Anthony Poerio (adp59@pitt.edu)
Andrew Masih (anm226@pitt.edu)
CS1699 – Cloud Computing
Tiny Google – Project Report

Overview

Components

Tiny Google is a simple search engine, written to run on **Hadoop**, using **Python streaming**. The project itself consists of three components:

1. User Interface (**CLI**)
2. Inverted Index (**MapReduce Job**)
3. Ranking and Retrieval (**MapReduce Job**)

This document will outline the implementation details of each component, as well the techniques used to optimize the system.

Implementations

User Interface

We chose to give **Tiny Google** a command line interface. This has the benefit of simplicity, both for the end-users and the programmers. Moreover, it fits well with our use-case scenario of demonstrating the use of Hadoop to perform complex queries for a classroom of students.

By using a CLI (command line interface), end-users gain transparency into what Hadoop is doing behind the scenes upon running a search. Feedback is immediate and verbose.

Our CLI has two options.

1. Index a document
    a. This is achieved by passing the argument **–i**, followed by the name of the document itself (assuming the document is in the same directory as **tinyGoogle.py**)
    Ex: `python tinyGoogle.py -i TheCountOfMonteCristo_by_AlexanderDumas.txt`
2. Run a Search Query

a. We store all date in the directory named **/books**.
b. Upon running a query, we will:
    i. Preprocess all documents in /books
    ii. Create an Inverted Index from these documents
    iii. Run a search query on that inverted index, with the user's search term
    iv. Return results to the user
    v. Cleanup files created during the search query, so that the user can ran another search without issue
    Ex: `python tinyGoogle.py -s the count escapes and gets revenge`

Inverted Index

The Inverted Index (II) is created via a MapReduce job that calls Hadoop Streaming. This MapReduce job works with the following methodology.

Preprocessing

Before any other actions are taken, we pre-process the data in two ways.

1. Append the file name to **each line** of a given file
2. Append a line number to **each line** of a given file

The results of this preprocessing are stored in a new directory, named **/books_preprocess.**

This is done to make the MapReduce job simpler. We need BOTH the file name AND the line numbers in our payloads. And by performing this preprocessing step, that means that each line passed into the MR job has all of the information necessary to construct a payload, all with one lookup.

The preprocessing itself is performed using **sed** and **nl** Unix commands. The script itself is found in **preprocess.sh**.

Mapper

During the MAP phase, we construct a payload in this format:

- **word\tbook_name:line_number**
- EX:
    - music BeowulfbyJLesslieHall.txt:4

This data is processed by Python using Hadoop Streaming, and passed directly the Reducer for this MR job, which aggregates this data into a **payload**.

Reducer

Our reducer then takes each of these lines and combines it into a format where we can get all data about any given word using this data structure.

Abstractly, we can think of this as:

- **word** TAB **payload**

The specific format is:

- **word**\tbook_name:total_ocurrences:[line,numbers,list],book_name:total_ocurrences:[line,numbers,list]; *…. Repeat for next book …*

The **`;` semicolon** is used as separator for each individual book, indexed by a given word. the

Here's a concrete example (first two books)

- **weather** AdventuresOfHuckleberryFinnByMarkTwain.txt:12:[10444, 10125, 6579, 2958, 1103, 5460, 2975, 6592, 8653, 4621, 2953, 4219]; DublinersbyJamesJoyce.txt:5:[3210, 604, 3473, 7339, 3713]; *…. And so on …*

This allows us to store all of the following data points for a given word, all on one line of our Inverted Index:

- Word (index)
- Book Name
- Count of occurrences in that book (need for ordering search results)
- The lines on which each occurrence appeared (need for generating context)

With all of this data intact, and in one place – our task of ranking and retrieval is greatly simplified. At any individual step, we will have all the relevant data available to our algorithm without having to look elsewhere.

Ranking and Retrieval

Overview of R&R

The relevance ranking system consisted of two main components, the term frequency and inverse document frequency. These two numbers make a numerical statistic that shows how important a word is to a document in a collection of documents. This seemed like a great choice for our implementation of the system

because that is exactly what we have, a collection of documents that we wish to run search queries on and retrieve relevant documents with context. The tf-idf works with how many times a word appears in the document, but also is offset by the frequency of the word in the collection itself. This creates a more promotional weight for search terms which helps to adjust to the fact that some words appear more frequently in general. Many variances of an algorithm using these components are used by central search engines.

Term Frequency(tf)

The term frequency is basically the raw frequency of a term in a document. We used a logarithmically scaled frequency, which is tf(t,d) = log(f,d) where f is the frequency a term appears in given d which denotes for the document.

Inverse Document Frequency(idf):

The inverse document frequency many times denoted as idf, is the measure of how much information the word itself provides. In other words, whether the term is common or rare across all documents. We used a logarithmically scaled inverse fraction fraction which is idf(t,D) = log(N/n(doc)) where N is the total number of documents, and n(doc) is the number of documents the term appears in.

