THE WEB LABORATORY: WEB GRAPH GENERATION

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

By

Manu Jain

Project Advisor: Professor William Y. Arms

Degree Date: August 2008

ABSTRACT

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: The Web Laboratory: Web Graph Generation

Author: Manu Jain

Abstract: *The Web Laboratory is a joint project of Cornell University and the Internet Archive to provide data and computing tools for research about the Web and the information on the Web. The early users of this research are mostly computer scientists with interests in the structure and evolution of the Web, and social scientists who see the Web as both a fascinating social phenomenon and a source of information about contemporary social events [1]. The Web Lab is built on the historical collected snapshots of the Web by the Internet Archive since 1996 and consists of various teams working on specific tasks such as the Web Graph Generation, Page Rank Calculation, Anchor Text Analysis, Data Movement and Tracking, etc. This particular work deals with the generation of Web Graph, which is the linking structure of the Web and is represented by its adjacency matrix using a compressed sparse row representation. Web graphs allow researchers to see the structure and evolution of the Web and are increasingly used for Page Rank Calculation and Social Network Research, etc. The major accomplishments of the project for this semester includes the improvement of the algorithm for Web Graph Generation so as to make it more time effective and less resource intensive, integration of the Web Graph Generation application with Page Rank Calculation program to make both functionally compatible and combination of the output of Web Graph Generation and Page Rank Calculation programs to obtain the final output in the desired format. The application is also successfully migrated to a new cluster setup with a higher version of Hadoop, specifically 0.17.1 from the previous version of 0.15.3. The system design, implementation and results are discussed in detail in the report.*

Report Approved by

Project Advisor: ____ Date: ______________

EXECUTIVE SUMMARY

The previous Web Graph Generation algorithm has been significantly improved as part of this project. In this work, the algorithm of the first step of the process has been completely modified in order to effectively utilize the resources and perform certain in-memory computations, rather than writing all the data to a local hard-drive, which enhanced the performance drastically as measured through experimental results. The total time taken to successfully complete the StepOne for *Universities Links*, around 4.52 GB of data, has been brought down from approximately 180 mins to the current time of only around 10 mins with this new algorithm.

The application has also been modified to get rid of the sequential indexing step in which all the data had to be copied over to a single local machine and tagged with a serial integer identifier. This was a cumbersome manual process consisting of copying the output files of StepOne from the Hadoop DFS to the local machine, compiling and running the C++ code on the local machine, then copying the output files of this local StepTwo back to Hadoop DFS. This back and forth copying of files was extremely time consuming while dealing with massive data sets. Also, this step cannot utilize the advantages of a distributed system and had to depend on the computational constraints of a single machine.

To make the output of Web Graph Generation compatible with Page Rank Calculation program, the StepThree of the process was significantly changed to compute the value of the total number of records present in the input file using JobConf attributes, which was required to evaluate the approximate value of Page Rank and provide that information to the Page Rank Calculation program for its effective functioning.

The output produced by the Web Graph Generation and the Page Rank Calculation programs, after completing all the specified iterations, were combined together to produce the final output in the specified format (#u, p, u), # u is the FNV hash code assigned to fromURL, p is the computed pagerank of the fromURL and u is the fromURL itself. This algorithm was designed keeping in mind that the input files for this job are not in the exact same format.

The application was successfully migrated to a new cluster setup in this semester which has a higher version of Hadoop and JRE installed, specifically 0.17.1 up from the previous version of 0.15.3 which has a default -Xmx value of 8 GB while configuring the Java Heap Space for JVM.

Contents

1. Introduction 5
2. Problem Statement 6
   2.1 Web Graph Generation 6
   2.2 Integration of Web Graph Generation with Page Rank Calculation 7
   2.3 Combination of Web Graph Generation and Page Rank Calculation Results 7
   2.4 Migration to the new cluster with Hadoop Version 0.17.1 8
3. Computing Environment 8
4. Process Flow 10
   4.1 Web Graph Generation 11
      4.1.1 StepOne 11
      4.1.2 StepTwo 12
      4.1.3 StepThree 13
   4.2 Integration of Web Graph Generation with Page Rank Calculation 14
   4.3 Combination of Web Graph Generation and Page Rank Calculation Results 16
      4.3.1 StepOne 16
5. Issues Faced / Solutions 17
   5.1 Lack of Memory Space for JVM 17
   5.2 Lack of Replication in the cluster 18
   5.3 Integration problems with Page Rank Calculation 19
6. Experimental Results 19
7. Code Compilation and Running 21
   7.1 Web Graph Generation 21
   7.2 Integration of Web Graph Generation with Page Rank Calculation 22
   7.3 Combination of Web Graph Generation and Page Rank Calculation Results 23
   7.4 Migration to the new cluster with Hadoop Version 0.17.1 23
8. Conclusion 24
9. Acknowledgement 24
10. References 25

Appendix 1: Source Code 26

1. Introduction

The Web Laboratory is a joint project of Cornell University and the Internet Archive to provide data and computing tools for research about the Web and the information on the Web. The Web Lab is built on the historical collected snapshots of the Web by the Internet Archive since 1996 and provides in-depth access for researchers who wish to analyze this collection in a much greater depth. It consists of various teams working on specific tasks such as the Web Graph Generation, Page Rank Calculation, Anchor Text Analysis, Data Movement and Tracking, etc.

This particular work deals with the generation of Web Graph, which is the linking structure of the Web and is represented by its adjacency matrix using a compressed sparse row representation. Web graphs allow researchers to see the structure and evolution of the Web and are increasingly used for PageRank calculation and Social Network research, etc. This work builds upon the previous work done in Fall 2007 by Anthony Jawad and Jie Teng [2]. They worked on the development of the tools used for generating a sparse matrix representation of an arbitrary subset of the web graph that is suitable for further computation; ensure its correctness, and access and improve the scalability of the process to arbitrarily large data sets.

The goal of the project for this semester has been to improve the algorithm for Web Graph Generation so as to make it more time effective and less resource intensive, integrate the Web Graph Generation application with Page Rank Calculation program to make both functionally compatible, combine the output of Web Graph Generation and Page Rank Calculation programs to obtain the final output in the desired format and migrate the entire application to the new cluster setup with a higher Hadoop and JRE version of 0.17.1.

The complete report is organized into various specific sections as follows. In Section 2, a formal description of the problem statement is provided with specific notations and brief background about web graphs in general. Section 3 provides information of the computing environment used for carrying out this project – Hadoop software framework with Map/Reduce programming methodology and distributed file system on a cluster of Linux machines. In Section 4, the step by step process of generating a Web Graph, integration of Web Graph Generation with Page Rank program and combination of Web Graph Generation and Page Rank results to obtain the final desired output is provided in detail. The major issues faced in the above processes are provided with their respective solutions in Section 5. Experimental results and their through analysis with graphs / charts is provided in Section 6. In Section 7, the methodology of code compilation and execution is described with actual command statements and instructions.

2. Problem Statement

2.1 Web Graph Generation

The Web Graph is a directed graph in which each node V is a single web page and each edge E is a hyperlink from one web page to another. Each node is identified by a Uniform Resource Locator or URL whereas each edge is represented by an ordered pair of URLs whose first component 'fromURL' identifies the page containing the link and whose second component 'toURL' identifies the page to which the link is targeted. The input used to generate the Web Graph is an ASCII encoded tab-separated-value (TSV) file, where each line is a hyperlink tuple, expressed in the form 'fromURL <TAB> toURL' which are extracted from various web crawls.

