<(j+wordlenhash):  #--default wordlenhash=8
                string+=wordlist[i]  #--word added to string
                h.update(wordlist[i])  #--word updated to hash
                                                        #--object
                i+=1
 #--iterator for making strings
        hashlist= open(destxmlpath, 'a+')
        hashlist.write("<hash>"+h.hexdigest()+
"</hash><text>"+string+"</text><filepath>"+srcpath+"</filepath
>\n") #--storing hash,text,filepath in file
        hashlist.close()
```

The application then checks for matches in the hashes in the samely processed plagiarised candidate text, identical to those of the source text, and if any are found, it is considered a hit and that source text is considered to be the source of that particular candidate text. Now we collect such hits for the particular file and display that as a graph result for how much plagiarism has taken place for that document. The snippet for this may be as follows:
[taken from */Openstack/py/hashmatch3.py*]

```
while i<=len(canddata)-1:
    candhash, candtext, candfilepath=procstrhash(canddata[i])
    hashdict=open(hashdictpath).readlines() #path to hashdict
    for line in hashdict: #while j<=len(hashdata)-1:
        corphash, corptext, corpfilepath= procstrhash(line)
        if (candhash==corphash):
                feature+="<feature
name=\"detected-plagiarism\" this_text="+candtext+"
source_reference="+corpfilepath+"
source_text="+corptext+"\"/>\n"
    hashdict.close()
    i+=1
```

The second working logic of the application is to look for similar word structure patterns (m*n) in the documents. This checks for for minimum 3 same words in in one 8-word overlapping groups made from the sentences of the document. From the source files a dictionary of all the words per file, all the files is generated. The snippet of generating such a dictionary may be as below: *[Snippet not included to save space]*

Now, to check, first the suspicious-document is processed as above, and then the output of that file is matched with all the words of one file at a time, for all the files. If atleast 3 of the words from the 8 word overlapping group (order: m) of corpus are in the 8 word overlapping group of candidate (order: n), then it is considered a hit and and the candidate text is considered to be influenced by the source text. *[Snippet not included to save space, but is similar in logic as the hash-generating snippet]*

To help and automate this process, the bash scripts *generator.sh* and *macher.sh*, and *refresher.sh* are written, the *generator.sh* runs in EC2 and downloads, passes all corpus documents in the S3 bucket to the python files to be processed, and compresses, saves the output files (the dictionaries) back in S3., this same script also generates the candidate hash and m*n-word matching files which are also put in S3 and later downloaded in Openstack. The *matcher.sh* runs in Openstack and matches each of the preprocessed candidate documents with the dictionaries and produces the result document, and sends it to S3. The *refresher.sh* refreshes the dictionaries in Openstack and looks for new documents in the corpus/candidate buckets that have not been processed yet.

IV. Results

In this application we produce two values for each document, respectively the amount plagiarised versus the original amount. These values are sent to the GAE and it uses the Google Chart API to display the amount of plagiarism in the document in GAE, as shown below,

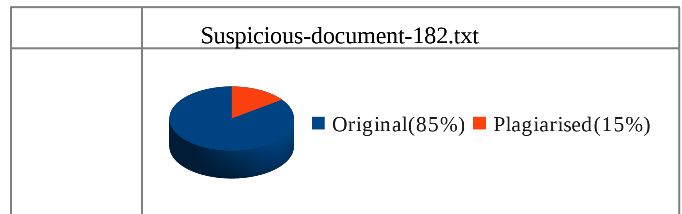

Fig 2: Output in Google Apps Engine

And we also produce two files that are the PAN-style XML documents that contain information about the possible sources and the text offset determines the beginning and end of the plagiarism in that document. The schema for such an output is given below, this can be used to prettify the display in GAE, or other further modification purposes.

```
<?xml version="1.0" encoding="UTF-8"?>
<document
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.uni-weimar.de/medien/webis/research/corpora/pan-pc-09/document.xsd"
reference="suspicious-document00482.txt">

<feature name="detected-plagiarism" this_offset="5"
this_length="1000"    source_reference="source-document00039.txt"
source_offset="25" source_length="8"/>

<feature name="detected-plagiarism" this_offset="5"
this_length="1000"    source_reference="source-document00275.txt"
source_offset="67" source_length="8"/>
</document>

. . .

