

1. INTRODUCTION

There is a lot of buzz around “big data” and rightly so. Organizations that are capturing and analysing large amounts of data in real time or near–real time are creating significant competitive advantages for themselves, their customers and business partners. Communications service providers (CSPs) are no exception. CSPs that can ingest and analyse network, location and customer data in real time or near–real time have much to gain. They will be able to quickly introduce new capabilities such as location-based services, intelligent marketing campaigns, next best actions for sales and service, social media insights, network intelligence and fraud detection to significantly increase revenues and reduce costs.

We are living in an information age and there is enormous amount of data that is flowing between systems, internet, telephones, and other media. The data is being collected and stored at unprecedented rates. There is a great challenge not only to store and manage the large volume of data, but also to analyze and extract meaningful information from it. There are several approaches to collecting, storing, processing, and analyzing big data.

1.1 OBJECTIVE

The objective of the system is to implement PageRank Algorithm in Hadoop to rank pages (or websites) to get relevant information for a user's query. Hadoop, an open source software platform managed by Apache Software Foundation is very helpful in storing and managing vast amounts of data in a distributed environment for page ranking. Outgoing links of the page are considered for ranking the pages.

1.2 MOTIVATION

Due to the advent of new technologies, devices, and communication means like social networking sites, the amount of data produced by mankind is growing rapidly every year. From this huge amount of data, user only requires the relevant information. Relevant pages are identified by giving ranks to these pages which can be given by implementing PageRank Algorithm. Relevant pages are important than the quantity of pages. Computing

PageRank has huge storage challenge. Hadoop, an open source software stores and manages such kind of large amount of data in a distributed system.

1.3 OVERVIEW

This system is focused on how to handle large amount of data and how to analyze the data. The technology used for this is Hadoop technology. In this project the data taken is Xml file that contains links to and from various websites. Then the data is stored into HDFS (Hadoop distributed file system) format in the form of clusters. After the storage is done, then the processing of data can be done based on the user requirements. The processing of data can be done using many nodes. Hadoop basically contains many ecosystems which provide different ways of processing or analyzing the data in different environments. The two basic components of Hadoop are hdfs and MapReduce. HDFS is used to store the data and MapReduce is used to process the data. In MapReduce we write code in java to analyze the data in whatever way we want to. The ecosystems in Hadoop are also for processing and analyzing the data. The different ecosystems of Hadoop are Pig, Hive, Chukwa, HBase, ZooKeeper, Sqoop etc. Here Pig, Hive and Sqoop have been implemented. So the first ecosystem implemented is Pig. Pig is scripting language. It can process both structured and unstructured data. In this Pig scripts are written on the data to get results. Then hive is a query language, it can handle only structured data. In this queries are written on to the data to analyze it. Then finally Sqoop, it is actually a support for Hadoop rather than an ecosystem. It is used to transfer data from one data base to other. And after the processing of data the results are displayed.

1.4 PROBLEM STATEMENT

As there is huge amount of data present all over the internet, the user only requires relevant data to his query. Possible method of giving relevant data or pages is by implementing PageRank Algorithm on Hadoop framework, by upload xml file consisting of related pages, parsing the given xml file, calculating the Page Rank for each page, Ordering the pages based on the ranks and displaying the output as page name along with its Rank.

Reported results of PageRank Algorithm using Hadoop can provide a more complete understanding.

1.5 LAYOUT OF THE REPORT

This thesis is organized as follows:

- (i) Chapter I deals with Introduction to the project, objective, motivation, contribution, and overview and problem statement.
- (ii) Chapter II explains about the Literature survey that includes, Big Data, Web Page Ranking and the algorithm, about Weighted Page Ranking, Big Data, Hadoop, Core Components of Hadoop, Hadoop Demos, Eco system of Hadoop.
- (iii) Chapter III discusses about I/O specifications and system requirements.
- (iv) Chapter IV gives the design details of the system like program flow of the system and design (UML) diagrams.
- (v) Chapter V explains the implementation of the system. It discusses in detail how the system is actually built and how the system functions process by process.
- (vi) Chapter VI discusses about the result showing the output screens .
- (vii) Chapter VII discusses about the conclusions and future work.

2. LITERATURE SURVEY

2.1 WEB PAGE RANKING

The Wikipedia has millions of articles and is still growing. Each article in one or the other way has links to other articles. With the help of all the incoming and outgoing links it is possible to determine which page has more importance than others, that is what the Page Ranking does.

2.1.1 What is Page Ranking

To retrieve required information from **WWW**, search engines perform number of tasks based on their respective architecture. When a user makes a query from search engine, it generally returns a large number of pages in response to user queries. This result-list contains many relevant and irrelevant pages according to user's query. As user impose more number of relevant pages in the search result-list. To assist the users to navigate in the result list, various ranking methods are applied on the search results.

It is first The Larry Page came up with an algorithm for page ranking to build a search engine and named it as Google. The search engine uses these ranking methods to sort the results to be displayed to the user. In that way user can find the most important and useful result first. There are a variety of algorithms developed.

2.1.2 Toolbar Page Ranking: PageRank can also be displayed on the toolbar of the web browser, but the toolbar PageRank only goes from 0-10 and looks like a logarithmic scale:

| Toolbar PageRank (log base 10) | Real PageRank |
|---|----------------------|
| 0 | 0 – 100 |
| 1 | 100 – 1,000 |
| 2 | 1,000 – 10,000 |
| 3 | 10,000 – 100,000 |
| 4 | and so on..... |

The exact details cannot be known as the PageRank of all the pages on the web changes every month when Google does its re-indexing. Sometimes the toolbar guesses the PageRank. That is the toolbar PageRank of the pages which are newly uploaded and cannot be indexed.

PageRank says nothing about the content or size of the page, the language it's written in, or the text used in the anchor of a link.

2.1.3 Page Ranking Algorithm

PageRank algorithm relies on the link structure of the web as an indicator of an individual page's value. PageRank is a numeric value that represents how important a page is on the web. When one page has a URL to another page, it is effectively casting a vote for the other page. The in-links to a page indicate the importance of the page. PageRank algorithm calculates a page's importance from the number of in-links it has. The importance of each in-link is also taken into account. The Page Rank imitate on the back link in deciding the rank score. Thus, a page gets hold of a Weighted Page Rank Algorithm Based on Number of Visits of Links of Web Page high rank if the addition of the ranks of its back links is high.

For example, consider four pages A, B, C and D, where D is a non-existing page i.e., this page has not created yet, but has links from C. The links between these pages are as follows

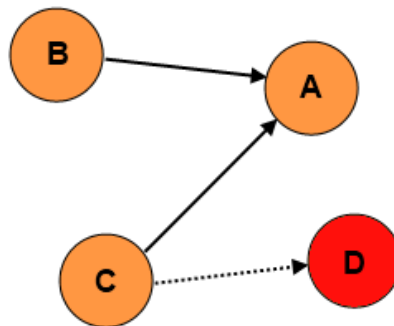


Fig 2.1 Pages with their outgoing links

As shown in the fig 2.1, the pages A,B,C,D contains both outlinks and inlinks. D is viewed in red color as it is the page which doesnot exist.

Rank of A is highest because it gets points from B and C.

PageRank of A= ‘Share’ of the PageRank of the pages linking to A

The formula for calculating the page rank is:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

This formula can be simplified as

$$PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))$$

Note: Page Ranks form a probability distribution over web pages, so the sum of all web pages’ Page Ranks will be one.

Where

PR(Tn) – Each page has a notation of its own self-importance. That’s “PR(T1)” for the first page in the web all the way up to “PR(Tn)” for the last page.

C(Tn) – Each page spreads its vote out evenly almost all of its outgoing links. The count, or number, of outgoing links for page 1 is “C(T1)”, “C(Tn)” for page n, and so on for all pages.

d – All the functions of votes are added together but, to stop the other pages having too much influence, this total vote is “damped down” by multiplying it by 0.85(the factor “d”).

(1-d) – The (1-d) bit at the beginning is a bit of probability math magic so the sum of all web pages’ page rank will be one. It also means that if a page has no links to it even then it gets a small PageRank of 0.15(i.e. 1 – 0.85).

If the formula is applied for the above example:

PageRank of A = $0.15 + 0.85 * (\text{PageRank(B)} / \text{outgoing links(B)} + \text{PageRank(...)}/\text{outgoing link(...)})$

Calculation of A with initial ranking 1.0 per page:

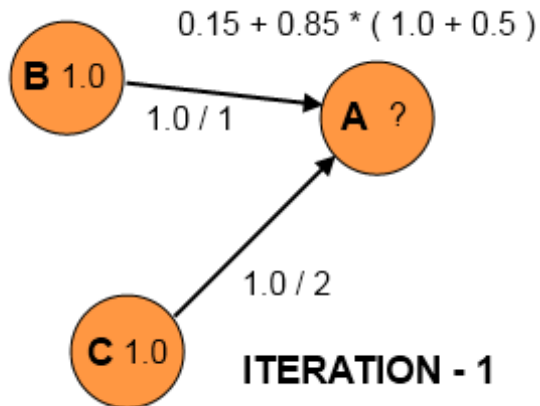


Fig 2.2 Iteration 1

As shown in fig 2.2, page rank for A is calculated with by considering the inlinks from B and C to A.

Using initial rank value of 1.0 for A, B, and C .The output is as follows .Page D is skipped as it not an existing page.

A. 1.425

B. 0.15

C. 0.15

Calculating rank of A from ITERATION-1:

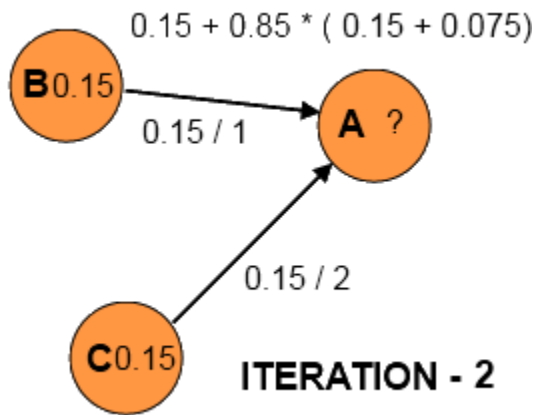


Fig 2.3 Iteration

As shown in fig 2.3, the rank for page A is calculated by considering the rank obtained in first iteration.

By using page ranks as input and calculating the output will be as follows

A.0.34125

B.0.15

C.0.15

As the number of iterations are increasing the rank of page A is decreasing and it will get more accurate. New pages, new links can be added and rank can be calculated. This is one of the tools used by search engines. We are going to implement this with a set of Wikipedia pages.

2.1.4 Weighted PageRank Algorithm:

Weighted page rank(WPR) algorithm. This algorithm is an extension of PageRank algorithm. WPR takes into account the importance of both the inlinks and the outlinks of the pages and distributes rank scores based on the popularity of the pages. WPR performs better than the conventional PageRank algorithm in terms of returning larger number of relevant pages to a given query. The more popular webpages are the more linkages that other webpages tend to have to them or are linked to by them. The Weighted PageRank

Algorithm– assigns larger rank values to more important (popular) pages instead of dividing the rank value of a page evenly among its outlink pages. Each outlink page gets a value proportional to its popularity (its number of inlinks and outlinks). The popularity from the number of inlinks and outlinks is recorded as $W^{in}_{(v,u)}$ and $W^{out}_{(v,u)}$, respectively.

$W^{in}_{(v,u)}$ is the weight of link(u, v) calculated based on number of inlinks of page ‘u’ and the number of inlinks of all reference pages of page ‘v’.

$$W^{in}_{(v,u)} = \frac{I_u}{\sum_{p \in R(v)} I_p}$$

Where I_u and I_p represent the number of in-links of page u and page p respectively.

$W^{out}_{(v,u)}$ is the weight of link(u, v) calculated based on number of out-links of page ‘u’ and the number of out-links of all reference pages of page ‘v’.

$$W^{out}_{(v,u)} = \frac{O_u}{\sum_{p \in R(v)} O_p}$$

Where O_u and O_p represent the number of out -links of page u and page p respectively.

2.2 BIG DATA

Big Data is an all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using on-hand data management tools or traditional data processing applications. The challenges include capture, curation, storage, search, sharing, transfer, analysis and visualization. The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, as compared to separate smaller sets with the same total amount of data, allowing correlations to be found to spot business trends, prevent diseases, combat crime and so on.

Big-Data needs to possess the following properties.

Volume: Enterprises are awash with ever-growing data of all types, easily amassing terabytes—even petabytes—of information.

- Turn 12 terabytes of Tweets created each day into improved product sentiment analysis
- Convert 350 billion annual meter readings to better predict power consumption

Velocity: Sometimes 2 minutes is too late. For time-sensitive processes such as catching fraud, big data must be used as it streams into your enterprise in order to maximize its value.

- Scrutinize 5 million trade events created each day to identify potential fraud
- Analyze 500 million daily call detail records in real-time to predict customer churn faster

Variety: Big data is any type of data - structured and unstructured data such as text, sensor data, audio, video, click streams, log files and more. New insights are found when analyzing these data types together.