With the above considerations in mind, the problem of generating the web graph is formulated as below:

Notation

(*u*', *v*'): a URL pair where *u* 'and *v*' are two URLs in uncanonicalized form.
u' is called the *fromURL*, *v*' is called the *toURL and* the page identified by *u*' contains an outgoing link to *v*'.
u, *v:* canonicalized forms of *u*', *v*' respectively.
i, *j*: integer indices assigned to nodes.
(*i*, *j*): an edge linking from node *i* to node *j*, where the nodes are represented by their indices.
u, # *v:* FNV hash code values assigned to nodes.
(# *u*, # *v*): an edge linking from node # *u* to node #*v*, where the nodes are represented by their FNV hash codes.
d is the outDegree i.e. number of links coming out of the *fromURL*
o is the output count i.e. total number of records present in the input file

Previous Problem Statement

Given a set of URL pairs in uncanonicalized form (*u*', *v*'):
a) Canonicalize every *u*' and *v*'.
b) Create a list of all nodes of the graph, i.e., the set of all unique u. And create a list of all edges, i.e., discard all duplicates and all (*u*, *v*) pairs where v is not a node of the graph.
c) Create an index list for the nodes, (*i*, *u*), where the *i* are consecutive integers.
d) For each node, with index *i,* create a list of the edges from that node, (*i*, *j*), where *j* is the set of nodes that have edges from node *i*.
e) Sort the set of (*i*, *j*) by *i* and then by *j,* and output them as the final output.

New Problem Statement

Given a set of URL pairs in uncanonicalized form (*u*', *v*'):

a) Canonicalize every *u*' and *v*'.

b) Create a list of all nodes of the graph, i.e., the set of all unique u. And create a list of all edges, i.e., discard all duplicates and all (*u*, *v*) pairs where v is not a node of the graph.

c) Generate hash codes for all the nodes, (# *u*, *u*), where the # *u* are the respective hash code values for the URLs.

d) For each node, with hash code # *u*, create a list of the edges from that node # *v*, where # *v* is the set of nodes that have edges from node # *u*, determine the total number outDegree d from each #u, and output in the format (#u, d, #v)

e) Determine the total number of records o present in the input file, generate two separate output files: (#u, u) and (#u, (1/o), d, #v) both of which have total number of reduce tasks that are running set to be 1 through the JobConf class, so that a single globally sorted final output file is created for each.

2.2 Integration of Web Graph Generation with Page Rank Calculation

The final output file from the Web Graph Generation should be created in the specified format i.e. (#u, (1/o), d, #v) as required by the PageRank algorithm to function correctly. The compatibility of this file should be tested as an input to the PageRank program and analysis must be carried out to determine the correctness of the output file produced by PageRank using this input file.

2.3 Combination of Web Graph Generation and Page Rank Calculation Results

The output produced by Web Graph Generation and the Page Rank Calculation, after completing all the specified iterations, are combined together to produce the final output in the specified format. The algorithm is designed keeping in mind that the input files for this job are not in the same format.

With the above considerations in mind, the problem of combining the Web Graph Generation and Page Rank Calculation results to obtain the final output file is formulated as below:

Notation

(#u, u): # u is the FNV hash code assigned to fromURL and u is the fromURL itself.

(#u, p, d, #v): # u is the FNV hash code assigned to fromURL, p is the computed pagerank of the fromURL, d is the outDegree i.e. number of links coming out of the fromURL and # v is a set of FNV hash code values assigned to all the toURLs

(#u, p, u): Format of the required final output

Problem Statement

Given the input in the form of *(#u, u*) and *(#u, p, d, #v):*

a) Parse out #u, u and p from the input in the Map phase and pass it to mapper output collector in the format (#u, u) and (#u, p)

b) Determine the u and p values for every input key #u and append them together in a tab separated manner to create a string.

c) Output (key, newstring) i.e. (#u, p, u) through the reducer output collector and set the total number of reduce tasks that are running to be 1 through the JobConf class, so that a single globally sorted final output file is created.

2.4 Migration to the new cluster with Hadoop version 0.17.1

The entire application needs to be migrated to a new cluster setup, with a higher version of Hadoop and JRE installed, specifically 0.17.1 up from the previous version of 0.15.3. The code should be made compatible for the newer version of Hadoop and JRE and should utilize the high performance features of this latest version in order to achieve better optimization of the complete cycle.

3. Computing Environment

The entire design and architecture of this work is based on an open-source software framework called Hadoop. Since the application deals with the processing of huge data sets in a reasonable time frame, a distributed environment is required which provides for the computation to take place simultaneously over a cluster of machines and deals with the complexities of replication, resource management and monitoring, failure detection and recovery, load balancing and service discovery. Hadoop is ideal for such an environment and provides the following key advantages:

Scalable: Hadoop can reliably store and process petabytes.

Economical: It distributes the data and processing across clusters of commonly available computers. These clusters can number into the thousands of nodes.

Efficient: By distributing the data, Hadoop can process it in parallel on the nodes where the data is located. This makes it extremely rapid.

Reliable: Hadoop automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

Hadoop is a framework for running applications on large clusters consisting of commodity hardware and transparently provides applications both reliability and data motion. Hadoop

implements a computational paradigm named map/reduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system HDFS that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both map/reduce and the distributed file system are designed so that node failures are automatically handled by the framework [3].

Map/Reduce is a programming paradigm that expresses a large distributed computation as a sequence of distributed operations on data sets of key/value pairs. The Hadoop Map/Reduce framework harnesses a cluster of machines and executes user defined Map/Reduce jobs across the nodes in the cluster. A Map/Reduce computation has two phases, a map phase and a reduce phase. The input to the computation is a data set of key/value pairs.

In the map phase, the framework splits the input data set into a large number of fragments and assigns each fragment to a map task. The framework also distributes the many map tasks across the cluster of nodes on which it operates. Each map task consumes key/value pairs from its assigned fragment and produces a set of intermediate key/value pairs. For each input key/value pair (K, V), the map task invokes a user defined map function that transmutes the input into a different key/value pair (K', V’).

Following the map phase the framework sorts the intermediate data set by key and produces a set of (K', V’*) tuples so that all the values associated with a particular key appear together. It also partitions the set of tuples into a number of fragments equal to the number of reduce tasks.

In the reduce phase, each reduce task consumes the fragment of (K', V’*) tuples assigned to it. For each such tuple it invokes a user-defined reduce function that transmutes the tuple into an output key/value pair (K, V). Once again, the framework distributes the many reduce tasks across the cluster of nodes and deals with shipping the appropriate fragment of intermediate data to each reduce task.

Tasks in each phase are executed in a fault-tolerant manner; if node(s) fail in the middle of a computation the tasks assigned to them are re-distributed among the remaining nodes. Having many map and reduce tasks enables good load balancing and allows failed tasks to be re-run with small runtime overhead.

The Hadoop Map/Reduce framework has master/slave architecture. It has a single master server or jobtracker and several slave servers or tasktrackers, one per node in the cluster. The jobtracker is the point of interaction between users and the framework. Users submit map/reduce jobs to the jobtracker, which puts them in a queue of pending jobs and executes

them on a first-come/first-served basis. The jobtracker manages the assignment of map and reduce tasks to the tasktrackers. The tasktrackers execute tasks upon instruction from the jobtracker and also handle data motion between the map and reduce phases.

Hadoop's Distributed File System is designed to reliably store very large files across machines in a large cluster. It is inspired by the Google File System. Hadoop DFS stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file. Files in HDFS are "write once" and have strictly one writer at any time.

