CS 453: Project 3 - PageRank and Indexing

Project Report

Creepy - Indexer

Travis Hall
trvs.hll@gmail.com

Brittany Thompson
miller317@gmail.com

Bhadresh Patel
bhadresh@wsu.edu

Washington State University Vancouver
October 31, 2010

Abstract

The main goal of this project is to index terms in all of the documents, rank them, and get ready for keyword queries. Both the page ranking and the indexing is done using the MapReduce paradigm.

1 Overview

In this project, we used the MapReduce paradigm developed by Google, for page ranking and term indexing. We take the documents that have already been tokenized, stopped, and stemmed in order to make a list of all the imperative terms, as well as determine the pages importance using a Google-like page ranking algorithm called *PageRank*.

2 PageRank

There are tens of billions of web pages, which makes a difficult problem for a search engine to rank web pages effectively. One very effective approach is to use the links between web pages as a way to measure popularity and rank web pages using its popularity. Algorithms that calculates such measures based on link structure between pages are often referred to as the *link analysis algorithm*. *PageRank*, is one of the most popular link analysis algorithm used by Google search engine. We will use PageRank to compute importance of a page for our search engine. PageRank is based on the idea of a *random surfer*, where the random surfer visits a web page with a certain probability which derives from the page's PageRank. The probability that the random surfer clicks on one link is solely given by the number of links on that page. The PageRank of a page u is defined as follows,

$$PR(u) = \frac{\lambda}{N} + (1 - \lambda) \cdot \sum_{v \in B_u} \frac{PR(v)}{L_v} \tag{1}$$

where N is the number of pages and typical value for λ is 0.15. Lawrence Page and Sergey Brin have published two different versions of their PageRank algorithm in different papers. In second paper[1] they represented PageRank of a page u as follows,

$$PR(u) = (1 - d) + d \cdot \sum_{v \in B_u} \frac{PR(v)}{L_v} \tag{2}$$

where d is a damping factor and its typical value is 0.85. In both equations, B_u is the set of pages that point to u, and L_v is the number of unique outgoing links from page v.

In the first equation, PageRank of a page is the actual probability for a surfer reaching that page after clicking on many links and the sum of all pages' PageRanks will be one. Contrary, in the second equation, the probability for the random surfer reaching a page is weighted by the total number of web pages. The both version of algorithm do not differ fundamentally from each other. Hence, we will use the later version of the equation to calculate PageRank of a page in our project. The reason is that in this version calculation of PageRank are easier because we don't need value of the total number of web pages.

[1] http://ilpubs.stanford.edu:8090/361/

2.1 Basic Implementation

First, we implemented calculation of PageRank sequentially using the link analysis graph we generated in the previous project. The link analysis graph contains lists all documents, their inlinks, and outlinks. The implementation simply consumes the xml document generated by link analysis module and calculate PageRank of each document. The process is repeated until all values *converge*; *i.e*, does not change significantly between iterations.

2.2 MapReduce Implementation

Implementing calculation of PageRank in MapReduce paradigm was a rather challenging task. The implementation is done in following three stages:

2.2.1 Data Preparation

In the first stage, we process the link analysis graph and generate necessary input data for the MapReduce program. We generate multiple input files, each containing 50 documents. The format of the input files is:

```
Docid Initial_PageRank Outlinks
```

Where, `Docid` is the id of the web page, `Initial_PageRank` is the initial PageRank for the web page, and `Outlinks` is the list of all outgoing links from the web page. For example,

```
1 1.0 2,3
2 1.0 3
3 1.0 1
```

2.2.2 Iterative MapReduce

The second stage consumes the input data and iteratively calculates PageRank using MapReduce paradigm until convergence. The *mapper* task takes input as described in the previous section. For each page, it outputs the `Docid` and list of `Outlinks`. For each outlinks, it also outputs `Docid` of the `Outlinks` and its PageRank. For example,

```
1 [2,3]
2 0.5
3 0.5
2 [3]
3 1.0
3 [1]
1 1.0
```

The *reducer* task consumes the output of mapper task and calculates actual PageRank of each pages. Next, it generates the output in the same way as the input to the mapper task. Thus, at the end of the first iteration example output from reducer task is:

```
1 1.0 2,3
2 0.575 3
3 1.425 1
```

We need to repeat this process until all the PageRank values are stabilize (does not change significantly). To run the above MapReduce implementation iteratively until PageRanks are converged, we wrote a driver script that starts a streaming job flow for Amazon Elastic MapReduce. The script utilizes a Python library called *boto*[2], which provides Python interface to Amazon Web Services. The script also uses another tool called *S3cmd*[3], for communicating with Amazon S3. Upon execution, the script first uploads the PageRank calculation code and input files to Amazon S3. Next, it creates a new job flow and add `s`, typically 10, steps to it. After each `s` steps, downloads the last two iteration output and check for convergence. If PageRanks are not converged, will add another `s` steps. This process is repeated until convergence, at the end, script executes the final step as described next, downloads the final output, and terminate the job flow.