Fig: 2.4 Big data

As shown in the fig 2.4, the essential properties of Big Data are volume, variety and velocity

2.3 Distributed computing

2.3.1 What is distributed computing?

Multiple autonomous systems appear as one, interacting via a message passing interface, no single point of failure.

2.3.2 Challenges of Distributed computing

- (i) Resource sharing: Access any data and utilize CPU resources across the system.
- (ii) Openness: Extensions, interoperability, portability.
- (iii) Concurrency: Allows concurrent access, update of shared resources.
- (iv) Scalability: Handle extra load. like increase in users, etc..
- (v) Fault tolerance: by having provisions for redundancy and recovery.
- (vi) Heterogeneity: Different Operating systems, different hardware, Middleware system allows this.
- (vii) Transparency: Should appear as a whole instead of collection of computers.
- (viii) Biggest challenge is to hide the details and complexity of accomplishing above challenges from the user and to have a common unified interface to interact with it. Which is where Hadoop comes in.

2.4 Hadoop

Hadoop is a free, open source Java-based framework that supports the processing of large data sets in a distributed computing environment. It facilitates scalability and takes care of detecting and handling failures. It is part of the Apache project sponsored by the Apache Software Foundation. Hadoop makes it possible to run applications on systems with

thousands of nodes involving thousands of terabytes. Its distributed file system facilitates rapid data transfer rates among nodes and allows the system to continue operating uninterrupted in case of a node failure. This approach lowers the risk of catastrophic system failure, even if a significant number of nodes become inoperative. The Hadoop framework is used by major players including Google, Yahoo and IBM, largely for applications involving search engines and advertising. The preferred operating systems are any flavor of Linux, windows (here we need to use Cygwin) but Hadoop can also work with BSD and OS X.

2.4.1 Hadoop eliminate complexities

Hadoop has components which take care of all complexities for us and by using a simple map reduce framework we are able to harness the power of distributed complexities like fault tolerance, data loss.

It has replication mechanism for data recovery and job scheduling and blacklisting of faulty nodes by a configurable blacklisting policy.

2.5 Hadoop Daemons

Hadoop follows a Master-Slave architecture which is comprised of 5 daemons. A daemon is a back ground service that runs on your Hadoop.

Briefly Hadoop daemons are as follows:

- **Master Daemons**

- (i) Name Node
- (ii) Secondary Name Node
- (iii) Job Tracker

- **Slave Daemons**

(i) Data Node

(ii) Task Tracker

2.5.1 Name Node:

The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself.

Client applications talk to the NameNode whenever they wish to locate a file, or when they want to add/copy/move/delete a file. The NameNode responds the successful requests by returning a list of relevant DataNode servers where the data lives.

The NameNode is a Single Point of Failure for the HDFS Cluster. HDFS is not currently a High Availability system. When the NameNode goes down, the file system goes offline. There is an optional SecondaryNameNode that can be hosted on a separate machine. It only creates checkpoints of the namespace by merging the edits file into the fsimage file and does not provide any real redundancy. Hadoop 0.21+ has a BackupNameNode that is part of a plan to have an HA name service.

2.5.2 Secondary Name Node:

The NameNode stores modifications to the file system as a log appended to a native file system file, edits. When a NameNode starts up, it reads HDFS state from an image file, fsimage, and then applies edits from the edits log file. It then writes new HDFS state to the fsimage and starts normal operation with an empty edits file.

The secondary NameNode merges the fsimage and the edits log files periodically and keeps edits log size within a limit. It is usually run on a different machine than the primary NameNode since its memory requirements are on the same order as the primary

NameNode. The secondary NameNode is started by bin/start-dfs.sh on the nodes specified in conf/masters file.

The secondary NameNode stores the latest checkpoint in a directory which is structured the same way as the primary NameNode's directory. So that the checkpointed image is always ready to be read by the primary NameNode if necessary.

2.5.3 Job Tracker:

The JobTracker is the service within Hadoop that farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack.

- Client applications submit jobs to the Job tracker.
- The JobTracker talks to the NameNode to determine the location of the data
- The JobTracker locates TaskTracker nodes with available slots at or near the data
- The JobTracker submits the work to the chosen TaskTracker nodes.
- The TaskTracker nodes are monitored. If they do not submit heartbeat signals often enough, they are deemed to have failed and the work is scheduled on a different TaskTracker.
- A TaskTracker will notify the JobTracker when a task fails. The JobTracker decides what to do then: it may resubmit the job elsewhere, it may mark that specific record as something to avoid, and it may even blacklist the TaskTracker as unreliable.
- When the work is completed, the JobTracker updates its status.
- Client applications can poll the JobTracker for information.

The JobTracker is a point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted.

2.5.4 Data Node:

A DataNode stores data in the HadoopFileSystem. A functional file system has more than one DataNode, with data replicated across them. On startup, a DataNode connects to the

NameNode; spinning until that service comes up. It then responds to requests from the NameNode for file system operations.

Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data. Similarly, MapReduce operations farmed out to TaskTracker instances near a DataNode, talk directly to the DataNode to access the files. TaskTracker instances can, indeed should, be deployed on the same servers that host DataNode instances, so that MapReduce operations are performed close to the data.

DataNode instances can talk to each other, which is what they do when they are replicating data.

- There is usually no need to use RAID storage for DataNode data, because data is designed to be replicated across multiple servers, rather than multiple disks on the same server.
- An ideal configuration is for a server to have a DataNode, a TaskTracker, and then physical disks one TaskTracker slot per CPU. This will allow every TaskTracker 100% of a CPU, and separate disks to read and write data.
- Avoid using NFS for data storage in production system.

2.5.5 Task Tracker:

A TaskTracker is a node in the cluster that accepts tasks - Map, Reduce and Shuffle operations - from a JobTracker.

Every TaskTracker is configured with a set of *slots*; these indicate the number find somewhere to schedule of tasks that it can accept. When the JobTracker tries to a task within the MapReduce operations, it first looks for an empty slot on the same server that hosts the DataNode containing the data, and if not, it looks for an empty slot on a machine in the same rack.

The TaskTracker spawns a separate JVM processes to do the actual work; this is to ensure that process failure does not take down the task tracker. The TaskTracker monitors these spawned processes, capturing the output and exit codes. When the process finishes, successfully or not, the tracker notifies the JobTracker. The TaskTrackers also send out heartbeat messages to the JobTracker, usually every few minutes, to reassure the JobTracker that it is still alive. These messages also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster work can be delegated.

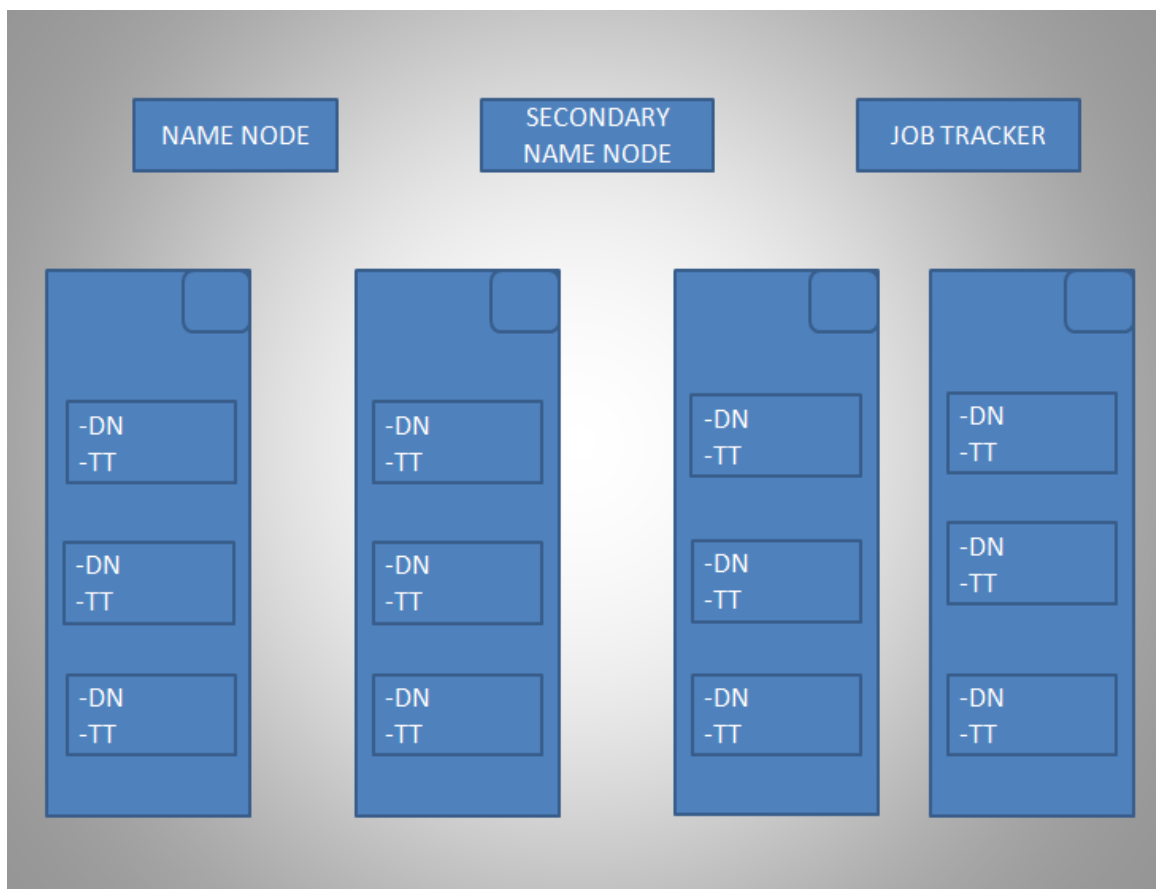


Fig : 2.5 Hadoop Demons

As shown in fig 2.5, the Hadoop contains master Demons which include Namenode, Secondary Namenode, Job Tracker and the slave nodes include Datanode and Task tracker.

2.6 Hadoop Core Components

Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of commodity computers using a simple programming model. It is designed to scale up from single servers to thousands of machines, each providing computation and storage. Rather than rely on hardware to deliver high-availability, the framework itself is designed to detect and handle failures at the application layer, thus delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

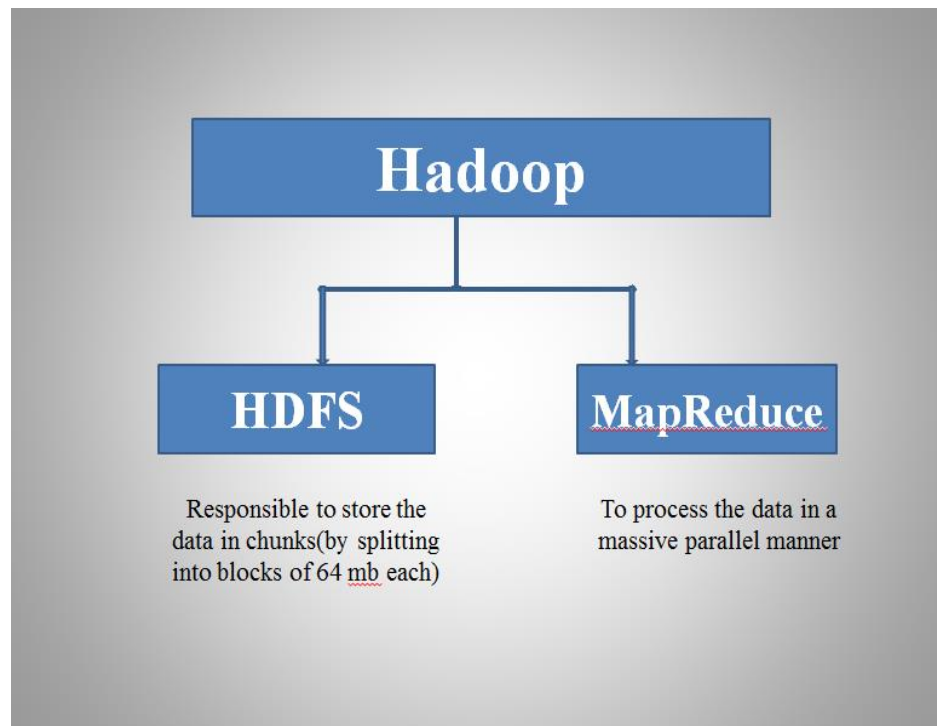


Fig:2.6 Hadoop Core Components

As shown in fig 2.6, Hadoop contains HDFS and MapReduce as two main components.

HDFS (storage) and **MapReduce** (processing) are the two core components of Apache Hadoop. The most important aspect of Hadoop is that both HDFS and MapReduce are designed with each other in mind and each are co-deployed such that there is a single cluster and thus provides the ability to move computation to the data not the other way around. Thus, the storage system is not physically separate from a processing system.

The Hadoop framework transparently provides both reliability and data motion to applications. Hadoop implements a computational paradigm named MapReduce, 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. It enables applications to work with thousands of computation-independent computers and petabytes of data. The entire Apache Hadoop "platform" is now commonly considered to consist of the Hadoop kernel, MapReduce and Hadoop Distributed File System (HDFS), as well as a number of related projects – including Apache Hive, Apache HBase, and others.

Hadoop Features

- (i) Hardware Failure
- (ii) Streaming Data Access
- (iii) Large Data Sets
- (iv) Simple Coherency Model
- (v) Portability Across Heterogeneous Hardware and Software Platforms

2.6.1 The Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large

data sets. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject.