Like Hadoop Map/Reduce, HDFS follows master/slave architecture. An HDFS installation consists of a single Namenode, a master server that manages the filesystem namespace and regulates access to files by clients. In addition, there are a number of Datanodes, one per node in the cluster, which manage storage attached to the nodes that they run on. The Namenode makes filesystem namespace operations like opening, closing, renaming etc. of files and directories available via an RPC interface. It also determines the mapping of blocks to Datanodes. The Datanodes are responsible for serving read and write requests from filesystem clients; they also perform block creation, deletion, and replication upon instruction from the Namenode [4].

All jobs for testing and development are run on the Hadoop01 cluster at the Cornell Center for Advanced Computing (CAC). Hadoop01 consists of 6 nodes, each of which is a DELL PowerEdge 2950, consisting of 750GB Hard Drive, 16GB RAM and two Quad Core processors and are running Red Hat Enterprise Linux 4.0. The version of Hadoop used for this work is 0.15.3, which runs in Java Runtime Environment (JRE) 1.5.0_13. Finally, all jobs were migrated to the new cluster WL01, which has a Hadoop version of 0.17.1.

4. Process Flow

The following section provides a comprehensive description of all the steps that are involved in the process of Web Graph generation, the integration of Web Graph Generation results with Page Rank program and the combination of the output of Web Graph Generation and Page Rank results to achieve the final output in the desired format.

4.1 Web Graph Generation

The algorithm of Web Graph Generation is implemented by dividing the complete process into three significant java programs consisting of a total of four Map/Reduce jobs, which as described in detail as follows:

4.1.1 StepOne

The input for this step is an ASCII encoded tab-separated-value (TSV) file, where each line is a hyperlink tuple, expressed in the form as shown below:
fromURL < TAB > toURL
Taking the input specified above, StepOne performs the following two main tasks:
1) Canonicalize each URL in the pair i.e. both the fromURL as well as to URL
2) Create a list of all nodes of the graph with the set of all edges to each, i.e., all (*u*,*v*) pairs such that v (*toURL)* that do not identify a node (*fromURL*) of the graph are removed. However, the duplicates links are retained, but are removed in Step 2.

Map

The mapper of StepOne first canonicalizes each pair of the URLs that are passed through the input file. These canonicalized *fromURL* and *toURL* are then passed to the mapper output collector in the form of *(toURL, fromURL)* and *(fromURL, Marker)*, where the *Marker* is a dummy value to indicate that a URL has appeared as a *fromURL* (to indicate that the page identified by the respective *fromURL* has been crawled and is in the data set of interest). The string '1' is used as the marker in this application. Conversely, if a page identified by a particular URL does not have a marker in the valuelist corresponding to its URL as the key, this page does not appear as a *fromURL* (i.e., it is not in the data set, it has not been crawled, or it is a "dangling" node) and thus should be excluded from the web graph.

Combiner

The combiner takes the output from the mapper, and groups the *(key, value)* pairs together on the basis of the key. It then examine whether the value is equal to '1' or not. If it is, then it sends it to combiner output collector as *(key, one).* If the value is not '1' then it keeps on appending it to an arraylist variable named *valuesList* (which comprises of a list of URLs) for a key (which is a URL) and pass this *(key, valuesList)* to the combiner output collector.

Reduce

The reducer takes the output from the mapper, and groups the *(key, value)* pairs together on the basis of the key. We examine whether a marker exists in the grouped *valuesList* (which comprises a list of URLs) for a key (which is a URL) and pass the *(key, valuesList)* to the reducer

output collector if a marker exists, otherwise, this *(key, valuesList)* is not collected because the page represented by the key URL is not a valid node. Since the Hadoop library does not specify an order in which the items for a particular key are returned, the dummy marker may appear at any time. Thus, each URL seen is stored into an arraylist and then emitted once when the marker had been seen. If the marker was not seen, then the data in the arraylist could be thrown away by requesting garbage collector.

4.1.2 StepTwo

The output of the above StepOne forms the input for this step i.e. this step receives the input in the following format:
toURL <TAB> {Set of any number of tab separated fromURL}

Map

The mapper reads the input, one line at a time, and generates a 64 bit FNV hashcode for every toURL it receives. FNV hashes are designed to be fast while maintaining a low collision rate. The high dispersion of the **FNV** hashes makes them well suited for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc. The implementation of FNV hashcode logic is as follows:

```
public  static final long FNV1_64_INIT = 0xcbf29ce484222325L;
private static final long FNV_64_PRIME = 0x100000001b3L;
public long hashCode64(String str)
{
        long hval = FNV1_64_INIT;
        int len = str.length();
        for (int i=0; i<len; i++)
        {
                hval *= FNV_64_PRIME;
                hval ^= (long)str.charAt(i);
        }
        return hval;
}
```

The Map step then outputs *(toURL, $FNV_Hashcode of toURL)* tuple and *(fromURL, FNV_Hashcode of toURL)* tuples. Here, the $ symbol acts as a marker which is used to determine the *toURL* in the following reduce step.

Reduce

The reduce step gets all the values for a particular URL and iterates through them to find the marker '$' inserted in the above Map job. If the marker is found, it means that this particular

URL has appeared both as a *toURL* and a *fromURL* and should be a node in the final web graph. The reduce class only collects the output if this marker is found, otherwise that particular data set is ignored. All the data is initially collected in a TreeSet data structure and then outputted in the following format: *(FNV_Hashcode of fromURL, fromURL, outDegree, tab-separated FNV_Hashcode values of all the toURLs)* where *outDegree* is the number of *toURLs* that come out of the fromURL in the tuple. The marker '$' is ignored as it is just used for identification purposes.

4.1.3 StepThree

The output of the above StepTwo forms the input for this step i.e. this step receives the input in the following format:
FNV_Hashcode of fromURL <TAB> fromURL <TAB> outDegree <TAB> {Set of any number of tab separated FNV_Hashcode values of all the toURLs}

This step consists of two separate Map/Reduce jobs which are completed one after the other and both use the same input file with the format as described above.

MapA

The mapper reads the input, one line at a time, and parses out the first two values i.e. FNV_Hashcode of fromURL and fromURL and then passes these values to the mapper output collector in the form of *(FNV_Hashcode of fromURL, fromURL)*. It also increments a counter for every line that it reads from the input file using the reporter class as shown below:

```
// Increment counter
reporter.incrCounter(counterValues.noOutputRecords, 1);
```

Where counterValues is a static enumeration type declared at the starting of this class. The function 'incrCounter' is described below:

> *void incrCounter (Enum key, long amount)*
> *Increments the counter identified by the key, which can be of any Enum type, by the specified amount.*
> *Parameters:*
> *key - key to identify the counter to be incremented. The key can be any Enum.*
> *amount - A non-negative amount by which the counter is to be incremented.*

This counter value is required in second Map/Reduce phase of this class.

ReduceA

The reduce step simply collects all the values received from the above Map class in the same format and writes them to a single output file in a sorted manner. The number of output files depend on the total number of reduce tasks that are running and can be set to 1 through the JobConf class.

MapB

The mapper reads the input, one line at a time, and parses out the first value i.e. FNV_Hashcode of fromURL, which acts as the key in the mapper output collector and all the rest of the values, only leaving aside fromURL, and concatenate them together, which acts as the value in mapper output collector. Thus, it passes the input in the form of (*FNV_Hashcode of fromURL, {outDegree, tab-separated FNV_Hashcode values of all the toURLs})* to the mapper output collector.