Term Frequency - Inverse Document Frequency(tf-idf)

The tf-idf is calculated by multiplying the tf and idf. This calculation tends to filter out common terms. This happens because the ratio inside the's log function is always greater than or equal to 1, the value of idf is greater than or equal to 0. As a term appears in more documents, the ratio inside the logarithm approaches 1, bringing the idf and tf-idf closer to 0.

Our Implementation

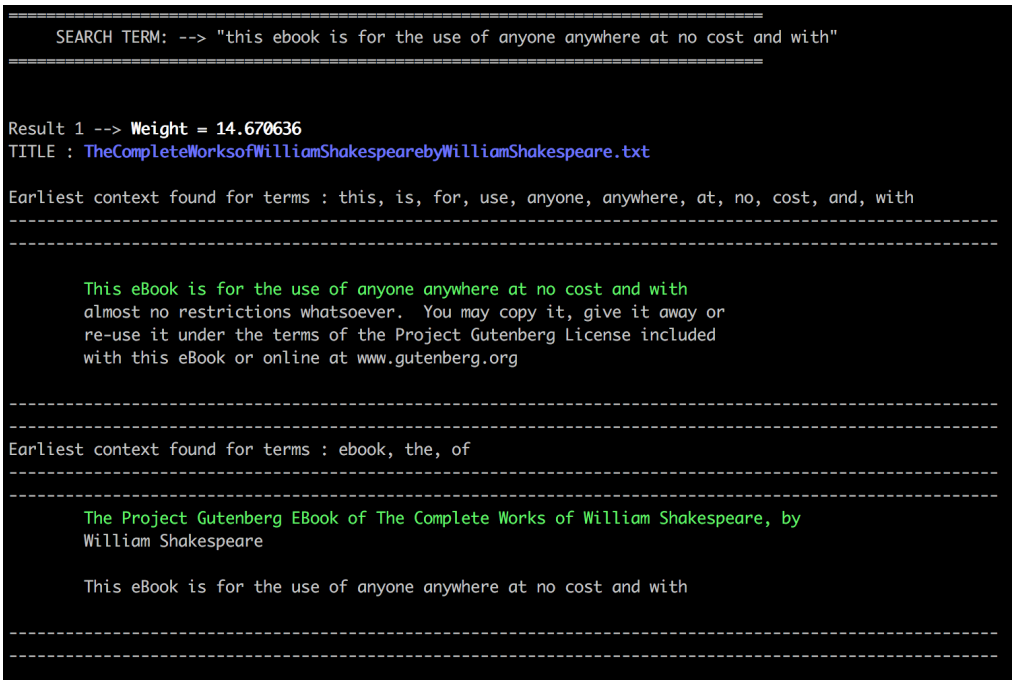
The implemented system basically creates a weight for each search term using the tf-idf, and adds them together for each document. This results in a weight for each document for a string of given search terms. This weight is use to rank the documents from the collection, and the top three weights are displayed with their given context.

How it Works

After calculating the weight for each term and adding them together to create weight per document based on the search term. The program prints out the context for the search term. So the way it prints out the context is that, it finds the earliest context seen for given set of terms and prints them out.

For Example.

Given search term : “this ebook is for the use of anyone anywhere at no cost and with” which is a phrase that appears in one of the documents exactly, the context will be shows as followed.



As you may be able to see, it first shows the list of terms who have the same context and it shows the earliest context found for it. Thought the terms “ebook”, “the”, and of “of” appear even before that whole phrase, it shows the context for those terms separately. This shows the book with the most weight which is 14.670.

The next result shows the next highest weight which is 14.0111, with its respective context.

```
Result 2 --> Weight = 14.011681
TITLE : TheCountOfMonteCristo_by_AlexanderDumas.txt

Earliest context found for terms : this, ebook, is, for, use, anyone, anywhere, at, no, cost, and, with
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------

        This eBook is for the use of anyone anywhere at no cost and with almost
        no restrictions whatsoever.  You may copy it, give it away or re-use it
        under the terms of the Project Gutenberg License included with this
        eBook or online at www.gutenberg.org

------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
Earliest context found for terms : the, of
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
        Project Gutenberg's The Count of Monte Cristo, by Alexandre Dumas, père

        This eBook is for the use of anyone anywhere at no cost and with almost
        no restrictions whatsoever.  You may copy it, give it away or re-use it

------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
```

The next and last of the top three results will show it's respective context.

Optimizations

Search Query Optimizations

Our biggest optimization was made in the **Ranking and Retrieval** MapReduce job. We were able to limit the amount of data that needed to be analyzed in our **R&R** algorithm by:

1. Within Python Streaming, ONLY allowing lines which match one of the user's keywords to enter our R&R algorithm
    a. This cuts down the amount of work the R&R algorithm must do in subsequent steps dramatically
    b. The algorithm will ONLY do as much work as minimally necessary with this setup

2. Keeping the **line numbers** for each word occurrence WITHIN our payload itself
    a. This means that when we need to display context for our users, we can go directly there in our source files, and ignore all other data in those files, which is irrelevant to our search