HDFS is a fault tolerant and self-healing distributed file system designed to turn a cluster of industry standard servers into a massively scalable pool of storage. Developed specifically for large-scale data processing workloads where scalability, flexibility and throughput are critical, HDFS accepts data in any format regardless of schema, optimizes for high bandwidth streaming, and scales to proven deployments of 100PB and beyond. Every file by default is split as 64 MB block each by default and could be configured.

2.6.1.1 HDFS Features:

- Scale-Out Architecture - Add servers to increase capacity
- High Availability - Serve mission-critical workflows and applications
- Fault Tolerance - Automatically and seamlessly recover from failures
- Flexible Access – Multiple and open frameworks for serialization and file system mounts
- Load Balancing - Place data intelligently for maximum efficiency and utilization
- Tunable Replication - Multiple copies of each file provide data protection and computational performance
- Security - POSIX-based file permissions for users and groups with optional LDAP integration

By default, every block in Hadoop is 64 MB and is replicated thrice.

The replications of the blocks will be as per the Rack Awareness and by default two replications in one rack and the other in another Rack.

2.6.1.2 File write in HDFS:

Data is written in HDFS as a pipeline write that too block by block. Acknowledgement will be passed to the client indicating that data was successfully written in HDFS. The blocks

are placed as the rack aware fashion. Each block is given a unique ID which will be stored in the metadata.

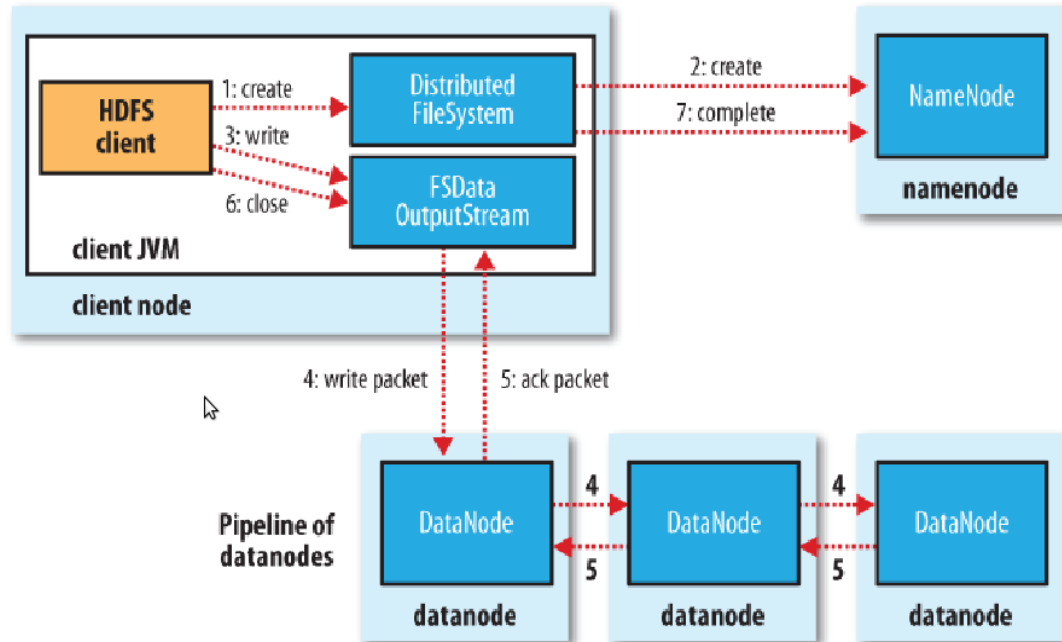


Fig:2.7 File Write in HDFS

As shown in fig 2.7, the HDFS client creates the distributed file system by giving a request to the Namenode. FSdata output stream writes the data to the data node by sending a data write packet to the pipeline of datanodes. If the data is successfully written then an acknowledgement is received from the datanodes.

File reading from HDFS:

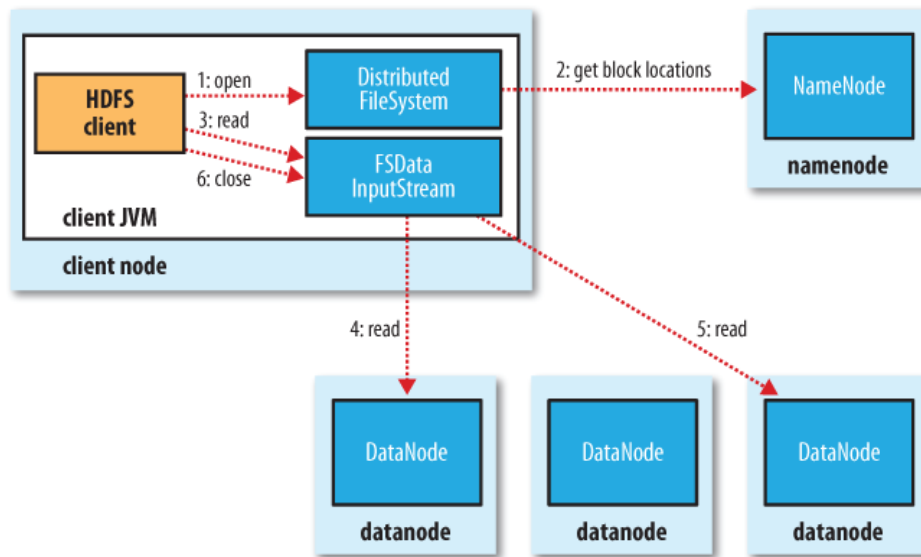


Fig:2.8 File Read from HDFS

As shown in fig 2.8, When client wants to read a file from HDFS, initially it is taken care by the Name Node. Name nodes verify the file matches and asks the nearest data node to handle the file for reading. If file name is not found in the metadata IO exception occurs.

2.6.1.3 Data replications in HDFS:

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

When a client wants to write file to HDFS, with a replication factor of three, it submits the file to cluster, NameNode will come into action, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

2.6.2 MapReduce

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Minimally, applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the job configuration. The Hadoop job client then submits the job and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

2.7 Word count example

Word Count is a simple application that counts the number of occurrences of each word in a given input set.

Driver code:

```
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 WordCount extends Configured implements Tool
{
    public int run(String[] args) throws Exception
    {
        //creating a JobConf object and assigning a job name for identification purpose
        JobConf conf = new JobConf(getConf(), WordCount.class);
        conf.setJobName("WordCount");

        //Setting configuration object with the Data Type of output Key and Value
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        //Providing the mapper and reducer class names
        conf.setMapperClass(WordCountMapper.class);
        conf.setReducerClass(WordCountReducer.class);

        //We wil give 2 arguments at the run time, one in input path and other is output path
        Path inp = new Path(args[0]);
        Path out = new Path(args[1]);

        //the hdfs input and output directory to be fetched from the command line
```

```

FileInputFormat.addInputPath(conf, inp);
FileOutputFormat.setOutputPath(conf, out);
JobClient.runJob(conf);
return 0;
}

```

Word count Mapper

```

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class WordCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>
{
    //hadoop supported data types
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    //map method that performs the tokenizer job and framing the initial key value pairs
    // after all lines are converted into key-value pairs, reducer is called.
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException
    {
        //taking one line at a time from input file and tokenizing the same
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        //iterating through all the words available in that line and forming the key value pair
        while (tokenizer.hasMoreTokens())
        {
            word.set(tokenizer.nextToken());

```



```

        //sending to output collector which inturn passes the same to reducer
        output.collect(word, one);
    }
}

}

public static void main(String[] args) throws Exception
{
    // this main function will call run method defined above.
    int res = ToolRunner.run(new Configuration(), new WordCount(),args);
    System.exit(res);
}
}

```

Word count Reducer

```

import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class WordCountReducer extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable>
{
    //reduce method accepts the Key Value pairs from mappers, do the aggregation based
    on keys and produce the final out put

    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException
    {
        int sum = 0;
        /*iterates through all the values available with a key and add them together and give
the

```

```

        final result as the key and sum of its values*/
        while (values.hasNext())
        {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

```

2.7.1 Mapper:

Maps are the individual tasks which transform input records into a intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs. Mapper is an Interface in old API and an abstract class in new API.

The Hadoop Map-Reduce framework spawns one map task for each InputSplit generated by the InputFormat for the job. Mapper implementations can access the Configuration for the job via the `JobContext.getConfiguration()`.

The framework first calls `setup(org.apache.hadoop.mapreduce.Mapper.Context)`, followed by `map(Object, Object, Context)` for each key/value pair in the InputSplit. Finally `cleanup(Context)` is called.

The Shuffle and Sort phase output is partitioned per Reducer. Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.

Users can optionally specify a combiner, via `Job.setCombinerClass(Class)`, to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.

2.7.2 Shuffle and Sort:

The map phase guarantees that the input to the reducer will be sorted on its key. The process by which output of the mapper is sorted and transferred across to the reducers is known as the shuffle.

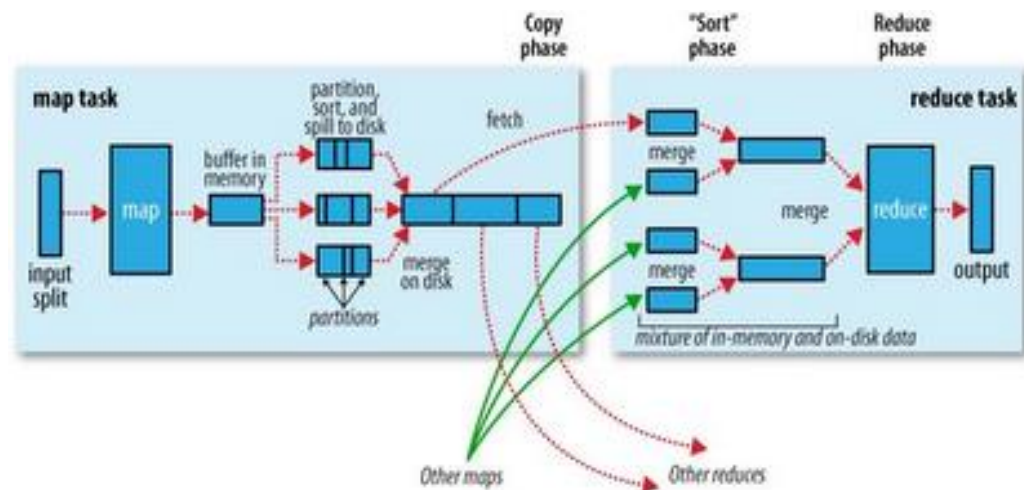


Fig: 2.9 Shuffle and Sort

As shown in fig 2.9, (taken from Hadoop-The Definitive Guide) illustrates the shuffle and sort-phase. The output obtained from the mapper is sorted based on the key which is the input for reducer where shuffling occurs.

2.7.3 Bufferingap Writes:

The mapper does not directly write to the disk rather it takes advantage of buffering the writes. Each mapper has a circular memory buffer with a default size of 100MB which can be tweaked by changing the `io.sort.mb` property. It does the flush in a very smart manner.

2.7.4 Copy phase to the Reducer:

Now, the output of the several map tasks is sitting on different nodes and it needs to get copied over to the node on which the reducer is going to run in order to consume the output of the map tasks. If the data from the map tasks is able to fit inside the reducer's tasktracker memory, then an in-memory merge is performed of the sorted map output files coming from different nodes. As soon as a threshold is reached, the merged output is written onto the disk and the process repeated till all the map tasks have been accounted for this reducer's partition. Then, an on-disk merge is performed in groups of files and a final group of files is directly feeded into the reducer performing an in-memory merge while feeding (thus saving an extra trip to the disk).

2.7.5 Reducer:

Reducing lets you aggregate values together. A reducer function receives an iterator of input values from an input list. It then combines these values together, returning a single output value. The number of reducers by default is 1 and could be modified even to 0. If the reducer class is specified as the Identity reducer in the driver class or not specified, Identity reducer plays the role.

2.7.6 Output Collector:

Output Collector is the generalization of the facility provided by the Map-Reduce framework to collect data output by either the Mapper or the Reducer i.e. intermediate outputs or the output of the job. Output collector is an Interface.

When a reduce task starts, its input is scattered in many files across all the nodes where map tasks ran. If run in distributed mode these need to be first copied to the local filesystem in a copy phase.

Once all the data is available locally it is appended to one file in an append phase. The file is then merge sorted so that the key-value pairs for a given key are contiguous (sort phase). This makes the actual reduce operation simple: the file is read sequentially and the

values are passed to the reduce method with an iterator reading the input file until the next key value is encountered.

At the end, the output will consist of one output file per executed reduce task. The format of the files can be specified with `JobConf.setOutputFormat`. If `SequentialOutputFormat` is used then the output key and value classes must also be specified.

2.7.7 Reporter:

A facility for Map-Reduce applications to report progress and update counters, status information etc.

Mapper and Reducer can use the Reporter provided to report progress or just indicate that they are alive. In scenarios where the application takes significant amount of time to process individual key/value pairs, this is crucial since the framework might assume that the task has timed-out and kill that task.

Applications can also update Counters via the provided Reporter.