ReduceB

The reduce step collects all the values received from the above Map class and configures the total number of records value that was set in MapA step and saves it into a floating point variable, as shown below:

```
private float outputCount = 0f;
public void configure(JobConf job)
 {
        outputCount = job.getFloat("outputCount", 1.0f);
}
```

It then evaluates the value of (1/outputCount) and appends it to the front of the value received from the above Map job. Then, the key is set as received from the Map and the new string is set as value and sent over to reducer output collector, which writes them to a single output file in a sorted manner. The number of output files depend on the total number of reduce tasks that are running and can be set to 1 through the JobConf class.

The below figure diagrammatically provides the process flow for the Web Graph Generation Algorithm, depicting the functionality of each Step using block diagrams:

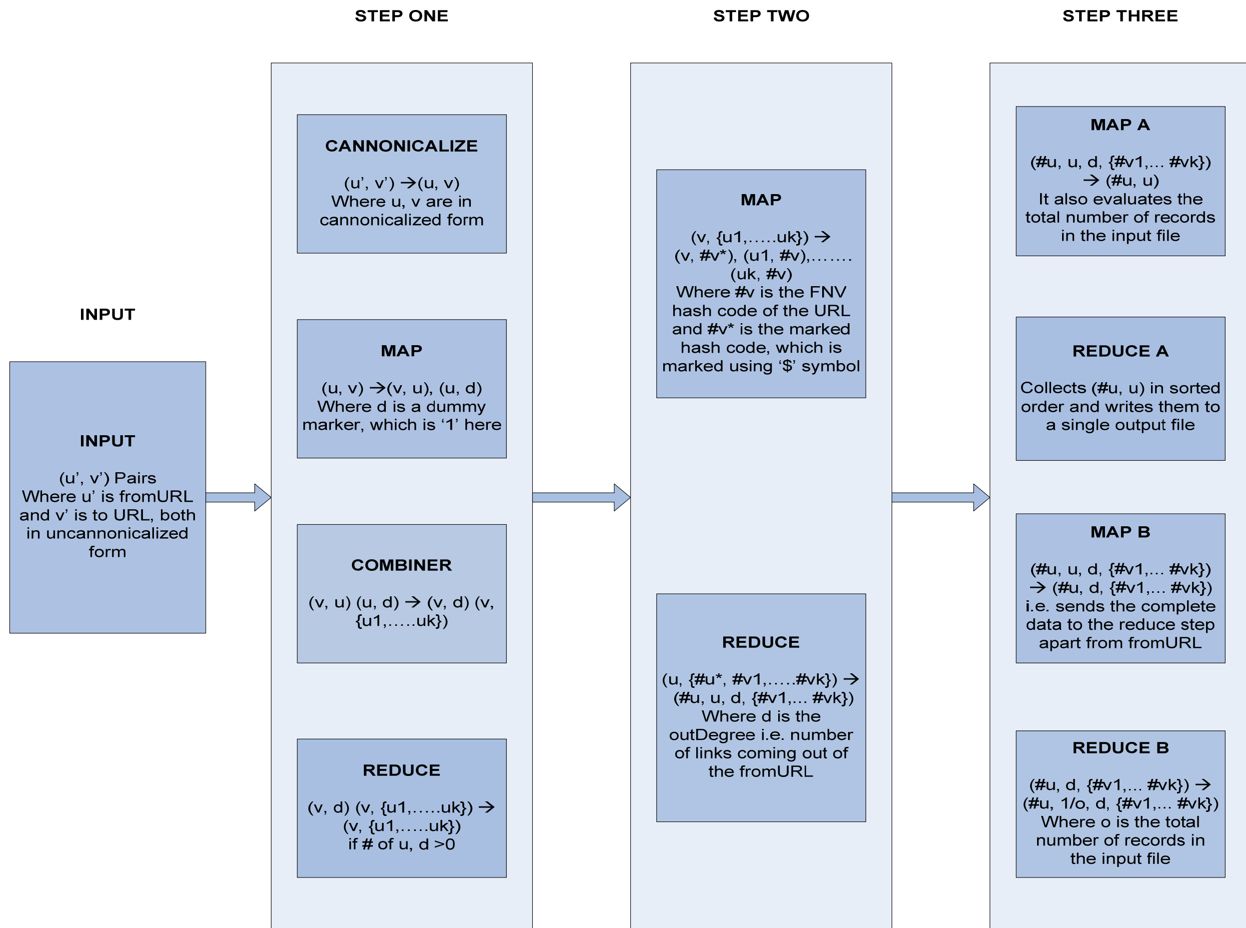

Figure: Process Flow for the Web Graph Generation Algorithm

4.2 Integration of Web Graph Generation with Page Rank Calculation

The output file from ReduceB of StepThree above is used as the input to the PageRank program. It is in the specified format as required by the PageRank algorithm to function correctly. This input filed is copied over to the HDFS directory 'pagerankData-1' and the following command is executed on the command prompt from the same folder where pagerank.jar is kept:

hadoop jar pagerank.jar [-e <#damping factor> : default 0.85] [-m <#map tasks> : default 20] [-r <#reduce tasks> : default 10] [-i <#iterations> : default 60] [-c <true / false> : use combiner or not default true>] [-pr :output pageId and pagerank only by default output contains pageID PageRank #outlinks outlinksIDs] [-input <input directory> : default pagerankData-1] [-output <output directory> : default pagerankData-#of iterations] [-default : uses the default values mentioned above if default is not used program lets hadoop choose the number of map and reduce tasks]

If none of the above parameters are provided while running the PageRank job, then this will by default take the input from pagerankData-1 and do 60 iterations with a damping factor of 0.85 and the final output will be stored in HDFS directory pagerankData-61.

4.3 Combination of Web Graph Generation and Page Rank Calculation Results

The output produced by ReduceA of StepThree of Web Graph Generation and the output of PageRank, after completing all the specified iterations, are combined together to produce the final output in the specified format. The algorithm of this job is designed keeping in mind that both the input files to the Map job are not in the same format but have 'FNV_Hashcode of fromURL' as the first value in every input line.

4.3.1 StepOne

The input for this step is output produced by ReduceA of StepThree of Web Graph Generation and the output of PageRank, after completing all the specified iterations, as shown below:
FNV_Hashcode of fromURL <TAB> fromURL
FNV_Hashcode of fromURL <TAB> PageRank <TAB> outDegree <TAB> {Set of any number of tab separated FNV_Hashcode values of all the toURLs}

Map

The mapper reads the input, one line at a time, and parses out the first two values i.e. FNV_Hashcode of fromURL and fromURL / PageRank and then passes these values to the mapper output collector in the form of *(FNV_Hashcode of fromURL, fromURL)* or *(FNV_Hashcode of fromURL, PageRank).*

Reduce

The reducer takes the output from the mapper, and groups the *(key, value)* pairs together on the basis of the key. It then iterates through the values and examines whether the first character of the value is 'h' or not, to determine if the value is fromURL or the PageRank. It then appends them together along with a tab separation. This value is then sent to the reducer output collector and the final output is in the format of (*FNV_Hashcode of fromURL, PageRank, fromURL).*

The below figure diagrammatically provides the process flow for the Combination of Web Graph Generation and Page Rank Calculation Results, depicting the functionality of each Step using block diagrams:

