EECS 485 Project 5:  
Make Search Engines Great Again

Due 9 PM Friday, December 9, 2016

You have now learned enough to build a scalable search engine that is similar to commercial systems. So, in this assignment, you will build an integrated web search engine that has several features:

* Indexing implemented with MapReduce so it can scale to very large corpus sizes
* Information retrieval based on both tf-idf and PageRank scores
* A new search engine interface with two special features: user-driven scoring and deep summarization.

**IMPORTANT**: Remember not to commit the Hadoop library, vagrant folders or large files (like the Wikipedia input or your inverted index) on your Github!!! It’s 100s of MBs and you will not only be adding headaches for the staff but also for yourself as syncing that with your teammates will be painful. So don’t do it. Just don’t. Update your .gitignore ASAP.

Since your large input files won’t be on Github, use the [**scp**](http://www.hypexr.org/linux_scp_help.php) command line program to copy large input files and your new inverted index to the server.

Don’t use the EECS server you were assigned to run MapReduce. Please run your MapReduce code locally and only run the IndexServer code on the EECS server. If you accidentally start running your MapReduce code on the EECS server, kill it.

# Part 0: Setup

You will create a new virtual machine for this project. The vagrant.sh for this new VM will download Hadoop, its dependencies and all of the starter files for this project. Download the Vagrantfile and vagrant.sh into an empty directory and run the command “vagrant up” to create the virtual machine. Make sure you have a good internet connection, this install will take a little while. This will create the project with the following main files (all located in the /vagrant directory):

* .gitignore This is very important!! We are dealing with lots of very large files that should not be committed to github! They will slow down your pushes and pull tremendously so make sure to add this gitignore to your github
* /hadoop: This folder will be created as a result the Vagrant setup. It should contain all of your mapreduce code. See Part 1 for details
* /index\_server: This folder will have all of the Python code for the index server, see Part 2 for details.
* /search\_interface\_server: This folder should have your search interface app, which is given to you in the starter files, see Part 3 for details.

# Part 1: MapReduce Indexing

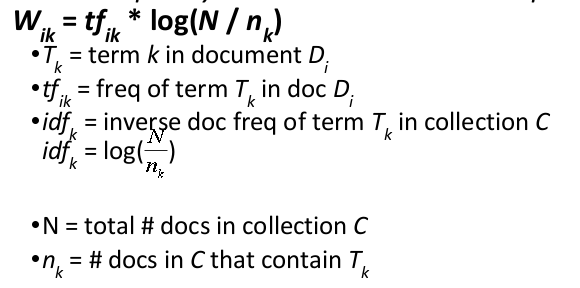
You will be building and using a MapReduce based indexer in order to process the incredibly large dataset for this project (over 300 MB!). You will be using Hadoop’s command line streaming interface that lets you write your Indexer in the language of your choice instead of using Java (which is what Hadoop is written in). However, you are limited to using Python 3 for this part of the project so that the course staff can provide better support to students.

**There is one key difference between the MapReduce discussed in class and the Hadoop streaming interface implementation: In the Java Interface (and in lecture) one instance of the reduce function was called for each intermediate key. In the streaming interface, one instance of the reduce function may receive *multiple keys*. However, each reduce function will receive *all the values* for any key it receives as input.**

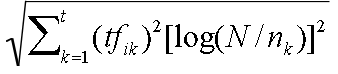
You will not actually run your program on hundreds of nodes: It’s possible to run a MapReduce program on just one machine: your local one. However, a good MapReduce program that can run on a single node will run fine on clusters of any size. In principle, we could take your MapReduce program and use it to build an index on 100B web pages.