2.7.8 Combiner:

When the map operation outputs its pairs they are already available in memory. For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

2.7.9 Partitioner:

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent for reduction. The default partitioner is the HashPartitioner which works on the Hash bucketing mechanism.

2.8 Input Formats

Input format specifies the input file type we are working on. Need to be specified in the driver class. There are several input formats in Hadoop as File Input Format, Sequence File Input Format, File Input Format, Sequence File As Text Input Format, XML File Input Format and so on.

2.8.1 FileInputFormat:

A base class for file-based InputFormat. This provides a generic implementation of `getSplits(JobConf, int)`. Subclasses of `FileInputFormat` can also override the `isSplittable(FileSystem, Path)` method to ensure input-files are not split-up and are processed as a whole by Mappers.

2.8.2 TextInputFormat :

An InputFormat for plain text files. Files are broken into lines. Either linefeed or carriage-return are used to signal end of line. Keys are the position in the file, and values are the line of text.

2.8.3 SequenceFileInputFormat:

A child class of `FileInputFormat`, to go ahead with the binary files. The other child classes of the `SequenceFileInputFormat` are

2.8.3.1 SequenceFileAsTextInputFormat:

This class is similar to SequenceFileInputFormat, except it generates SequenceFileAsTextRecordReader which converts the input keys and values to their String forms by calling toString() method.

2.8.3.2 SequenceFileAsBinaryInputFormat:

InputFormat reading keys, values from SequenceFiles in binary (raw) format.

2.8.3.3 SequenceFileInputFilter:

A class that allows a map/red job to work on a sample of sequence files. The sample is decided by the filter class set by the job.

2.8.4 KeyValueTextInputFormat

An InputFormat for plain text files. Files are broken into lines. Either linefeed or carriage-return are used to signal end of line. Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty.

2.8.5 LineDocInputFormat :

An InputFormat for LineDoc for plain text files where each line is a doc. This is the in dealing with the documents.

2.9 Hadoop Ecosystem

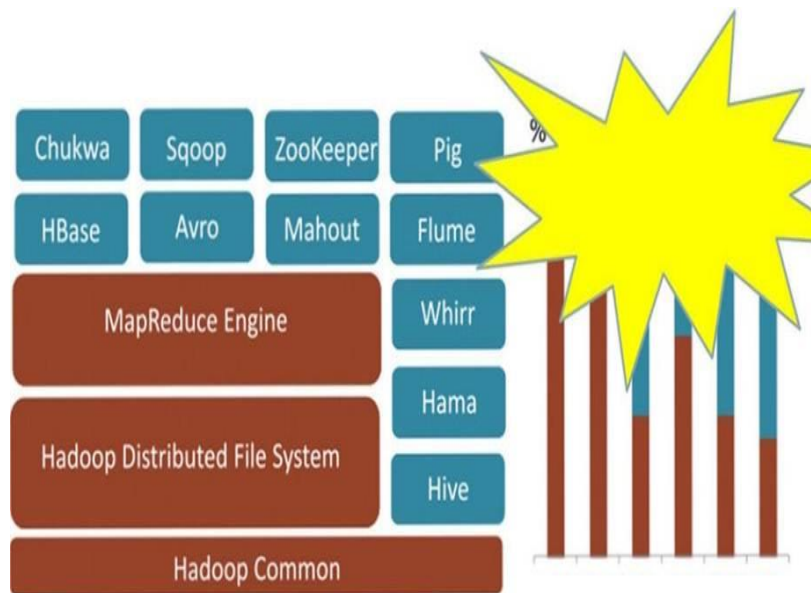


Fig 2.10 Hadoop Ecosystem

As shown in above fig 2.10, Hadoop Ecosystem includes HDFS and MapReduce which form the important components of Hadoop and some of the infrastructures based on Hadoop.

2.9.1 HIVE

Hive is a data warehousing infrastructure based on the Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing (using the map-reduce programming paradigm) on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides a simple query language called Hive QL, which is based on SQL and which enables users familiar with SQL to do ad-hoc querying, summarization and data analysis easily. At the same time, Hive QL also allows traditional map/reduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis that may not be supported by the built-in capabilities of the language.

2.9.2 PIG

Apache Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets.

- (i) Ease of programming: It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
- (ii) Optimization opportunities: The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
- (iii) Extensibility: Users can create their own functions to do special-purpose processing.

Pig Latin queries execute in a distributed fashion on a cluster. The current implementation compiles Pig Latin programs into Map-Reduce jobs, and executes them using Hadoop cluster.

2.9.3 SQOOP

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle into the Hadoop Distributed File System (HDFS), transform the data in HadoopMapReduce, and then export the data back into an RDBMS.

Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance.

2.9.4 Some other Ecosystems

2.9.4.1 OOZIE

Oozie is a Java Web-Application that runs in a Java servlet-container - Tomcat and uses a database to store:

- Workflow definitions
- Currently running workflow instances, including instance states and variables

Oozie workflow is a collection of actions (i.e. Hadoop Map/Reduce jobs, Pig jobs) arranged in a control dependency DAG (Direct Acyclic Graph), specifying a sequence of actions execution.

2.9.4.2 Flume

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application.

2.9.4.3 Chukwa

Chukwa is a Hadoop subproject devoted to large-scale log collection and analysis. Chukwa is built on top of the Hadoop distributed filesystem (HDFS) and MapReduce framework and inherits Hadoop's scalability and robustness. Chukwa also includes a flexible and powerful toolkit for displaying monitoring and analyzing results, in order to make the best use of this collected data.

2.9.4.4 Thrift

A cross-language RPC framework that powers many of Facebook's services, include search, ads, and chat. Among other things, Thrift defines a compact binary serialization format that is often used to persist data structures for later analysis.

2.9.4.5 Scribe

A Thrift service for distributed log file collection. Scribe was designed to run as a daemon process on every node in your data center and to forward log files from any process running on that machine back to a central pool of aggregators. Because of its ubiquity, a major design point was to make Scribe consume as little CPU as possible.

3. SYSTEM REQUIREMENTS

3.1 INPUT SPECIFICATION

The input specifications are:

a) Xml File

The xml file which is downloaded from the web page and containing all the links to other pages is used.

b) HDFS

The xml file which is used as input is stored in HDFS which is a file system of Hadoop.

3.2 OUTPUT SPECIFICATION

The output specifications are:

a) HDFS

From the input which is stored in HDFS is taken and processed to generate the final output. The output is also stored in HDFS .

b) Iterations

The entire accuracy of the output depends on the number of iterations. As the number of iterations increases the accuracy increases.

3.3 SYSTEM REQUIREMENTS

The essential requirements for developing the software of this system include, a minimum of Linux 14.04(64-bit) OS. Also coding language like Java, Python can be suggestable for this system. The software that is needed for this system is Eclipse IDE, Hadoop(single-node setup or multi-node setup), Java any version above 1.6.

At the same time, the essential requirements for developing the hardware of this system includes, a minimum of Processor 1.80 GHz, a Hard disk of 40 GB and a Ram of 4 GB.

The Functional Requirement defines a function of a software system and how the system must behaves when presented with specific inputs or conditions. These may include calculations, data manipulation and processing and other specific functionality.

The Non-Functional requirements, as the name suggests, are those requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability response time and store occupancy. For e.g., in this system storage and managing a large input file is such a requirement should be taken care of. And also response time is important. By using Hadoop, these requirements are satisfied.

4. DESIGN OF WEB PAGE RANKING USING HADOOP

System design serves to bridge the gap between the requirements specified for this system and the final solution for satisfying the requirements. While the requirements specification activity is entirely in the problem domain, the system design is the first step in moving from the problem domain towards the solution domain.

4.1 PROJECT FLOW

Project flow describes the operations which we are going to perform during the project, diagrammatically. This project consists of three Hadoop jobs, Parsing, Calculating and Ordering.

4.1.1 Parsing:

In parsing phase, the xml file is parsed to obtain the key, value pairs of all the links even if they are present multiple times by the mapper. The reducer will store the pages with initial page rank.

4.1.2 Calculating:

During calculation the rank of each page is found with the help of number of outlinks associated with that page. The reducer will calculate the rank and store the output in the same format as that of input.

4.1.3 Ordering:

In ordering the ranks obtained are sorted based on the keys in ascending order by the mapper.

4.2 Analysis and Design

The design of the system is represented in the form of UML diagrams.

4.2.1 DFD

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system, modelling its process aspects. A DFD is often used as a preliminary step to create an overview of the system, which can later be elaborated. It shows what kind of information will be input to and output from the system, where the data will come from and go to, and where the data will be stored. It does not show information about the timing of process or information about whether processes will operate in sequence or in parallel.

4.2.1.1 DFD Level 0

This is a context level DFD, in which the whole system is seen as a single process and emphasizes the interaction between the system, PageRank Algorithm and the input xml file.

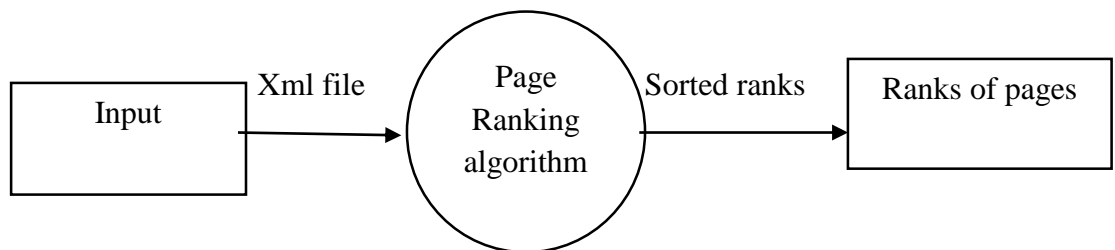


Fig 4.1 DFD of Web Page Ranking using Hadoop at Level 0

As shown in Fig 4.1, an xml file which consists of related pages is given as input to the Page Ranking algorithm which gives output containing page along with their ranks.

4.2.1.2 DFD Level 1

This is a graphical representation that shows the processing of the data in the system and highlights the main functions carried out by the system, such as, Parsing the input xml file, Calculating the Rank and Ordering based on the Rank.

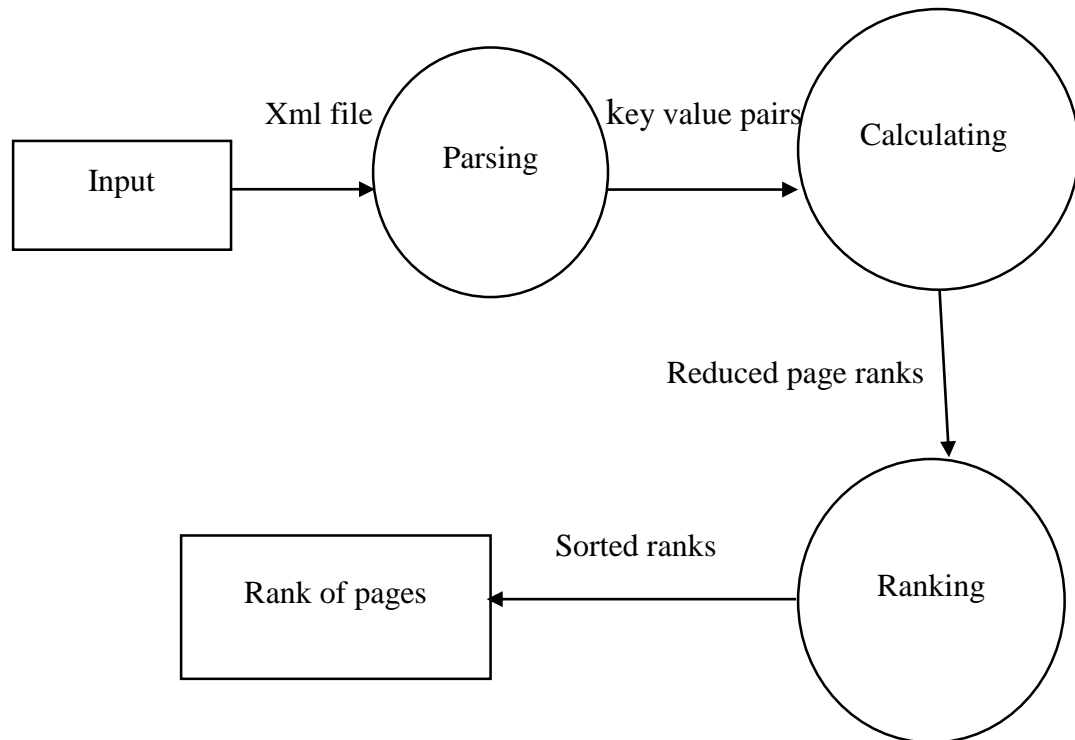


Fig 4.2 DFD of Web Page Ranking using Hadoop at Level 1

As shown in Fig 4.2, an xml file consisting of pages related pages is given as input to Page Ranking algorithm which consists of three jobs. They are Parsing, Calculating and Ranking.

The input xml file is parsed to get the page name and its outgoing links and also to get each page the links to other pages and store the page, initial rank and outgoing links. While calculating, map each outgoing link to the page with its rank and total outgoing links, then calculate the new page rank for the pages. Store the page, new rank and outgoing links and repeat these for more accurate results.

In Ranking job, ordering is done on the rank and they are stored.