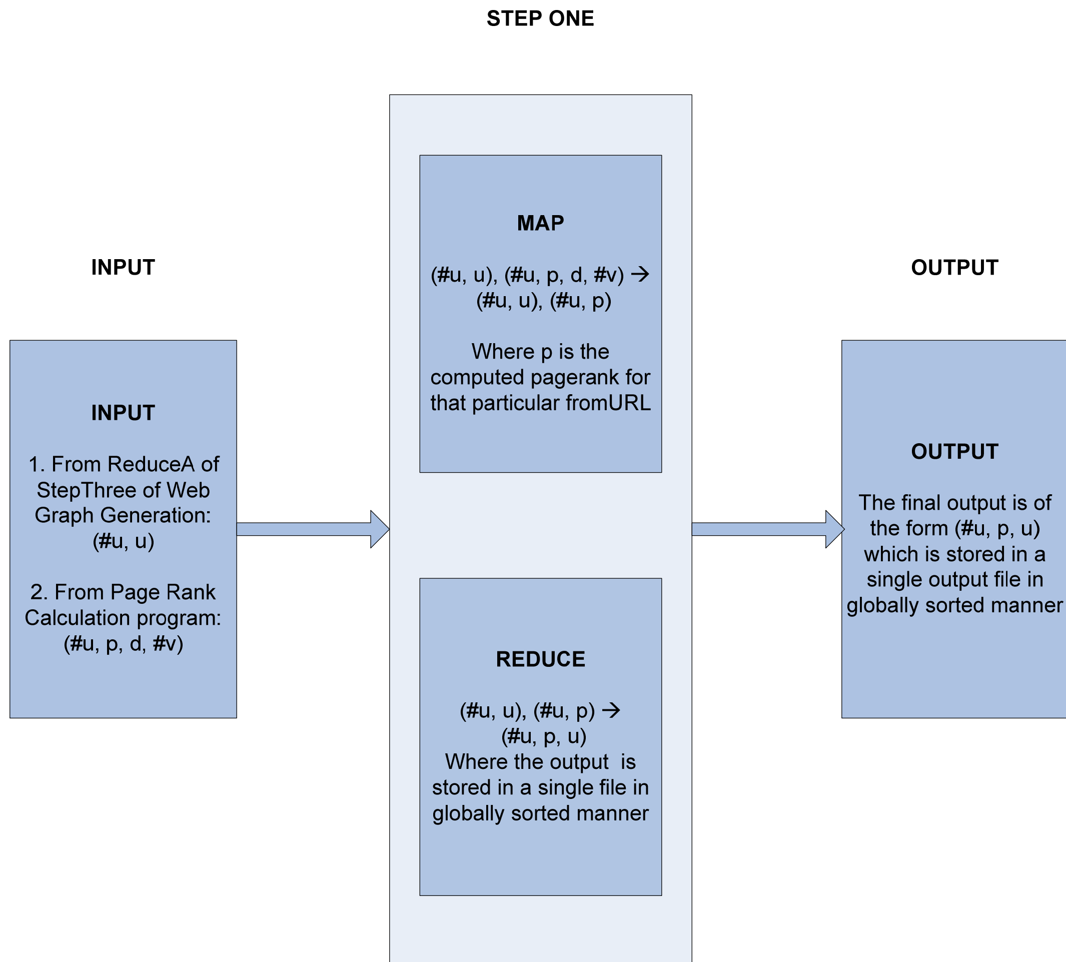

Figure: Process Flow for the Combination of Web Graph Generation and Page Rank Calculation Results

5. Issues Faced / Solutions

The following section provides the information on the major technical issues that were faced in the implementation of the designed algorithm and their respective solutions:

5.1 Lack of Memory Space for JVM

Some of the jobs that dealt with huge data sets like that of Amazon Links would run out of memory space while performing intensive in-memory computations and throw a "java.lang.OutOfMemoryError: Java heap space at java.lang.String". This was solved by reserving an appropriate amount of memory at the time of JVM creation using the -Xmx

parameter. This was set to maximum possible value in the user defined hadoop-site.xml configuration file as shown below:

```
<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx3072m</value>
  <description>Java opts for the task tracker child processes. Subsumes 'mapred.child.heap.size' (If a mapred.child.heap.size value is found in a configuration, its maximum heap size will be used and a warning emitted that heap.size has been deprecated). Also, the following symbol, if present, will be interpolated: @taskid@ is replaced by current TaskID. Any other occurrences of '@' will go unchanged. For example, to enable verbose gc logging to a file named for the taskid in /tmp and to set the heap maximum to be a gigabyte, pass a 'value' of: -Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc
  </description>
</property>
```

5.2 Lack of Replication in the cluster

Initially, all the input files were copied over only to the hadoop01 node, thus providing no replication of input data and poor performance because of lack of effective utilization of DFS on all the five nodes. This was solved by appropriately setting the number of machines a single file should be replicated to before it becomes available. This was set to value of 3 in the hadoop-default.xml configuration file as shown below:

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
  <description>Default block replication.
  The actual number of replications can be specified when the file is created.  The default   is used if replication is not specified in create time.
  </description>
</property>
```

Here, the dfs.replication variable specifies the default block replication. It defines how many machines a single file should be replicated to before it becomes available. If this value is set higher than the number of slave nodes (more precisely, the number of Datanodes) that are available, then there will be a lot of “Zero targets found, forbidden1.size=1” type errors in the log files.

5.3 Integration problems with Page Rank Calculation

The integration of Web Graph Generation output with the Page Rank Calculation program initially resulted in "*java.lang.NullPointer Exception*" because certain type of datasets were ignored while generating the web graph. In the Reduce class of StepTwo of Web Graph Generation, the following computation of taking place after it was determined that that particular URL is a valid node in the web graph:

```
if(foundMarker)
{
    Iterator iterator = treeSet.iterator();
    String finalOutput = "";
    int count = 0;
    if(iterator.hasNext())
    {
        finalOutput = iterator.next().toString();
        count = 1;