After you have your environment setup, examine the hadoop/ directory. You will notice that there are many subdirectories, most of which are used by the hadoop library and you will not need to modify them. These subdirectories are bin/, etc/, include/, lib/, libexec/, sbin/, and share/, and are all included in the .gitignore. The subdirectories that you will be working with are data/ and mapreduce/. The data/ directory contains the map\_reduce\_input\_data. The mapreduce/ directory is where the MapReduce executables and the input/ subdirectory belong; the purpose of the input/ subdirectory is explained in the Splitter section. There is a starter map.py and reduce.py in the mapreduce/ directory which implements WordCount on the files in the mapreduce/sampleInput/ directory. We can run this job by executing run.sh. Now open the run.sh file and examine the contents to see how the script works. It is imperative that you understand this before proceeding!

For this project you will create an inverted index for the documents in map\_reduce\_input\_data through a series of MapReduce jobs. This inverted index will contain the idf, term frequency and document normalization factor as specified in the lecture slides. Sample of formatted output can be found in /hadoop/sample.txt. For your reference here are the definitions of term frequency and idf:



And here is the definition for normalization factor for a document:



## Splitter Script

Each document has three properties: **doc\_id**, **doc\_title**, and **doc\_body**. Your mapper code will be receiving input via standard in and line-by-line. As a result, the input is newline separated and each document is represented by 3 lines: 1st is the doc\_id, 2nd is the doc\_title, and the 3rd is the doc\_body. This will allow you to read the input in your mapper function easily.

Input for Hadoop programs is an input *directory*, not file. As you will notice, our input is in one large file called map\_reduce\_input\_data but your code runs with multiple mappers, and each mapper gets one input file. Consequently, you must write a custom (recommended Python) script to break the large input file with over 3,000 documents (3 lines each) into smaller files and places these files into the mapreduce/input/ directory. This splitting script will NOT be submitted to the autograder and thus you can name it whatever you please.

## Document Count Job

The first MapReduce job that you will create counts the total number of documents. This should be run with 30 mapper workers and only ONE reducer worker. The mapper and reducer executables should be named map0.py and reduce0.py respectively. The reducer should save the total number of documents in a file called total\_document\_count.txt. The only data in this file will be an integer representing the total document count. NOTE: This job will be executed using run.sh and thus the total\_document\_count.txt file will be created in the hadoop/ directory, not the hadoop/mapreduce/ directory.

## Pipeline Jobs

You will be going from large datasets to an inverted index, which involves calculating quite a few values for each word. As a result, you will need to write several MapReduce jobs and run them in a pipeline to be able to generate the inverted index. The first MapReduce job in this pipeline will get input from mapreduce/input/ and write its output to a directory that you specify in run.sh. All future jobs in this pipeline will use the output directory from the previous job as the input directory for the current job, until the inverted index is constructed. Inspect the sample run.sh, which shows how to pipe the output from one MapReduce job into the next one.

To test your MapReduce program, we recommend that you make a new test file, with only 10-20 of the documents from the original large file. Once you know that your program works with a single input file, try breaking the input into more files, and using more mappers.

Each of your MapReduce jobs in this pipeline should have **30 mappers** and **30 reducers**. However, your code should still work when the number of mappers/reducers is changed. You may only use a maximum of 9 MapReduce jobs in this pipeline (but the inverted index can be produced in fewer). The first job in the pipeline (the document counter) must have mapper and reducer executables named map0.py, reduce0.py respectively, the second job should be map1.py reduce1.py, etc.

A sample inverted index file can be found in sample.txt. In here we give you the format of the inverted index, which you should follow. The order of words in the inverted index does not matter. If a word appears in more than one doc it does not matter which doc\_id comes first in the inverted index. Note, that the log for idf is computed with base 10 and that some of your decimals may be slightly off due to rounding errors. In general, you should be within 5% of these sample output decimals.

When building the inverted index file, you should use a list of predefined stopwords to remove words that are so common that they do not add anything to search quality ("and", "the", "etc", etc). We have given you a list of stopwords to use in stopwords.txt. NOTE: This file is found in the /hadoop directory, not the /hadoop/mapreduce directory. So when opening it in your MapReduce executables, use the filename ‘stopwords.txt’, not ‘mapreduce/stopwords.txt’

When creating your index you should treat capital and lowercase letters as the same (case-insensitive). You should also only include alphanumeric words in your index and ignore any other symbols. If a word contains a symbol, simply remove it from the string. Do this with the following code snippet:

import re

re.sub(r'[^a-zA-Z0-9]+', '', word)

Your inverted index should include both **doc\_title** and **doc\_body** for each document.