4.2.2 UML DIAGRAMS

UML is the international standard notation for object-oriented analysis and design. The object management group defines it. The heart of object-oriented problem solving is the construction of a model. The model abstracts the essential details of the underlying problem from its usually complicated real world. Several modeling tools are wrapped under the heading of unified modeling language. It is a language for visualizing, specifying, constructing, documenting.

4.2.2.1 Usecase Diagram

Use case diagrams are a set of use cases, actors and their relationships. They represent the use case view of a system. A use case represents a particular functionality of a system. It captures the functionality of the system developed to calculate page rank using Hadoop.

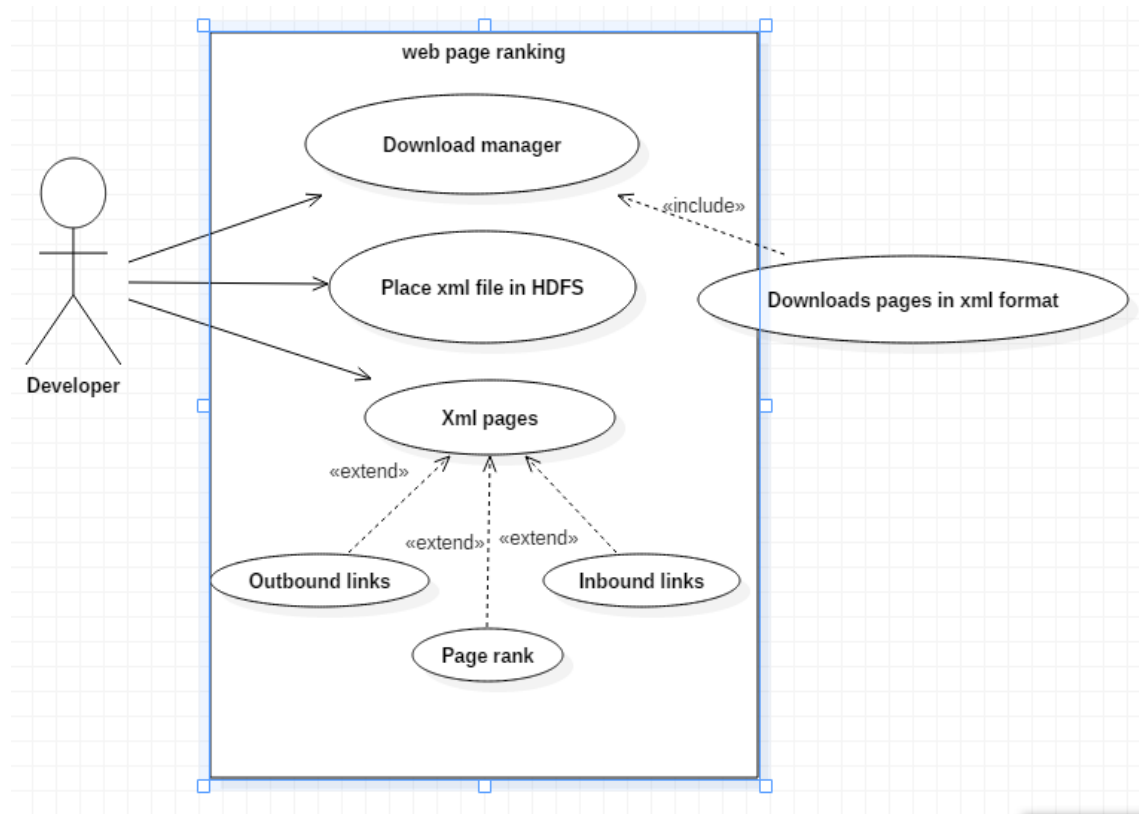


Fig.4.3 Use case Diagram of Web Page Ranking using Hadoop

As shown in Fig. 4.3, the input is download using download manager app by developer in xml format and then the input is placed in HDFS for computing PageRank and then the input data is processed. The input xml file consists of pages which have both inbound links and outbound links. Outbound links are used for calculating the rank.

4.2.2.2 Class diagram:

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations and collaboration. It represents the structure of the system

by showing the classes present in our system along with their members and relationships.

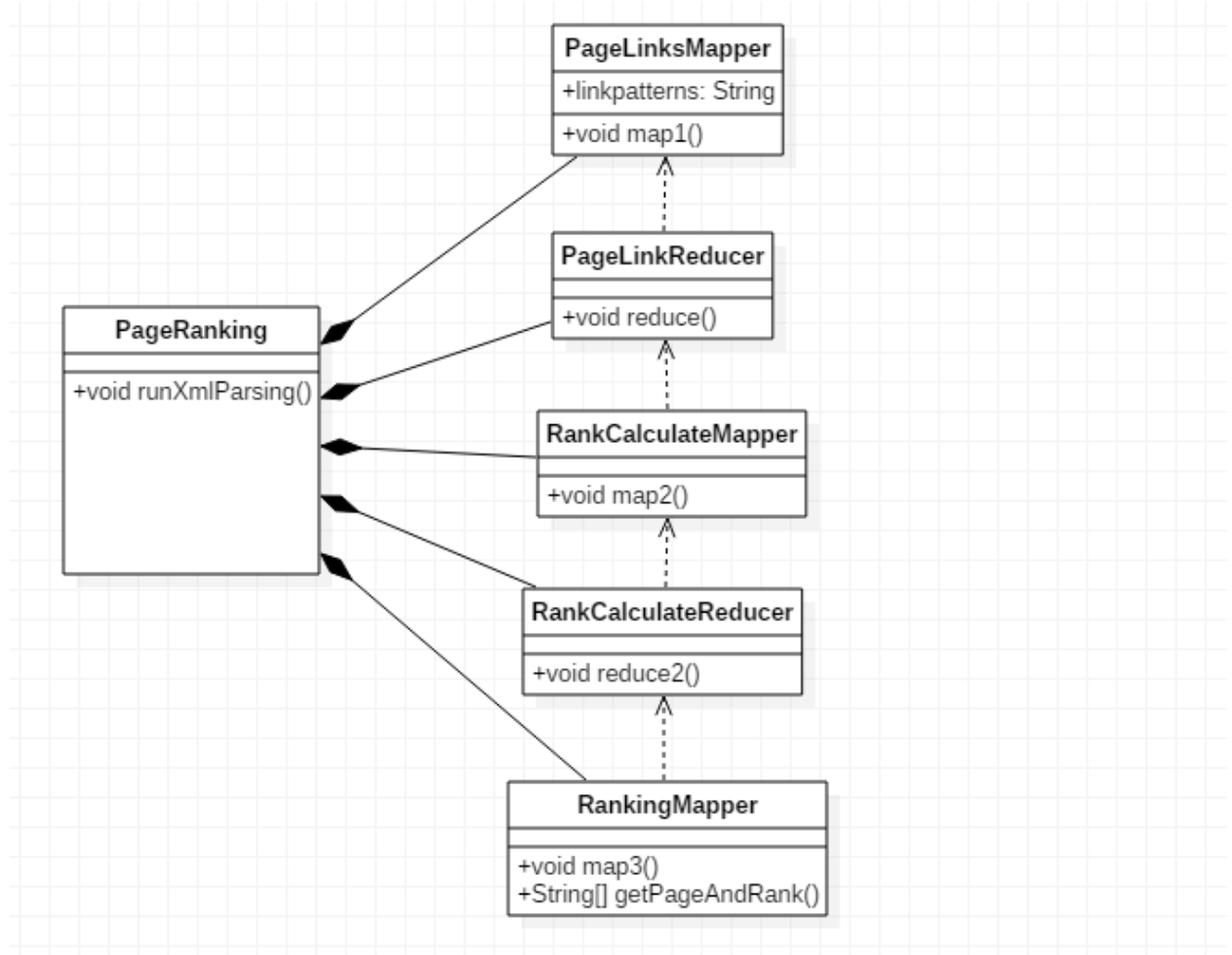


Fig.4.4 Class Diagram of Web Page Ranking using Hadoop

As shown in Fig. 4.4, our system consists of six classes out of which WikiPageRanking is a main class that runs against the hadoop cluster. It mainly consists of all the remaining jobs of the algorithm. Remaining classes are PageLinksMapper which is a mapper that will parse the chunks of xml to key(page) and value(outLinks) tuples, PageLinksReducer which is a reducer that will store the page with the initial PageRank and the outgoing links, RankCalculateMapper which is a mapper that will give output for each link with the combined value page, rank and totalLinks by removing any non-existing page, RankCalculateReducer which is a reducer that will receive the key, values ordered by key.

It will calculate the new pageRank and write it to output for existing pages with the original links, RankingMapper which is a mapper which gives us the output page and its rank.

4.2.2.3 Sequence diagram:

This is a graphical representation that shows the flow of data among the objects Mapper and Reducer of Parsing, Calculating and Ranking. This shows how processes in the system operate with one another and in what particular order giving page and its respective rank at the end.

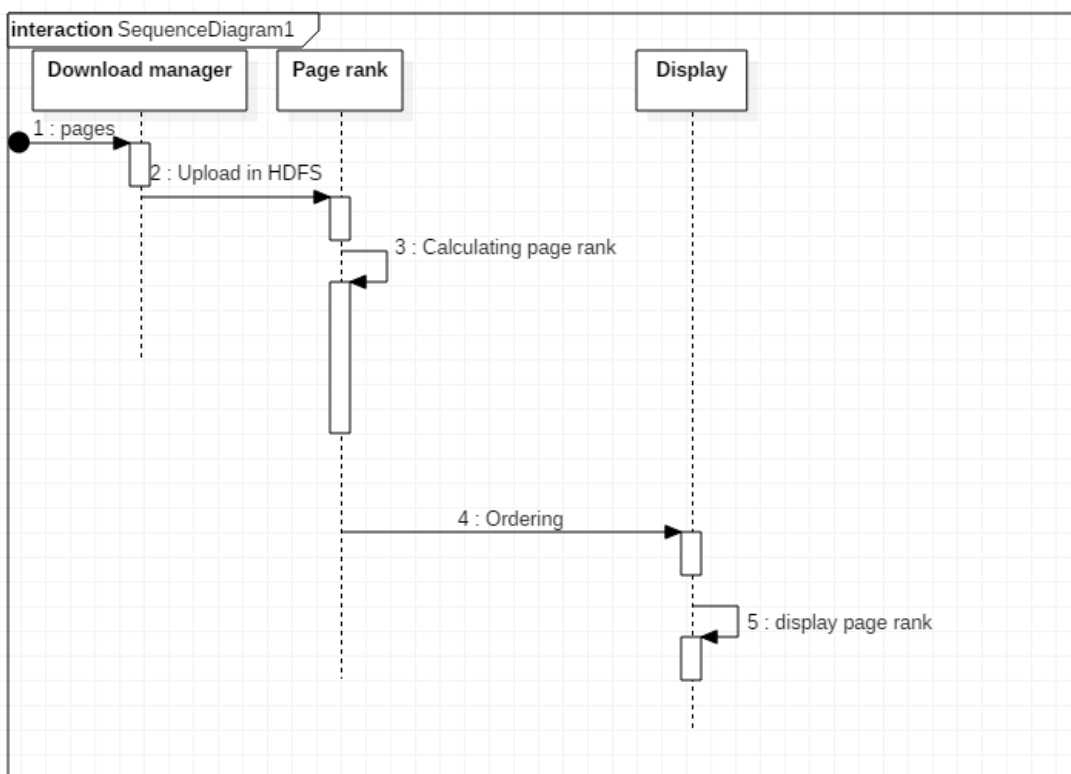


Fig.4.5 Sequence Diagram of Web Page Ranking using Hadoop

As shown in Fig. 4.5, initially input i.e., the related pages are download using download manager app in xml format. This xml file is stored in HDFS and then the PageRank algorithm calculates the rank, then ordering is done on the ranks. At the end, Page and its respective Rank are displayed.

4.2.2.4 Activity Diagram:

Activity diagram describes the flow of control in a system. So it consists of activities and links. The flow can be sequential, concurrent or branched. Starting from downloading input xml file, uploading it into HDFS, Parsing the xml file, Calculating the Rank and Ordering based on the Rank and displaying the Page along with its Rank.