        while(iterator.hasNext())
        {
            count++;
            finalOutput += '\t' + iterator.next().toString();
        }
      output.collect(marker, new Text(key.toString()+"   "+count+"   "+finalOutput));
    }
}
```

Since the output.collect statement is inside the 'if' loop, it only evaluates if there are some outlinks from that particular URL. If there are no outlinks i.e. outDegree is zero, then that particular node was simply ignored, thus adversely impacting the Page Rank algorithm. This was rectified by moving the output.collect statement outside of the 'if' statement, thus creating records with output format "*FNV_Hashcode of fromURL <TAB> fromURL <TAB> 0*", as the value of count here will be zero.

6. Experimental Results

This section discusses the experimental results obtained by running multiple jobs with varying parameters over the actual input data obtained through the web crawls of different domains, which is used to establish a relationship between the delivered performance and the required

computational power to determine the overall scalability of the algorithm. The performance is measured in terms of total elapsed time obtained by varying different parameters like the size of input file, number of map tasks, number of reduce tasks, etc. StepOne of the Web Graph Generation algorithm is the most computationally intensive step of the entire process as it deals with the maximum amount of input data and performs heavy in-memory computations. Thus, it forms the bottleneck of the entire application. All the other steps are completed in a significantly lower time frame as compared to StepOne.

To establish a correlation between the time needed to complete a particular job and the size of input file, the size of input file is sequentially increased without changing any other parameter. The results obtained through this experiment are tabulated below:

SIZE OF INPUT DATA ***(in GB)***	**TIME** ***(Seconds)***	**OUTPUT** ***(toURL, fromURL) Tuples***
0.5 GB *Cornell Links*	149	351093
4.52 GB *Universities Links*	828	2367416

Figure: Table showing the comparison between 0.50 GB and 4.52 GB data sets

This shows that the increase in the size of input data is directly proportional to the increase in the total time required to finish that job as well as to the total number of output records.

To establish a correlation between the time needed to complete a particular job and the number of map and reduce tasks, the *JobConf* parameters *setNumMapTasks()* and *setNumReduceTasks()* are used to vary the values sequentially. The results obtained through this experiment are display in the below graph:

MAP TASKS	REDUCE TASKS	TIME	RATIO
1	1	106	1
2	1	109	2
3	1	111	3
4	1	120	4
5	1	112	5
6	1	108	6
7	1	110	7
8	1	104	8
9	1	100	9
10	1	97	10

Figure: Table showing the data collected for generating the below graph

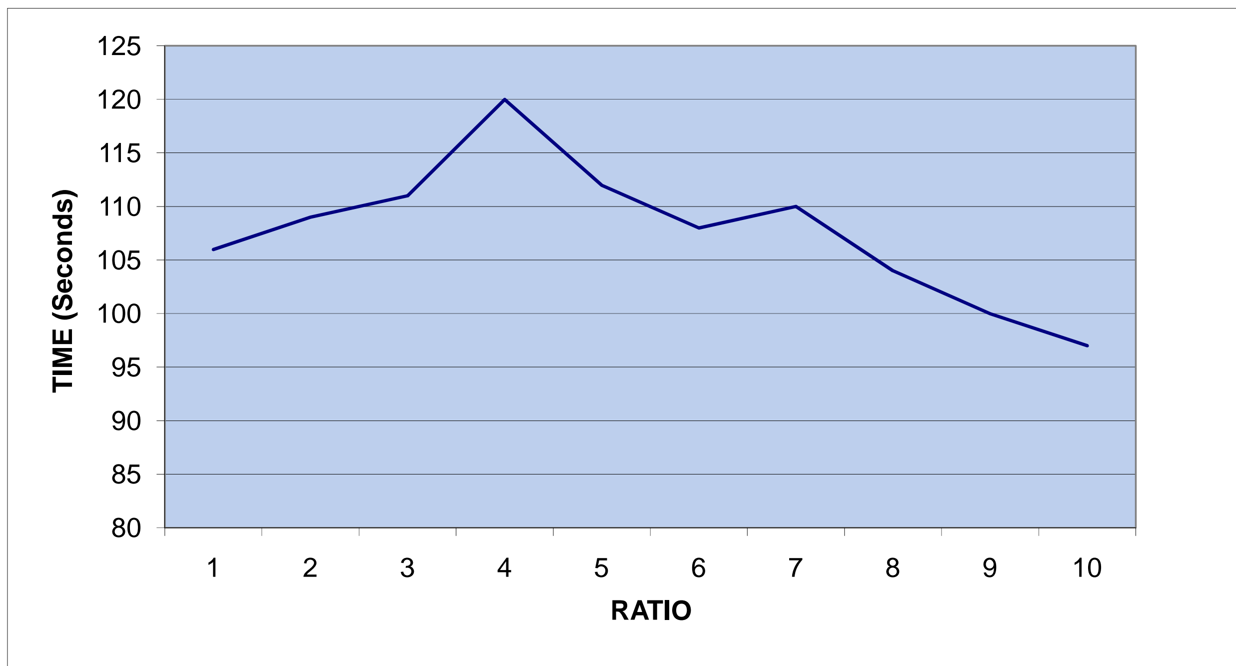

Figure: Graph depicting the correlation between the time needed to complete a particular job and the ratio of the number of map and reduce tasks

The above graph depicts a relationship between the time needed to complete a particular job and the ratio of the number of map and reduce tasks that are performed by it.

7. Code Compilation and Running

7.1 Web Graph Generation

The below section provides step by step information on how to compile and build each program for Web Graph Generation and run them using a common shell script file. All the programs for web graph generation are written in Java language and complied into classes which are aggregated into a single JAR file for each particular Step of the process. The compilation of various steps is very similar with the only major difference being that the Canonicalizer is used only for Step 1. In the Step 3 of the process, the mapred.reduce.tasks parameter must be given a value of 1 so that the output appears in only one sorted file.

Compilation

StepOne

1) mkdir StepOne_classes

2) javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar –d StepOne_classes Canonicalizer.java StepOne.java
3) jar -cvf StepOne.jar -C StepOne_classes/ .

StepTwo
1) mkdir StepTwo_classes
2) javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar –d StepTwo_classes StepTwo.java
3) jar -cvf StepTwo.jar -C StepTwo_classes/ .

StepThree
1) mkdir StepThree_classes
2) javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar –d StepThree_classes StepThree.java
3) jar -cvf StepThree.jar -C StepThree_classes/ .

Running

The following command is used to run the job on the cluster:
sh webGraphJob.sh {INPUT_DIR} {OUTPUT_DIR} {CONFIGURATION_DIR}

Here, the shell script file 'webGraphJob.sh' should be present in the same folder which contains all the three JAR files, INPUT_DIR is the path of the input folder on the HDFS (Hadoop Distributed File System) where the original data files in the form of "fromURL(Tab)toURL" are present, OUTPUT_DIR is the path of the folder on the HDFS (Hadoop Distributed File System)where the output folders created by this job will be stored and CONFIGURATION_DIR is the path on the local machine which contains the user-defined XML configuration files specific to that particular job. The INPUT_DIR and OUTPUT_DIR arguments are mandatory for the job to start successfully while the CONFIGURATION_DIR path is required only when the user wants to change some parameters in accordance with the current job while the other two arguments are mandatory.

7.2 Integration of Web Graph Generation with Page Rank Calculation

Running

The output file from ReduceB of StepThree above is used as the input to the PageRank program. It is in the specified format as required by the PageRank algorithm to function correctly. This input filed is copied over to the HDFS directory 'pagerankData-1' and the following command is executed on the command prompt from the same folder where pagerank.jar is kept:

hadoop jar pagerank.jar [-e <#damping factor> : default 0.85] [-m <#map tasks> : default 20] [-r <#reduce tasks> : default 10] [-i <#iterations> : default 60] [-c <true / false> : use combiner or not default true>] [-pr :output pageId and pagerank only by default output contains pageID PageRank #outlinks outlinksIDs] [-input <input directory> : default pagerankData-1] [-output <output directory> : default pagerankData-#of iterations] [-default : uses the default values mentioned above if default is not used program lets hadoop choose the number of map and reduce tasks]

If none of the above parameters are provided while running the PageRank job, then this will by default take the input from pagerankData-1 and do 60 iterations with a damping factor of 0.85 and the final output will be stored in HDFS directory pagerankData-61.

7.3 Combination of Web Graph Generation and Page Rank Calculation Results

The below section provides information on how to compile and build the program for obtaining the Web Graph / Page Rank Integrated Result and run it through the command prompt. This application is entirely written in Java language and complied into classes which are aggregated into a single JAR file. For this application, the mapred.reduce.tasks parameter must be given a value of 1 so that the output appears in only one sorted file.

Compilation

StepOne

1) mkdir WebgraphPagerankCombined_classes
2) javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar –d WebgraphPagerankCombined_classes StepOne.java
3) jar -cvf WebgraphPagerankCombined.jar -C WebgraphPagerankCombined_classes/ .

Running

The following command is used to run the job on the cluster:
hadoop jar WebgraphPagerankCombined.jar StepOne {INPUT_DIR} {OUTPUT_DIR}

Here, the jar file 'WebgraphPagerankCombined.jar' should be present in the same folder from which the above job is started, INPUT_DIR is the path of the folder on the HDFS (Hadoop Distributed File System) where the output files of ReduceA of StepThree of WebGraphGeneration and PageRank programs are present and OUTPUT_DIR is the path of the folder on the HDFS (Hadoop Distributed File System), where the output files created by this application will be stored.

7.4 Migration to the new cluster with Hadoop version 0.17.1

For compiling the code on the new cluster and running all of the above mentioned commands, use "/usr/local/hadoop/bin/hadoop" instead of "hadoop" and replace HADOOP_VERSION by 0.17.1 instead of 0.15.3.

8. Conclusion

The goals identified for the project at the starting of this semester i.e. to improve the algorithm for Web Graph Generation, integrate the Web Graph Generation application with Page Rank Calculation program, combine the output of Web Graph Generation and Page Rank Calculation programs to obtain the final output in the desired format and migrate the entire application to the new cluster setup with a higher Hadoop and JRE version of 0.17.1, were successfully implemented and thoroughly tested along with a detailed analysis of the improvement in performance with experimental results.

Since the new cluster was setup almost at the end of the course of the project, the future teams working on Web Graph Generation can better utilize the advanced computational features of the new version of Hadoop and higher capabilities of the new machines that comprise this cluster.

9. Acknowledgement

I would like extend my thanks to my M.Eng Project Advisor, Prof. William Arms, for his constant guidance and support throughout the course of this project. I would also like to thank Lucy Walle from the Center for Advanced Computing for sharing her expertise on hadoop framework which helped in solving various issues and Manuel Calimlim for providing the input data consisting of webcrawls from various domains which was required for the project.

The Cornell Web Lab is funded in part by the National Science Foundation grants CNS-0403340, DUE-0127308, SES-0537606, and IIS 0634677.

10. References

[1] http://weblab.infosci.cornell.edu/

[2] Anthony Jawad and Jie Teng, "Web Graph Generation: Fall 2007 Report" in Cornell University, December 2007.

[3] http://hadoop.apache.org/core/

[4] http://wiki.apache.org/hadoop/ProjectDescription

[5] Sangwoo Kim, Sanjay Rajan, and Sean Seguin, "Web Graph Generation" in Cornell University, May 2007.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters" in Usenix SDI, 2004.

[7] H. G. Sanjay Ghemawat and S.-T. Leung, "The google file system" in 19th ACM Symposium on Operating Systems Principles, October 2003.

Appendix 1: Source Code

Web Graph Generation

StepOne