To construct the inverted index file used by the index server, ‘cat’ all the files in the output directory from the final MapReduce job, and put them into a new file (i.e. cat mapreduce/final\_output\_dir/\* > outfile.txt) . You can name outfile whatever you like and move it to your index\_server/ directory to be used in the index server.

## Deliverables

* Write a custom script to convert the one giant input file into multiple smaller files.
* Build a pipeline of mappers and reducers that go from newline-separated input to an inverted index with the same exact format as sample.txt.
* **Submit**: mapreduce.tar.gz: This should be a tarball containing:
  + mapreduce/ -- this should be the folder containing mapper & reducer files but do not upload your input or output folders. This folder can also contain stopwords.txt.
    - Each map and reduce file should follow the following naming scheme: map0.py, reduce0.py, map1.py, reduce1.py,...map9.py, reduce9.py. If your MapReduce files have different names, the autograder will not be able to run them.
* Note that your input is from the hadoop/mapreduce/input/ folder and your final reducer should send the inverted index to the hadoop/mapreduce/output/ directory.

**Remember**: When implementing this part, please do not run MapReduce tasks on your assigned EECS server. Run it locally!

# Part 2: Index Server

The index server is a separate service from the search interface that handles search queries and returns a list of relevant results.

This part of the project should be written in **Python 2**. Please note that this is different from the MapReduce part, which should be in Python 3!

## PageRank Integration

In lecture, you learned about [PageRank](https://en.wikipedia.org/wiki/PageRank), an algorithm used to determine the relative importance of a website based on the sites that link to that site, and the links to other sites that appear on that website. Sites that are more important will have a higher PageRank score, so they should be returned closer to the top of the search results.

In this project, you are given a set of pages and their PageRank scores in **pagerank.out**, so you do *not* need to implement PageRank. However, it is still important to understand how the PageRank algorithm works!

Your search engine will rank documents based on both the query-dependent tf-idf score, as well as the query-independent PageRank score. The formula for the score of a query **q** on a single document **d** should be:

where **w** is a decimal between 0 and 1, inclusive. This value **w** will be a URL parameter. The final score contains two parts, one from pagerank and the other from a tf-idf based cosine similarity score. The **PageRank(d)** is the pagerank score of document **d**, and the **tfIdf(q, d)** is the cosine similarity between the query and the document. Treat query **q** as a simple AND, non-phrase query.

Integrating PageRank scores will require creating a second index, which maps each docID to its corresponding precomputed PageRank score, which is given to you in **pagerank.out**. You can do this where you read the inverted index. This index should be accessed at query time by your index server.

## Returning Hits

When the Index server is run, it will load the inverted index file, pagerank file, and stopwords file into memory and wait for queries. It should only load the inverted index and pagerank into memory *once,* when the index server is first started. Every time you send an appropriate request to the index server, it will process the user’s query and use the inverted index loaded into memory to return a list of all of the “hit” docIDs. A hit will be a document that is similar to the user’s query. When finding similar documents, only include documents that have every word from the query in the document. The index server should *not* load the inverted index or pagerank into memory every time it receives a query!

#### Search endpoint: /

Your index server should only have one endpoint, which receives the pagerank weight and query as URL parameters **w** and **q**, and returns the search results, including the relevance score for each result (calculated according to the previous section). For example, the endpoint /?w=0.3&q=hello would correspond to a pagerank weight of 0.3, and a query “hello”.