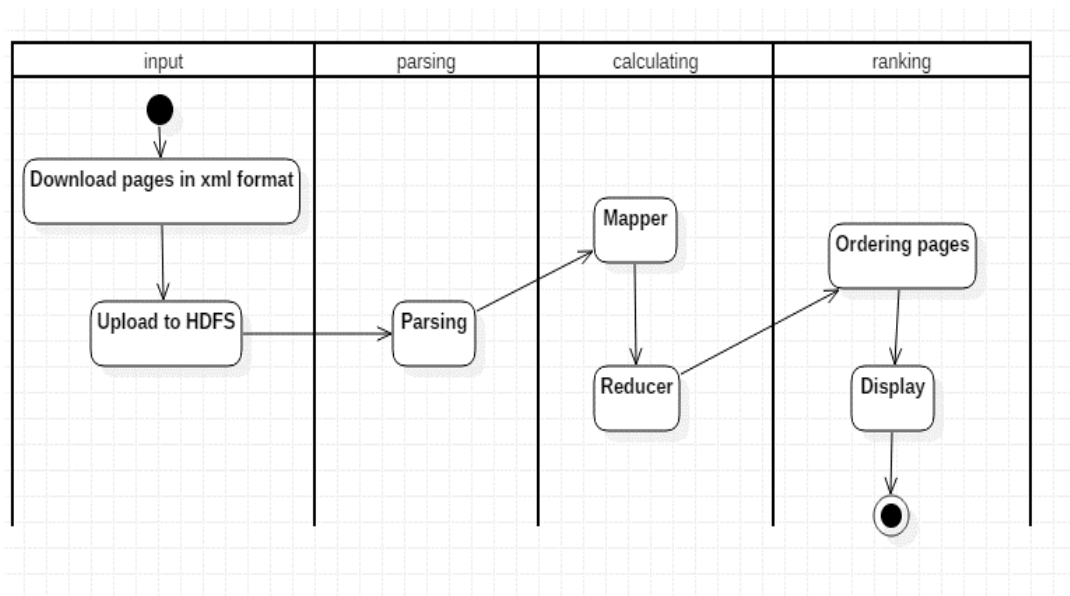


Fig.4.6 Activity Diagram of Web Page Ranking using Hadoop

As shown in Fig. 4.6, there are four phases in this algorithm. They are input, Parsing, Calculating, Ranking. Initially, the input is downloaded in xml format and it is stored in HDFS. The input given in xml format is parsed to get page and its outgoing links. Then the using this information page rank is calculated in two jobs, mapper and reducer. In Ranking phase, ordering of ranks is done and then the pages along with their ranks are displayed.

4.2.2.5 State chart diagram:

StateChart diagram showcases the state transitions which are responsible for state change of the system. The state transits from an initial stage, where an xml file is uploaded into HDFS to a final state where an output is displayed i.e., Page along with its Rank.

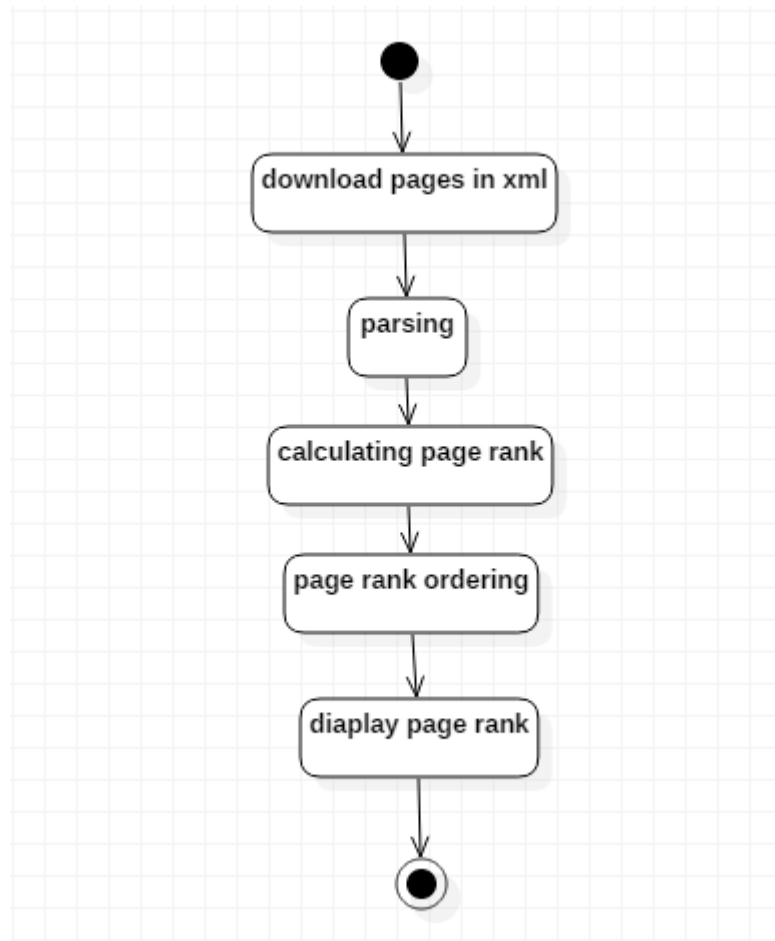


Fig.4.7 State Chart diagram of Web Page Ranking using Hadoop

As shown in Fig. 4.7, the input is downloaded in xml format and then sent for parsing, where only useful information like page and its outgoing links are given as output. Then page rank is calculated by PageRank algorithm and ordering is done. Then the result is displayed i.e., page with its rank.

5. SYSTEM IMPLEMENTATION

The first phase of our project is installing Hadoop cluster. The proper approach would be a 2 step approach. First step would be to install single-node Hadoop machines, configure and test them as local Hadoop systems. Second step would be to merge that single-node systems into a multi-node cluster. So first let us see how to setup the single node machine.

Environment Setup

- Download Ubuntu Linux 14.04 LTS.
- Download Hadoop -2.5.1
- Once the Linux server is set-up, install openssh, by executing the following command.

```
$sudo apt-get install openssh-server
```

5.1 HADOOP NODE SETUP

PREREQUISITES.

- Java™ must be installed.

Command:

```
$ tar -zxvf jdk-8u17-linux-x64.tar.gz (to unzip the tar file)
```

```
$ sudo update-alternative --install "/usr/bin/java" "java"  
"/usr/lib/java/jdk1.8.0_17/bin/java" 1
```

```
$ sudo update-alternative --install "/usr/bin/javac" "javac"  
"/usr/lib/java/jdk1.8.0_17/bin/javac" 1
```

```
$ sudo update-alternative --install "/usr/bin/javaws" "javaws"  
"/usr/lib/java/jdk1.8.0_17/bin/javaws" 1
```

- ssh must be installed and sshd must be running to use the Hadoop scripts that manage remote Hadoop daemons.

Command :

```
$ sudo apt-get install ssh
```

```
$ sudo apt-get install rsync
```

5.1.1 SINGLE NODE SETUP:

Hadoop can also be run on a single-node in a pseudo-distributed mode where each Hadoop daemon runs in a separate Java process.

Configuration

Command:

```
$gedit core_site.xml
```

```
<configuration>
<property>
<name>fs.defaultFS</name>
<value>hdfs://localhost:9000</value>
</property>
</configuration>
```

Command:

```
$gedit hdfs_site.xml
```

```
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```


Command:

```
$gedit mapred_site.xml
```

```
<configuration>  
<property>  
<name>mapreduce.framework.name</name>  
<value>yarn</value>  
</property>  
</configuration>
```

Command:

```
$gedit yarn_site.xml
```

```
<configuration>  
<property>  
<name>yarn.nodemanager.aux-services</name>  
<value>mapreduce_shuffle</value>  
</property>  
</configuration>
```

5.1.2 HADOOP MULTIPLE NODE SETUP

Initially a static ip has to be set up if the systems which are in different networks to be connected to same network.

Command:

```
$sudo vi etc/network/interfaces
```

```
auto eth0
```

```
iface eth0 inet static
```

```
    address 10.5.60.7
```

```
    network 10.5.60.0
```

```
netmask 255.255.255.0
broadcast 10.5.60.255
gateway 10.5.60.1
dns-nameservers 10.5.60.2
```

command:

\$ifup

ifup brings a network interface up, making it available to transmit and receive data.

\$ifdown

ifdown takes a network interface down, placing it in a state where it cannot transmit or receive data.

Passphraseless ssh

Execute the following commands

Command:

\$ssh

\$ssh-keygen -t rsa

\$ssh-copy-id -i ~/ .ssh/id_rsa.pub

\$ssh-copy-id -i /home/hadoop/ .ssh/d_rsa.pub hadoop@master

\$ssh-copy-id -i /home/hadoop/ .ssh/d_rsa.pub hadoop@slave1

If not able to ssh without password then

Command:

\$ eval “\$(ssh-agent -s)”

\$ssh-add

Configuration

core-site.xml: \$gedit core_site.xml

<configuration>

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://master:54310</value>
</property>
```

```
</configuration>
```

hdfs-site.xml: \$gedit hdfs_site.xml

```
<configuration>
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/usr/local/hadoop/hadoop-2.5.1/hadoop_data/hdfs/datanode</value>
</property>

</configuration>
```

yarn-site.xml: \$gedit yarn_site.xml

```
<configuration>

<!-- Site specific YARN configuration properties -->
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
```

```

<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>master:8025</value>
</property>
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>master:8030</value>
</property>
<property>
<name>yarn.resourcemanager.address</name>
<value>master:8050</value>
</property>

</configuration>

```

mapred-site.xml: \$gedit mapred_site.xml

```

<configuration>
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
</configuration>

```

.bashrc:\$gedit ~/.bashrc

```

# JAVA_HOME directory setup
export JAVA_HOME=/usr/lib/java/jdk1.8.0_73
export PATH=$PATH:$JAVA_HOME/bin

```

```
# HADOOP_HOME directory setup
export HADOOP_HOME=/usr/local/hadoop/hadoop-2.5.1
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
hadoop-env.sh:
# The java implementation to use.
export JAVA_HOME=/usr/lib/java/jdk1.8.0_73
```

To create a jar file with all the .class files

Command: `$jar cvf [name of jar file] [input file] *.class`

If Data node doesn't run:

Command:
`$rm -rf /usr/local/hadoop/hadoop-2.5.1/hadoop_data/*`
`$bin/hdfs namenode -format`

To check for Hadoop Demons:

Command:
`$start-all.sh`
to start all the services in master node and even the slave node in case of multiple node cluster.
`$jps`
to check the services running.
`$stop-all.sh`
to stop the services.

5.2 IMPLEMENTATION

The entire process is split into three different Hadoop jobs:

Parsing

Calculating and
ordering

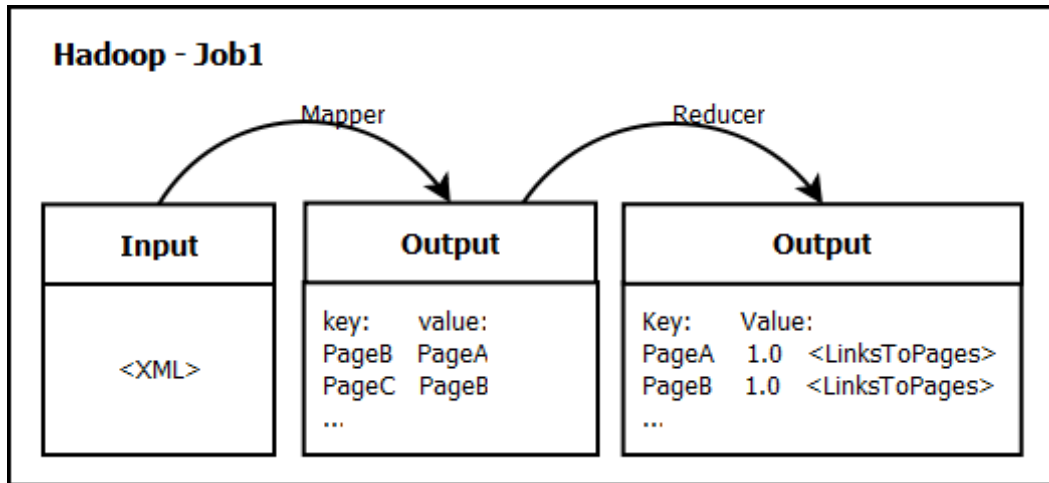


Fig 5.1 Hadoop- Job 1

As shown in fig 5.1, In the Hadoop job 1 the big wiki xml is parsed into articles. In the Hadoop mapping phase, the page name and its outgoing links are retrieved. In the Hadoop reduce phase, for each wiki page the links to other pages are drawn. Finally, the page, initial rank and outgoing links are stored.

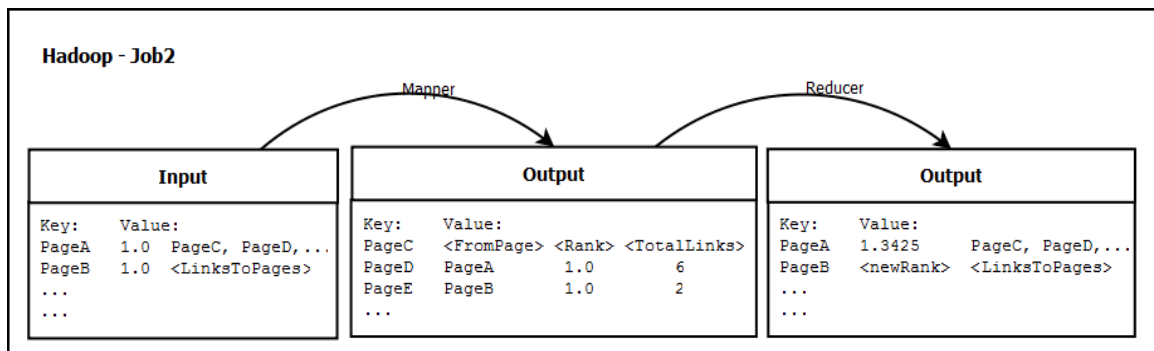


Fig 5.2 Hadoop- Job 2

As shown in fig 5.2, Hadoop job 2: for calculating the new PageRank. In the mapping phase, each outgoing link is mapped to the page with its rank and total out going links. In the reduce phase, new page rank for the pages is calculated. Finally, the page, initial rank and outgoing links are stored. This process is repeated for more accurate results.