2.2.3 Final MapReduce Step

Once the PageRank calculation has been converged, one additional MapReduce step is executed to generate necessary output as required by the project. In this step, the mapper task simple prints `Docid` and `PageRank`, whereas the reducer task outputs as follows,

```
1:1.163369
2:0.644432
3:1.192199
```

3 Indexing

After the stopping and stemming is complete, the indexer is then ran to catalog all the terms in every document. The indexer is written in python and uses an inverted list structure similar to the indices in the back of textbooks. I used a nested dictionary to hold the list of terms and with each term is attached a list of pages and the occurrences of that term. On the test documents that I used, the output looks something like this: ..., 'international': '3testPage.txt': 1, '4testPage.txt': 1, '5testPage.txt': 1, 'security': '3testPage.txt': 11, '4testPage.txt': 2, ... where the term is listed along with the page that it is found on and the number of times it appears on that page.

4 MapReduce Indexing

While the previous indexer certainly works when run on a single machine, it runs into trouble when being distributed. Since our document corpus can become quite large, we clearly needed a way to distribute the creation of our indices: this is where MapReduce comes in.

One of the first things that was required for doing MapReduce was some extra processing done to the files. Since the way MapReduce works is to simply stream the file into the mapper's STDIN, we needed to find a way to tag the file with its ID. Fortunately, since all the files had, at this point, been processed and stopped and stemmed, we knew that there would not be any HTML (or XML)-like tags still within the document. The obvious answer to tagging was then to simply introduce a new tag as the first line of the document: <`file=[filename]`>. We also needed to ensure that there's a newline character at the end of each file so that when processing we only had to run a regular expression on each line instead of each word individually. All this is done in

[2] http://code.google.com/p/boto/
[3] http://s3tools.org/s3cmd

`prep_data.rb`. Obviously, doing this required a bit of extra processing time, so ideally this would be implemented into Creepy-proper and the tags written as everything else is.

Then the mapper (`lib/mapreduce/map.rb`) and Reducer (`lib/mapreduce/reduce.rb`) are fairly straight forward. Whenever the mapper sees a line containing the tag, it considers all input between that line and the next occurrence of the tag as belonging to the indicated document. From there, it simply breaks up the lines into the individual words and tallies up a count. Once completed, it outputs the various words with its list of occurrences (and their various document-based tallies) that will get sorted and fed to the reducer.

The reducer simply takes the input from the mapper and creates another hash, mapping words to (*document* : *occurrence*) pairs. It also maintains a cumulative tally this time. Finally, it sorts the hash alphabetically by key (word) and outputs in a format identical to the assignment document.

Running this through MapReduce was done in the same way as we did for homework three. In this case, the processed copies of the documents were uploaded to S3 and then the whole process was started via SSH into WSUV's CS server. Once completed. The resulting partial-indices can then be merged, split, or updated as needed in the future.

5 Roles

Travis Hall MapReduce indexing,

Brittany Thompson Indexing

Bhadresh Patel Basic PageRank, MapReduce PageRank, script to run MapReduce PageRank iteratively until convergence.

6 Test Environment

For testing/production purpose, we set up a machine instance on Amazon EC2. The instance id of the machine is `i-7d1e0d17`. The source code is checked out at `/home/ubuntu/creepy/`.

7 Usage Guide

7.1 PageRank

- To calculate PageRank sequentially using link map

  ```
  $./lib/pagerank/PageRank -l linkmap.xml
  ```

- Prepare data for MapReduce PageRank calculation

  ```
  $./lib/pagerank/PageRank.py -p storage/links.xml
  ```

- Run MapReduce PageRank iteratively until convergence (boto and s3cmd is required to run this script)

  ```
  $./lib/pagerank/run.py -c 0.0000001 -n 5
  ```

7.2 Indexing