**Running PageRank on Wikipedia**

**Goals:**

1. Understand the PageRank algorithm and how it works in MapReduce
2. Implement PageRank and execute it on a large corpus of data
3. Examine the output from running PageRank on Wikipedia to measure the relative importance of pages in the corpus

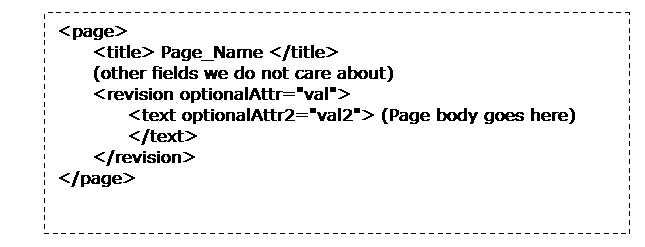
**PageRank:[100]**

This lab is a more involved process containing several different MapReduce passes used sequentially. The input to the program are pages from the English-language edition of Wikipedia. Rather than operating on individual text file inputs, however, it is modified to work on the Wikipedia data, which is not stored in a large number of flat text files.

Storing large numbers of individual files in the Hadoop DFS is inefficient. The DFS can hold large amounts of data (many terabytes), but expects this data to be primarily in a small number of very large files. The English-language Wikipedia corpus is about 15 GB, spread across 2 million files. If we were to load these files into the DFS individually, and then run a MapReduce process on this, then Hadoop would need to perform 2 million file open--seek--read--close operations: very time consuming!

Instead, the pages are stored in an XML format, with several (many thousands) of pages per file. This has been further preprocessed such that all the XML for a single page is on the same line. This makes it easy for us to use the default InputFormat, which performs one map() call per line of each file it reads. The mapper will still perform a separate map() for each page of Wikipedia, but since it is sequentially scanning through a small number of very large files, performance is much higher than in the separate-file case.

Each page of Wikipedia is represented in XML as follows:



**MapReduce Steps:**

This presents the high-level requirements of what each phase of the program should do: (While there may be other, equivalent implementations of PageRank, this is a straightforward one we can use for this lab.)

**Step 1: Create Link Graph**

The PageRank algorithm operates by distributing PageRank value from pages to their neighbors by traversing the link graph representation of the pages. This step parses all the page-to-page links out of the page bodies, and maps each page to a list of the pages it links to. The reducer is simply the identity function, as no data relevant to a particular page needs to be combined in from other pages: each page and its out-links is a self-contained unit.

As for the mapper:

         The input to the mapper is a key consisting of the byte offset into this large XML file of the current line, and a value consisting of a single line of flattened XML.

         The output of the mapper should have the page name (between <title>....</title>) as its key.

         The output value should be all of the links extracted from the line of text, as well as an initial PageRank value for the page equal to "d". (d is the "random jump probability"; use the value 0.15) The format of this output value should be something you can easily parse into its individual components, later.

         Links are represented in the input as either "[[TargetPageName]]" or "[[PageName|DisplayText]]". You should discard any display text.

         Note that we are uninterested in the text of the article: while this is useful for indexing, this is orthogonal to PageRank.

**Step 2: PageRank Distribution**

The actual algorithm itself is encoded in the PageRankMapper and PageRankReducer classes. Because PageRank is an iterative algorithm, this MapReduce task is run several times over.

         You do not need to determine end criteria for the PageRank algorithm; just use a fixed loop of 10 iterations in your driver

         The mapper receives as input a slice of the link graph:

  The key is the page name

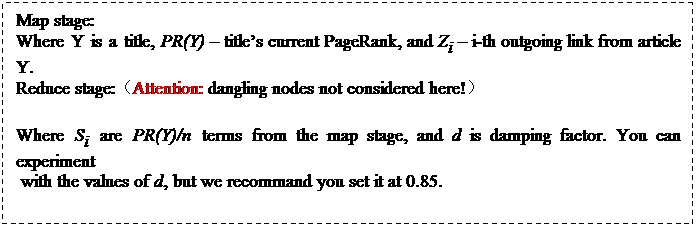
  The value is the outgoing page list and PageRank value from the previous iteration

   (This is the same format as you emitted in step 1)

The mapper receives as input a Text object representing a slice of the link graph: one page and all its outbound links, as well as its current PageRank (carried over from the previous iteration, or an initial seed value). It then allocates equal fractions of the current PageRank of the input page to each of its outbound links. The outputs of the mapper are pairs of the form: <TargetPageName, PageRankFragment>. Because we want to preserve the link graph for future iterations, we also emit the link graph slice to our own PageName.

The reducer receives as input everything with the same PageName key. This is a set of PageRankFragments -- floating-point values representing fractions of the PageRanks of all pages which point to the current PageName -- as well as the outbound link set for this PageName. The reducer sums these fragments and calculates the new PageRank for the current page using the formula *(1 - D) + D\*Sum(PR\_fragment\_inputs)*. It then formats a final output string using the same format as the mapper input, incorporating the link graph slice and the current PageRank.

**Hint for writing a MapReduce program to perform a single iteration of PageRank.**



\*Dangling Node: The page which doesn’t have any outgoing links.