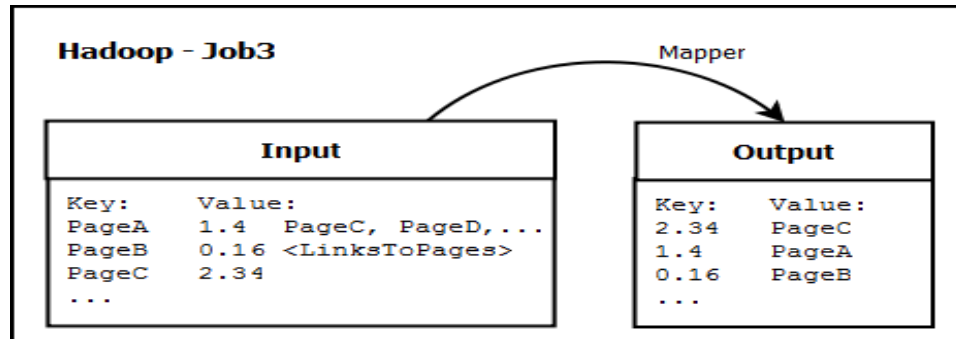


Fig 5.3 Hadoop- Job 3

As shown in fig 5.3, Hadoop job 3 to map the rank and the page. The rank and the page are stored (ordered on rank). Pages are displayed in sorted order.

5.2.1 Hadoop Job 1: Parse The XML to Page with Links

A page can be downloaded as a xml file by adding Special:Export to the URL.
E.g. to get the XML for the wiki page about wiki

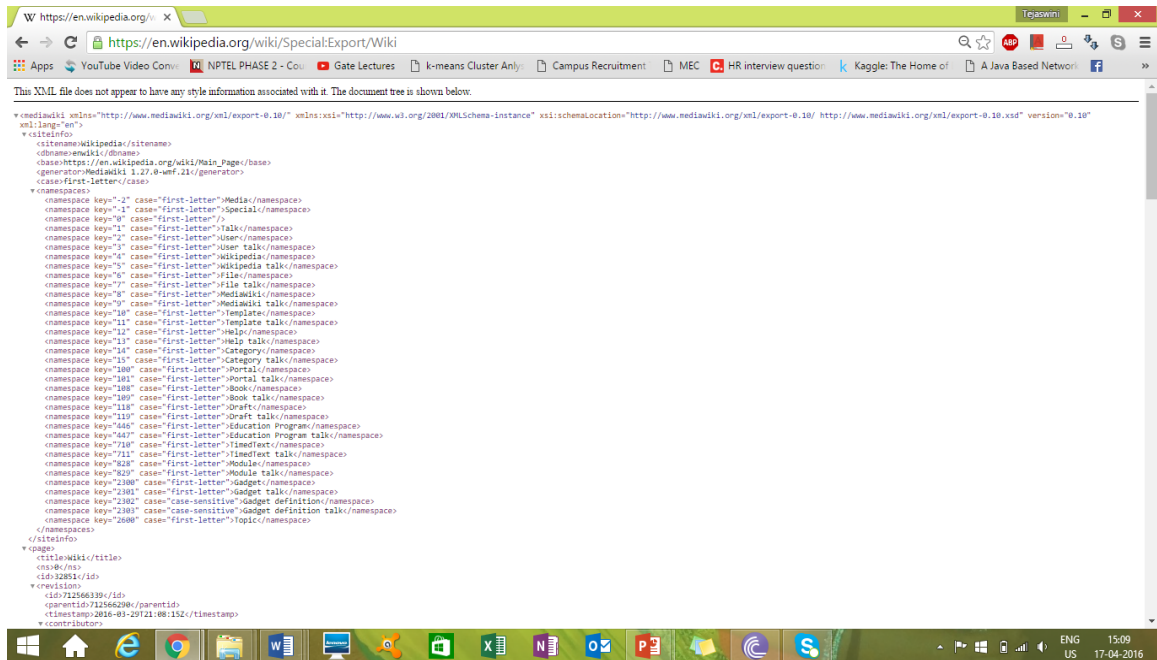


Fig 5.4 Sample xml file of wiki

As shown in fig 5.4, the xml file along with the links to other pages

In the xml structure with meta data about the site and the latest revision, the main part needed is within the title and the text tags. The xml file is downloaded and is placed in HDFS in /usr/[hostname]/[user]/wiki/input dir. When we run the job , the location where the file should be placed is shown, so that it can be placed in correct directory after the first run.

Coming to the classes for job 1. The first class needed is the main class that should be run against Hadoop cluster. The main class contains all the jobs.

WikiPageRanking.java

Main class runs against Hadoop cluster and more jobs can be added.

The normal InputFormat class is the TextInputFormat to get nice input for the map. As the whole xml is needed as input, mahout XmlInputFormat is used to get nice input for the mapper interface. It will chop the xml into parts within the given start and end tag <page>.

WikiPageLinksMapper.java

The mapper class that will parse the chunks of xml to key page and value outlinks tuples. In the implementation all the links are added to the map, even if they appear multiple times on the page.

WikiLinksReducer.java

The reducer class will store the page with the initial Page Rank and the outgoing links. This output format is used as input format for the next job. Key<tab>rank<tab>CommaSeparatedList-of-inksOtherPages.

5.2.2 Hadoop Job 2: Calculating New Page Rank

This will run after job 1. This job is used to calculate the new ranking and to generate the same output format as that input, so this job runs multiple times. The PageRank will become more accurate after multiple runs, so the job is executed for a few times.

Mapper

This job has its own mapper and reducer classes.

RankCalculateMapper.java

Some links point to wiki pages that do not exist (yet). In the browser we can see them as red links.

For each link there is an output with the combined value page, rank and total link.

The last output of the mapper is the page and the original links. We need the link so the reducer is able to produce the correct output.

Reducer

The reducer will receive the key, values ordered by key. In a distributed environment the map is cut in slices and all nodes will get a share. The reducer will calculate the new PageRank and write it to output for the existing page with the original links.

RankCalculatorReduce.java

The output of the reducer contains the new PageRank for the existing pages with the links on those pages.

Main class has to be configured so that the new job is executed for a couple of times after the xml-parsing job.

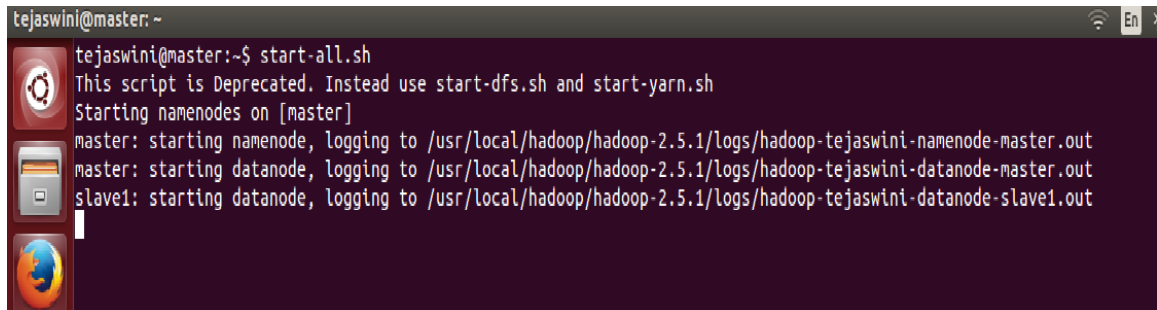
5.2.3 Hadoop Job 3: Last Run On Page Rank

This is a simple job that uses the input to get the page and rank. And map the key: rank to value: page. Hadoop will do the sorting on key. There is no need to implement a reducer. The mapper and sorting is enough for results, the ordered list.

RankingMapper.java

The sorting on the key is ascending, so at the bottom is the highest rank page. In this only mapper is sufficient to sort the obtained pages ranks in the ascending order.

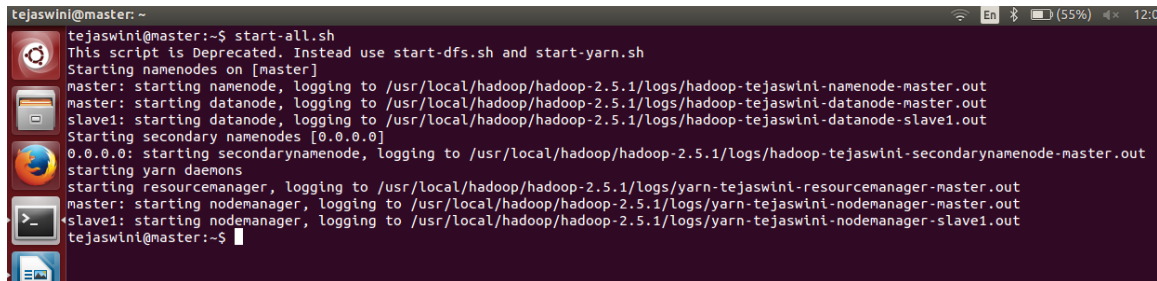
6. RESULTS AND DISCUSSION

A terminal window titled 'tejaswini@master: ~' with a dark purple background. The prompt is 'tejaswini@master:~\$'. The user enters 'start-all.sh'. The output shows a deprecation warning, followed by 'Starting namenodes on [master]'. Then, it shows 'master: starting namenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-namenode-master.out', 'master: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-master.out', and 'slave1: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-slave1.out'. On the left side of the terminal, there are four icons: a gear, a folder, a document, and a globe.

```
tejaswini@master:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [master]
master: starting namenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-namenode-master.out
master: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-master.out
slave1: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-slave1.out
```

Fig 6.1 starting Hadoop demons

As shown in the fig 6.1, in the process of starting nodes,the above is during starting the namenode and datanode of master and the datanode of slave.

A terminal window titled 'tejaswini@master: ~' with a dark purple background. The prompt is 'tejaswini@master:~\$'. The user enters 'start-all.sh'. The output shows a deprecation warning, followed by 'Starting namenodes on [master]'. Then, it shows 'master: starting namenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-namenode-master.out', 'master: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-master.out', 'slave1: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-slave1.out', 'Starting secondary namenodes [0.0.0.0]', '0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-secondarynamenode-master.out', 'starting yarn daemons', 'starting resourcemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-resourcemanager-master.out', 'master: starting nodemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-nodemanager-master.out', 'slave1: starting nodemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-nodemanager-slave1.out', and finally 'tejaswini@master:~\$'. On the left side of the terminal, there are four icons: a gear, a folder, a document, and a globe.

```
tejaswini@master:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [master]
master: starting namenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-namenode-master.out
master: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-master.out
slave1: starting datanode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-datanode-slave1.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/hadoop-2.5.1/logs/hadoop-tejaswini-secondarynamenode-master.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-resourcemanager-master.out
master: starting nodemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-nodemanager-master.out
slave1: starting nodemanager, logging to /usr/local/hadoop/hadoop-2.5.1/logs/yarn-tejaswini-nodemanager-slave1.out
tejaswini@master:~$
```

Fig 6.2 starting Hadoop Demons

As shown in fig 6.2, starting the demons at master and slave by the master. All the 5 nodes of the master and Datanode and Node manager of slave will start.

```
tejaswini@slave1: ~  
tejaswini@slave1:~$ jps  
2288 Jps  
2355 NodeManager  
2237 DataNode  
tejaswini@slave1:~$
```

Fig 6.3 Hadoop Demons at slave

As shown in the fig 6.3, The Hadoop Demons which are in active in slave1. jps is used to check for if the Demons are started.

Namenode information - Mozilla Firefox

Namenode information x Connecting... x +

master:50070/dfshealth.html#tab-overview

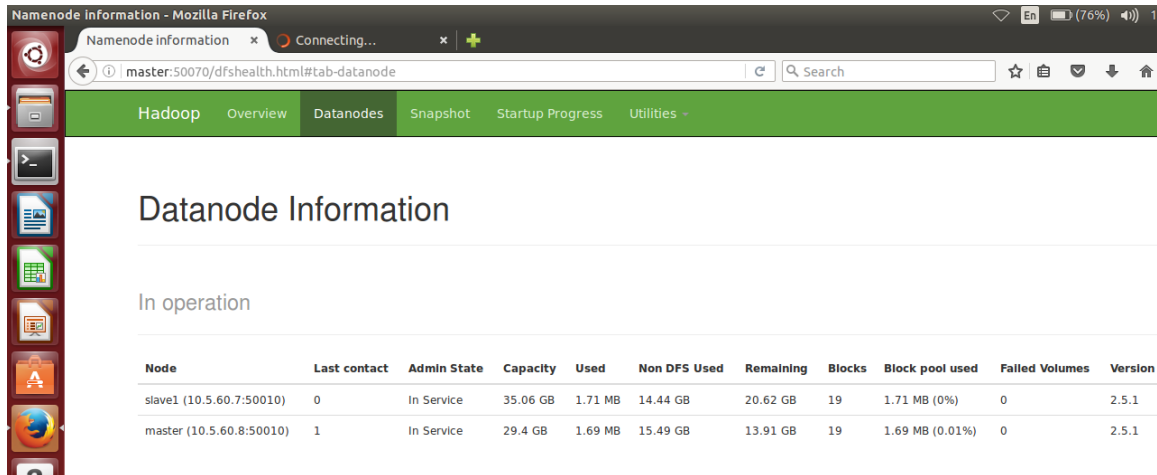
Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Overview 'master:54310' (active)

| | |
|-----------------------|---|
| Started: | Mon Apr 18 12:07:42 IST 2016 |
| Version: | 2.5.1, r2e18d179e4a8065b6a9f29cf2de9451891265cce |
| Compiled: | 2014-09-05T23:11Z by jenkins from (detached from 2e18d17) |
| Cluster ID: | CID-2676de62-a708-4680-93e6-33b308a31cab |
| Block Pool ID: | BP-1094236171-10.5.60.8-1458104228004 |

Fig 6.4 Overview in browser

As shown in the fig 6.4, The overview of the master in the browser when the master is active.