```
import java.io.*;
import java.util.*;
import java.lang.Math;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class StepOne {
	public static class MapClass extends MapReduceBase implements Mapper {
		private Text fromURL = new Text();
		private Text toURL = new Text();
		private final static Text one = new Text("1");

		public void map(WritableComparable key, Writable value, OutputCollector output,
		Reporter reporter) throws IOException {
			String splitString[] = ((Text)value).toString().split("\t");

			Canonicalizer can = new Canonicalizer();
			String canFromURL = can.canonicalize(splitString[0].trim());
			String canToURL = can.canonicalize(splitString[1].trim());

			if (canFromURL.length()>0 && canToURL.length()>0)
			{
				fromURL.set(canFromURL);
				toURL.set(canToURL);

				output.collect(toURL, fromURL);
				output.collect(fromURL, one);
			}
		}
	}


	public static class CombinerClass extends MapReduceBase implements Reducer
```

```
{
    private final static Text one = new Text("1");

    public void reduce(WritableComparable key, Iterator values, OutputCollector output,
    Reporter reporter) throws IOException
    {
        boolean fromURLSeen = false;
        ArrayList valuesList = new ArrayList();

        while(values.hasNext())
        {
            String s = ((Text)values.next()).toString().trim();
            if(s.equals("1"))
            {
                output.collect(key, one);
            }
            else
            {
                valuesList.add(s);
                fromURLSeen = true;
            }
        }

        if(fromURLSeen)
        {
            StringBuffer resultValue = new StringBuffer();
            for (int i = 0; i < valuesList.size(); i++)
            {
                resultValue.append(valuesList.get(i).toString());
                resultValue.append("\t");
            }
            output.collect(key, new Text(resultValue.toString()));
        }
    }
}
public static class ReduceClass extends MapReduceBase implements Reducer
{
    public void reduce(WritableComparable key, Iterator values, OutputCollector output,
    Reporter reporter) throws IOException
    {
        boolean oneSeen = false;
        boolean fromURLSeen = false;
        ArrayList valuesList = new ArrayList();
```

```
			while(values.hasNext())
			{
				String s = ((Text)values.next()).toString().trim();
				if(s.equals("1"))
				{
					oneSeen = true;
				}
				else
				{
					valuesList.add(s);
					fromURLSeen = true;
				}
			}

			if(oneSeen && fromURLSeen)
			{
				StringBuffer resultValue = new StringBuffer();
				for (int i = 0; i < valuesList.size(); i++)
				{
					resultValue.append(valuesList.get(i).toString());
					resultValue.append("\t");
				}
				output.collect(key, new Text(resultValue.toString()));
			}
		}
	}
	public static void main(String[] args) throws IOException {
		JobConf conf = new JobConf(StepOne.class);
		conf.setJobName("StepOne");
		conf.setOutputKeyClass(Text.class);
		conf.setOutputValueClass(Text.class);
		conf.setMapperClass(MapClass.class);
		conf.setCombinerClass(CombinerClass.class);
		conf.setReducerClass(ReduceClass.class);
		conf.setInputPath(new Path(args[0].trim()));
		conf.setOutputPath(new Path(args[1].trim()));
		JobClient.runJob(conf);
	}
}
```

StepTwo

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;

public class StepTwo {
	public static class MapClass extends MapReduceBase implements Mapper {
		private Text index = new Text();
		private Text toUrl = new Text();
		private Text fromUrl = new Text();

		public  static final long FNV1_64_INIT = 0xcbf29ce484222325L;
		private static final long FNV_64_PRIME = 0x100000001b3L;

		public long hashCode64(String str)
		{
			long hval = FNV1_64_INIT;
			int len = str.length();
			for (int i=0; i<len; i++)
			{
				hval *= FNV_64_PRIME;
				hval ^= (long)str.charAt(i);
			}

			return hval;
		}

		public void map(WritableComparable key, Writable value, OutputCollector output,
		Reporter reporter) throws IOException {
			String indexedUrlToFrom = value.toString();
			String[] splitString = indexedUrlToFrom.split("\t");
```

```
                index.set("$" + hashCode64(splitString[0].trim()));
                toUrl.set(splitString[0].trim());

                output.collect(toUrl, index);

                index.set(hashCode64(splitString[0].trim())+"");

                for(int i = 1; i < splitString.length; i++) {
                        fromUrl.set(splitString[i].trim());
                        output.collect(fromUrl, index);
                }
        }
}

public static class Reduce extends MapReduceBase implements Reducer {
        public void reduce(WritableComparable key, Iterator values, OutputCollector output,
        Reporter reporter) throws IOException {
                TreeSet<Long> treeSet = new TreeSet<Long>();
                Text marker = null;
                boolean foundMarker = false;
                String currentPIN = null;

                try
                {
                        while (values.hasNext())
                        {
                                currentPIN = values.next().toString();

                                if(currentPIN.charAt(0) == '$')
                                {
                                        marker = new Text(currentPIN.substring(1));
                                        foundMarker = true;
                                }
                                else
                                {
                                        treeSet.add(new Long(currentPIN));
                                }
                        }

                        if(foundMarker)
                        {
                                Iterator iterator = treeSet.iterator();
```

```
                    String finalOutput = "";
                    int count = 0;
                    if(iterator.hasNext())
                    {
                        finalOutput = iterator.next().toString();
                        count = 1;

                        while(iterator.hasNext())
                        {
                            count++;
                            finalOutput += '\t' + iterator.next().toString();
                        }
                    }

                output.collect(marker, new Text(key.toString()+"   "+count+"
                "+finalOutput));

                }
            }
            catch (Exception e)
            {
                System.out.println(e);
             }
        }
    }

    public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(StepTwo.class);
        conf.setJobName("StepTwo");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);
        conf.setMapperClass(MapClass.class);
        //conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputPath(new Path(args[0].trim()));
        conf.setOutputPath(new Path(args[1].trim()));
        JobClient.runJob(conf);
    }
}
```