Your index server should return a JSON object in the following format:

{

“hits”: [ {

“docid”: 0,

“score”: 0.8931

},

{

“docid”: 2,

“score”: 0.6408

}

]

}

The documents in the hits array must be sorted in order of relevance score, with the most relevant documents at the beginning, and the least relevant documents at the end. If multiple documents have the same relevance score, sort them in order of docID (with the smallest docID first).

When you get a user’s query make sure you remove all stopwords. Since we removed them from our inverted index, we need to remove them from the query. Also rid the user’s query of any non-alphanumeric characters as you did with the inverted index.

Note: make sure that stopwords.txt is in the same directory as your index server executable.

The index server can be deployed on your EECS server in the same way that you deployed your photo album app in earlier projects, using gunicorn and your assigned port number. For example,

gunicorn -b class3.eecs.umich.edu:3000 -w 2 -D app:app

## Deliverables

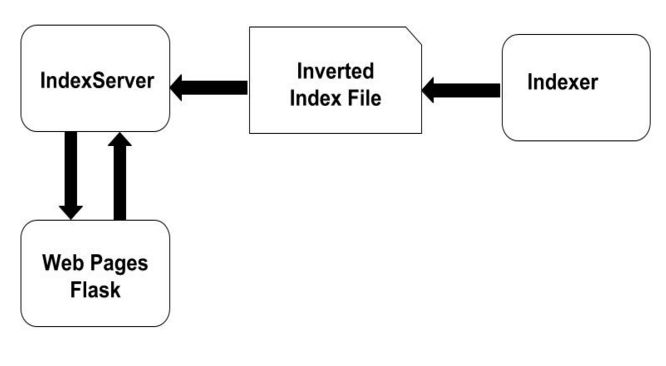
* Write an index server which uses the inverted index and pagerank files to return an appropriately formatted JSON object with a vector of documents that are similar to the search query.

# Part 3: The New Search Interface

Once you have completed the index server, you will be able to integrate it with a new HTML interface for the search engine, which we have provided for you. The search interface app will provide a GUI for the user to enter a query and specify a Pagerank weight, and will then send a request to your index server. When it receives the search results from the index server, it will display them on the webpage.

We have provided the search interface app in the search\_interface\_server/ directory, but you will need to edit config.py to match your group’s settings. Deployment instructions are included in the next section.

This part of the project should be written in **Python 2**. Please note that this is different from the MapReduce part, which should be in Python 3!



## MySQL Updates

You will need to create a new database for project 5, with a table called **Documents** as follows:

* docid - INT and PRIMARY KEY
* title - VARCHAR with max length 100
* categories - VARCHAR with max length 5000
* image - VARCHAR with max length 200
* summary - VARCHAR with max length 5000

The SQL to create this table, and load the necessary data into it is given in **wikipedia.sql**.

In order to deploy the search interface app on your EECS server, you will need to edit the search\_interface\_server/ files. You can update the host and database information in config.py as in previous projects, but your assigned port should be used for your index server, so you will need to use a different port number for the search interface app. **Set the port to be your assigned EECS server port + 1.**

## Deliverables

* Update the search\_interface\_server files as necessary to run the search interface app and observe how the search interface integrates with your index server.
  + If your index server isn’t compatible with the search\_interface\_server app, then it is a sign that your index server isn’t working as we expect it to! (And you will probably have problems on the autograder.)

# Part 4: Submission

We will post a link to the autograder when it is ready for this project. Please submit the following to the autograder:

* Your mappers and reducers, named map1.py, reduce1.py, …, map9.py, reduce9.py.
  + You can have a maximum of 9 mapreduce jobs, but you should not need that many.
  + The files *must* be named according to this pattern, or the autograder will not be able to test your mapreduce program!

Your index server will be tested via direct interactions with your deployed endpoint, and interactions with your deployed search interface app. See previous sections for deployment information.

In the README.md at the root of your repository please provide the following details:

* Group Name (if you have one)
* List the contribution for each team member:   
  User Name (uniqname): "agreed upon" contributions
* Any need-to-know comments about your site design or implementation.