**Step 3: Cleanup and Sorting**

The goal of this lab is to understand which pages on Wikipedia have a high PageRank value. Therefore, we use one more "cleanup" pass to extract this data into a form we can inspect. Write a PageRankCleanupMapper to do this step.

The mapper in this step receives a <PageName, (PageRank, OutboundLinkList)> datum, just like the PageRank calculator. It emits as output <PageRank, PageName>, discarding the link list. Note that the PageRank is now the key, and PageName is the datum. This implicitly sorts the values by PageRank, as keys passed to a reducer are processed in sorted order. The reducer is simply the identity function.

At this point, the data can be inspected and the most highly-ranked pages can be determined.

**Exercise:**

Implement the PageRank algorithm described above. You will need a driver class to run this process, which should run the link graph generator, calculate PageRank for 5 iterations, and then run the cleanup pass. Run PageRank, and find out what the top fifty highest-PageRank pages are.

Some important points:

* Each pass will require its own input and output directory; one output directory is used as the input directory for the next pass of the algorithm.
* Remember that you need to remove your intermediate/output directories between executions of your program.
* The PageRank for each page will be a very small floating-point number. You may want to multiply all PageRank values by a constant 10,000 or so in the cleanup step to make these numbers more readable.

Testing your program:

1. Scottish wiki-small: 8MB - [http://harvard-cs264.s3.amazonaws.com/scowiki-20090929-one-page-per-line.gz](http://harvard-cs264.s3.amazonaws.com/scowiki-20090929-one-page-per-line.gz" \t "_blank)
2. Africa wiki-small: 67 MB - [http://harvard-cs264.s3.amazonaws.com/afwiki-20091002-one-page-per-line.gz](http://harvard-cs264.s3.amazonaws.com/afwiki-20091002-one-page-per-line.gz" \t "_blank)
3. English wiki-big: 2 GB - [http://harvard-cs264.s3.amazonaws.com/enwiki-20090929-one-page-per-line-part3.gz](http://harvard-cs264.s3.amazonaws.com/enwiki-20090929-one-page-per-line-part3.gz" \t "_blank)

|  |  |
| --- | --- |
|  | **Sun, Dec 09, 2012 -- AWS EMR mini-HW steps**  Jeetendra has detailed the EMR miniHW steps. You could follow this for the miniHW.  ------------------------------------------------------   Requirements: Java 1.6 (Might not work with Java 1.7), an AWS account and access to Elastic Mapreduce(EMR) service. You must be signed up for EC2 and S3 to access EMR (you will be prompted to do so if you are not).   IDE used: Eclipse Juno(Not mandatory)   STEP 1: Creating MapReduce project  Create a new java project- WordCount. Create a new class WordCount.java inside package org.mapreduce. A sample code for wordcount program is available at <http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html.> You can download the JAR- hadoop-core 1.0.3 from <http://mvnrepository.com/artifact/org.apache.hadoop/hadoop-core/1.0.3.>   STEP 2: Creating a custom JAR file for AMAZON EMR  Now create a JAR file of the project.  Optional: You could also build an executable JAR file by specifying the main class in manifest. Those of you using eclipse will have an option of specifying the main class while using the export feature.  We will call it WordCount.jar   STEP 3: Downloading sample data  Once done visit <http://www.gutenberg.org/> and download some sample ebooks in Plain Text UTF-8 format.   STEP 4: Creating Amazon S3 buckets  Now visit the s3 console at <https://console.aws.amazon.com/s3/home> and create bucket(s) for your JAR file and sample input.  For example:  -Create a bucket with name emr.programs  -Create a folder- jars inside the bucket  -Upload your WordCount jar file in jars.   -Create another bucket emr.data  -Create a folder wordcount in the bucket.  -Create folder input inside the folder wordcount  -And upload all your text files.   You don't need to create an output folder- it will be automatically created.   STEP 5: Creating an EMR job flow  Goto EMR console at <https://console.aws.amazon.com/elasticmapreduce.> And click Create New Job Flow.   (i) Job flow name: WordCount  Hadoop version 1.0.3(Amazon Distribution)  Select- Run your own application and choose Custom JAR feature in job type.  click continue   (ii) JAR location: emr.programs/jars/WordCount.jar  JAR arguments: org.mapreduce.WordCount s3n://emr.data/wordcount/input s3n://emr.data/wordcount/output  click continue   JAR arguments:  The JAR arguments are input parameters to your custom JAR.  parameter 1: Main class where MapReduce job was written  parameter 2: Input path (The folder of all the text files)  parameter 3: Output path (This will be auto-generated)   Note: If you added the main class in manifest of the JAR just add in the arguments. You don't need parameter1 in that case.   Rest of the steps can be directly proceeded.  Optional log path: You may want to give Amazon s3 log path under advanced options for a log file of your job. The logs would be automatically created once the job finishes(fails or succeeds). Example: s3n://emr.programs/logs/wordcount (don't use emr.data)   STEP 6: Review  Review your JOB flow and click on create job flow.   STEP 7: Check the output path  After your job finishes(may take 5-10 min for a couple of txt files). Visit your s3 console and inside your bucket- emr.data check output directory wordcount/output. You will see files with filenames starting with "part-". Download the files and verify the output. |