StepThree

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.RunningJob;
import org.apache.hadoop.mapred.Counters;
import org.apache.hadoop.mapred.MapReduceBase;

public class StepThree{

	static enum counterValues { noOutputRecords }

	public static class MapClassA extends MapReduceBase implements Mapper
	{
		public void map(WritableComparable key, Writable value, OutputCollector output,
		Reporter reporter) throws IOException
		{
			String[] splitString = value.toString().split("\\s+");
			output.collect(new Text(splitString[0]), new Text(splitString[1]));

			// Increment counter
			reporter.incrCounter(counterValues.noOutputRecords, 1);
		}
	}

	public static class ReduceClassA extends MapReduceBase implements Reducer
	{
		public void reduce(WritableComparable key, Iterator values, OutputCollector output,
		Reporter reporter) throws IOException
		{
			while (values.hasNext())
			output.collect(new Text(key.toString()),new Text(values.next().toString()));
```

```
        }
}

public static class MapClassB extends MapReduceBase implements Mapper
{
        public void map(WritableComparable key, Writable value, OutputCollector output,
        Reporter reporter) throws IOException
        {
                String[] splitString = value.toString().split("\\s+");
                String newString = new String();
                for (int i = 2; i < splitString.length; i++)
                        newString = newString + "   " + splitString[i];
                output.collect(new Text(splitString[0]), new Text(newString));
        }
}

public static class ReduceClassB extends MapReduceBase implements Reducer
{
        private float outputCount = 0f;

        public void configure(JobConf job)
        {
                outputCount = job.getFloat("outputCount", 1.0f);
        }

        public void reduce(WritableComparable key, Iterator values, OutputCollector output,
        Reporter reporter) throws IOException
        {
                String newValue = new String();
                while (values.hasNext())
                {
                        newValue = (1.0/outputCount) + "\t" + values.next().toString();
                        output.collect(new Text(key.toString()),new Text(newValue));
                }
        }
}

public static void main(String[] args) throws IOException
{

        float outputCount;

        //Configuration parameters for JobA
```

```
		JobConf confA = new JobConf(StepThree.class);
		confA.setJobName("StepThreeA");
		confA.setOutputKeyClass(Text.class);
		confA.setOutputValueClass(Text.class);
		confA.setMapperClass(MapClassA.class);
		confA.setNumReduceTasks(1);
		confA.setReducerClass(ReduceClassA.class);
		confA.setInputPath(new Path(args[0].trim()));
		confA.setOutputPath(new Path(args[1].trim()));
		outputCount =
		JobClient.runJob(confA).getCounters().getCounter(counterValues.noOutputRecords);

		//Configuration parameters for JobB
		JobConf confB = new JobConf(StepThree.class);
		confB.setJobName("StepThreeB");
		confB.setOutputKeyClass(Text.class);
		confB.setOutputValueClass(Text.class);
		confB.setMapperClass(MapClassB.class);
		confB.setNumReduceTasks(1);
		confB.setReducerClass(ReduceClassB.class);
		confB.setInputPath(new Path(args[0].trim()));
		confB.setOutputPath(new Path(args[2].trim()));
		confB.set("outputCount",outputCount+"");
		JobClient.runJob(confB);

	}
}
```

WebGraphJob.sh

```
	#!/bin/sh

	if test $# != 2 && test $# != 3
	then
	  echo "Please Pass InputPath, OutputPath and Configuration Path (if any) as Arguments"
	  exit 1
	fi

	inputPath=$1
	outputPath=$2
	confPath=$3

	#Create Output Folder
```

```
hadoop dfs -mkdir $outputPath

if test $# == 3
then
	#Run StepOne
	hadoop --config $confPath jar StepOne.jar StepOne $inputPath
	$outputPath/optStepOne_v1/
	#Run StepTwo
	hadoop --config $confPath jar StepTwo.jar StepTwo $outputPath/optStepOne_v1/
	$outputPath/optStepTwo_v1/
	#Run StepThree
	hadoop --config $confPath jar StepThree.jar StepThree $outputPath/optStepTwo_v1/
	$outputPath/optStepThreeA_v1/ $outputPath/optStepThreeB_v1/
else
	#Run StepOne
	hadoop jar StepOne.jar StepOne $inputPath $outputPath/optStepOne_v1/
	#Run StepTwo
	hadoop jar StepTwo.jar StepTwo $outputPath/optStepOne_v1/
	$outputPath/optStepTwo_v1/
	#Run StepThree
	hadoop jar StepThree.jar StepThree $outputPath/optStepTwo_v1/
	$outputPath/optStepThreeA_v1/ $outputPath/optStepThreeB_v1/
fi
```

Combination of Web Graph Generation and Page Rank Calculation Results

StepOne

```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.RunningJob;
```

```
import org.apache.hadoop.mapred.Counters;
import org.apache.hadoop.mapred.MapReduceBase;

public class StepOne{

    public static class MapClass extends MapReduceBase implements Mapper
    {
        public void map(WritableComparable key, Writable value, OutputCollector output, Reporter
        reporter) throws IOException
        {
            String[] splitString = value.toString().split("\\s+");
            output.collect(new Text(splitString[0]), new Text(splitString[1]));

        }
    }

    public static class ReduceClass extends MapReduceBase implements Reducer
    {
            String[] collectedValues = new String[2];
            String testValue = null;
            String newValue = null;

            public void reduce(WritableComparable key, Iterator values,
            OutputCollector output, Reporter reporter) throws IOException
            {
                while (values.hasNext())
                {
                    testValue = values.next().toString();
                    if(testValue.charAt(0) == 'h')
                    {
                        collectedValues[0] = testValue;
                    }
                    else
                    {
                        collectedValues[1] = testValue;
                    }
                    }
                    newValue = collectedValues[1] + "\t" + collectedValues[0];
                    output.collect(new Text(key.toString()),new Text(newValue));
                    }

            }
```

```
	public static void main(String[] args) throws IOException
	{

		//Configuration parameters for Job
		JobConf conf = new JobConf(StepOne.class);
		conf.setJobName("StepOne");
		conf.setOutputKeyClass(Text.class);
		conf.setOutputValueClass(Text.class);
		conf.setMapperClass(MapClass.class);
		conf.setNumReduceTasks(1);
		conf.setReducerClass(ReduceClass.class);
		conf.setInputPath(new Path(args[0].trim()));
		conf.setOutputPath(new Path(args[1].trim()));
		JobClient.runJob(conf);

	}
}
```