. . .
```

V. IMPLEMENTATION AND TESTING

The first task was to create a non-cloud system that works and is scalable enough that when we put it on the cloud, we can expect it to do the same work, only better and faster. The implementation and small amounts of testing goes side by side, except the extensive testing that takes place after the application built.

The application is based on typically 3 modules, the GAE, EC2 and Openstack. For the local application, first the python functions to generate and match hashes were written, and then put in respective modules. Then the bash-scripts were written and all the code was debugged, cleaned and tested for independent operations.

At this point, the interface for GAE was started to build. As the application goes live as soon as we put it on the cloud, it was necessary to make sure each of the 3 modules can operate and process data independently. Most of the coding time was spent here.

Then the cloud architecture was set up, some notable tasks in this phase was configuring the AMI, Setting up the EMR function and bootstrap codes, Setting up Openstack instances, installing and configuring Boto, setting up access-rights etc.

After setting up the cloud system, the main task was to test the data transfer points and how well they work with each other, and at the end, how well the application works itself. After all that is done, the focus goes to optimizing and extending the application, and add extra things like second working logic, IO optimization, improved UI etc.

The allocated time frame is roughly 2 months(25th March to 14th May 2013) for presenting this application. The time division is as follows:

TABLE I. TIMELINE FOR IMPLEMENTATION & TESTING.

Weeks:	To Do:
25th March – 31st March	Make a non-cloud system and test the basic functionalities in own system.
01st April – 07th April	Make GAE Interface and test it locally.
08th April – 14th April	Set up clouds EC2 and Openstack, test if all functionalities are working as needed, set up access rights.
15th April – 21st April	Put application on cloud. Make appropriate test cases to test application.
22nd April – 28th April	Testing/Optimizing phase 1. Start writing submission 2.
29th April – 05th May	Testing phase 2, Keep writing submission 2.
06th May – 14th May	Finish writing submission 2.
14th May – 21th May	Extra Days.

VI. CRITICISM

The goal of this coursework is to combine the properties of a Private Infrastructure as a Service cloud, a public Infrastructure as a Service cloud and the Public Software as a Service and do simple but quite efficient plagiarism detection. However, the prospects to improve and optimize this application are very good.

The application can currently detect identical plagiarism using the chunk based hashing approach, and can identify sources from used words to a certain extent using a structure metric approach, but support for detecting complex plagiarism or obfuscation is not yet included. Also general IO or disk-read optimizations are needed to increase efficiency.

The main privacy and security issue of any automated plagiarism detecting system are protecting the intellectual property of the authors. One main criticism of the system would be that it exposes the documents publicly in the S3 buckets, although this can be partially handled by using login-authentication-authorization techniques and restricting public access to data, still data breach can happen. Same things can be said for data leakage / segregation / data-ownership / disaster recovery etc issues. One approach can be to remove the text data and keep only the hashes, but at a cost of dumbing down the quality of the results. Otherwise encryption techniques is can be implemented, but then it becomes a security vs. accessibility issue.

The current application displays a graph created by the Google Chart API display the amount of plagiarism in a document on the web-page and a PAN style XML document. But that is not enough information for all the clients. The output of the application can be prettified greatly, like adding option for client to upload his/her personal files to S3 for checking, ability to change the length of overlapping groups, highlighting the plagiarised part in the candidate document, improving the overall look of GAE, introducing HTML5, CSS etc. The latter are not in direct relation with the application, but more on the topic of web development.

VII. CONCLUSION

Plagiarism is a big problem not only for academics but also for the corporate world. The system discussed here does simple but efficient plagiarism detection, and can scale gracefully. Results also tend to be quite accurate and sufficient. Although the application is currently at its toddler times, it reveals enough potential to be considered as a step to the future.

References

[1] Amazon Web Services - https://aws.amazon.com/ [accessed Mar 25, 2013]

[2] Openstack Cloud Computing - http://www.Openstack.org/ [accessed Mar 25, 2013]

[3] Google App Engine - https://appengine.google.com [accessed Mar 25, 2013]

[4] http://gigaom.com/2012/11/30/why-amazon-thinks-big-data-was-made-for-the-cloud/ [accessed may 14, 2013]

[5] Google Chart API - https://developers.google.com/chart/

[6] Openstack - https://en.wikipedia.org/wiki/Openstack

[7] EC2 - https://aws.amazon.com/ec2/

[8] GAE - https://en.wikipedia.org/wiki/Google_App_Engine

[9] ELB - https://aws.amazon.com/elasticloadbalancing/

[10] REST - https://en.wikipedia.org/wiki/Representational_state_transfer

[11] EMR - https://aws.amazon.com/elasticmapreduce/