Namenode Information - Mozilla Firefox

master:50070/dfshealth.html#tab-datanode

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

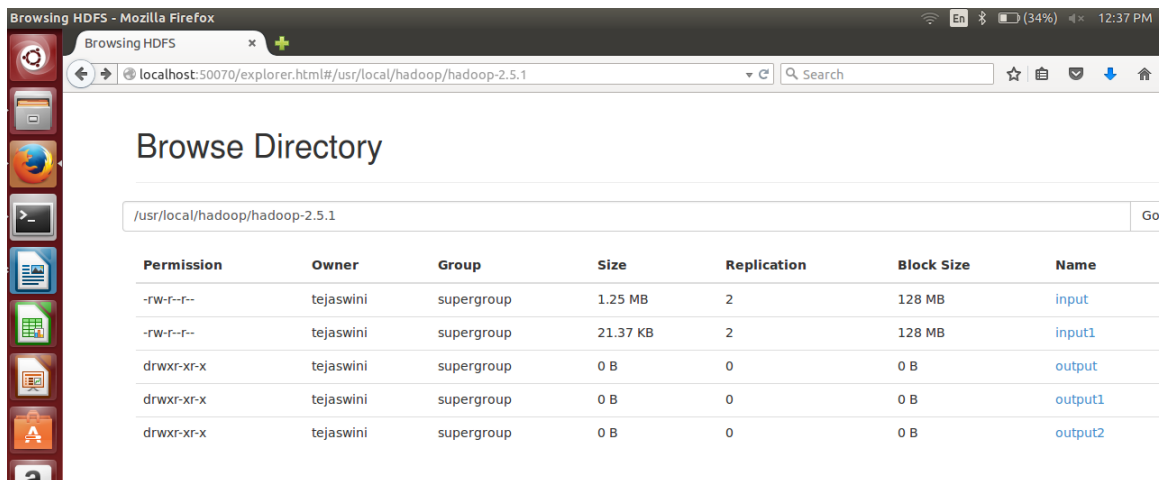
Datanode Information

In operation

| Node | Last contact | Admin State | Capacity | Used | Non DFS Used | Remaining | Blocks | Block pool used | Failed Volumes | Version |
|--------------------------|--------------|-------------|----------|---------|--------------|-----------|--------|-----------------|----------------|---------|
| slave1 (10.5.60.7:50010) | 0 | In Service | 35.06 GB | 1.71 MB | 14.44 GB | 20.62 GB | 19 | 1.71 MB (0%) | 0 | 2.5.1 |
| master (10.5.60.8:50010) | 1 | In Service | 29.4 GB | 1.69 MB | 15.49 GB | 13.91 GB | 19 | 1.69 MB (0.01%) | 0 | 2.5.1 |

Fig 6.5 Datanode Information in browser

As shown in the fig 6.5, Information about the data node. That is about the data stored in both master and slave along with their size and number of blocks.



Browsing HDFS - Mozilla Firefox

localhost:50070/explorer.html#/usr/local/hadoop/hadoop-2.5.1

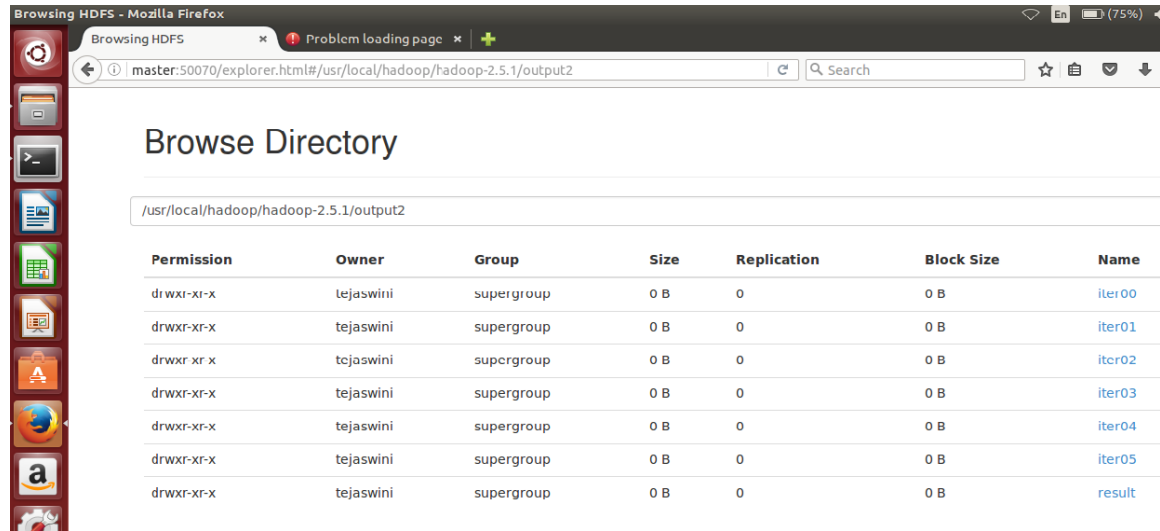
Browse Directory

/usr/local/hadoop/hadoop-2.5.1 Go

| Permission | Owner | Group | Size | Replication | Block Size | Name |
|------------|-----------|------------|----------|-------------|------------|-------------------------|
| -rw-r--r-- | tejaswini | supergroup | 1.25 MB | 2 | 128 MB | input |
| -rw-r--r-- | tejaswini | supergroup | 21.37 KB | 2 | 128 MB | input1 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | output |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | output1 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | output2 |

Fig 6.6, Browse directory in the browser.

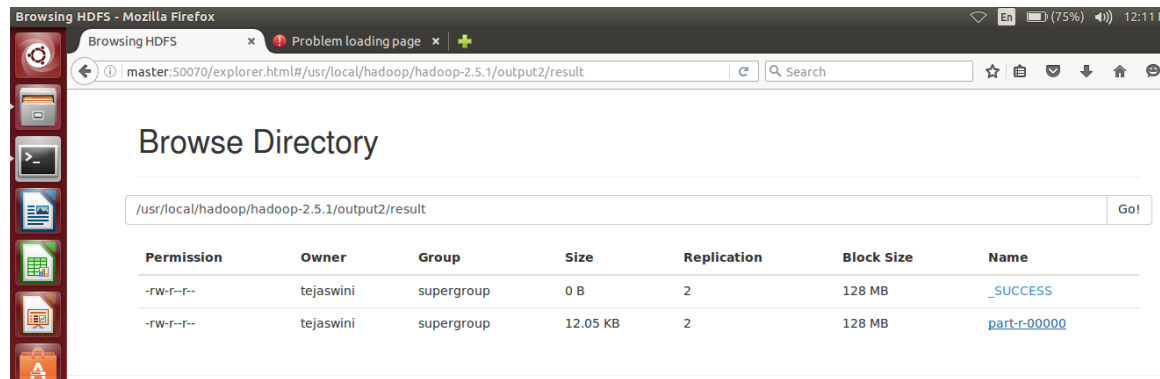
As shown in fig 6.6,the browse directory consists of all the input files loaded into HDFS and the output files with their file size, replication factor and the block size.



| Permission | Owner | Group | Size | Replication | Block Size | Name |
|------------|-----------|------------|------|-------------|------------|------------------------|
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter00 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter01 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter02 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter03 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter04 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | iter05 |
| drwxr-xr-x | tejaswini | supergroup | 0 B | 0 | 0 B | result |

Fig 6.7, Browse directory with iterations.

As shown in fig 6.7, The output file with the result at each iteration and the final result is stored in the file result.



| Permission | Owner | Group | Size | Replication | Block Size | Name |
|------------|-----------|------------|----------|-------------|------------|------------------------------|
| -rw-r--r-- | tejaswini | supergroup | 12.05 KB | 2 | 128 MB | _SUCCESS |
| -rw-r--r-- | tejaswini | supergroup | 12.05 KB | 2 | 128 MB | part-r-00000 |

Fig 6.8, Result file in the browser directory

As shown in the fig 6.8, the result file consists of two file one is `_SUCCESS` which just indicates that the process has been successfully completed and the other contains the final output.

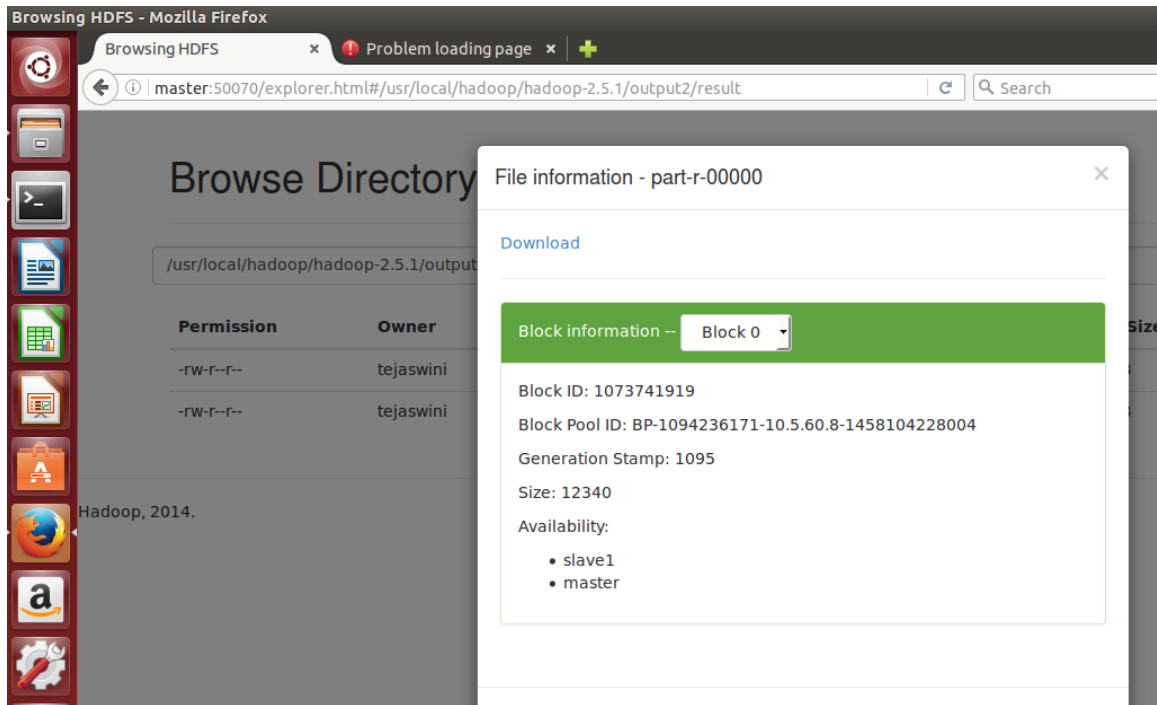


Fig 6.9, Resultant file information

As shown in the fig 6.9, the output obtained at the master node can be downloaded by any other slave which are in connection with the master.

```

tejaswini@master: ~/Desktop/wrd
0.23794532 Janner
0.23794532 Januarius
0.23794532 Januar
0.23794532 Januoudpioç
0.23794532 Janetiro
0.27749097 mutu
0.27749097 banda
0.28254735 April
0.28945726 agua
0.3098902 february
0.31079185 mei
0.31079185 mai
0.31230193 esposa
0.31230193 mujer
0.31230193 femme
0.31230193 épouse
0.31230193 vrouw
0.31230193 wife
0.31230193 woman
0.32589066 januari
0.34882104 abril
0.40499994 lipnçmi
0.42891455 water
0.50824654 sEtémbe
0.50988305 sánzá_ya_ltbwá
0.62769663 máyi
1.0 Haiti
1.2195992 mái
1.4187865 mwasi
1.6054894 sánzá_ya_mibalé
1.6429617 febwalli
1.6865076 sánzá_ya_mitáno
1.7262713 máyi
1.835226 sánzá_ya_yambo
1.8375689 sánzá_ya_misato
1.8375689 mársi
1.8764079 yanwalli
2.7916455 mwasi
2.964305 aprili
3.0641499 yúni
tejaswini@master:~/Desktop/wrd$

```

Fig 6.10, Output in terminal

As shown in the fig 6.10, the ranks of all the pages after sorting.

```

H:\project\final project\ouptu of 1gb\part-r-00000
N 1.0 azoti
Nikarago 1.0 Nikalagwa
O 1.0 oksijÉ>InÁ
Rafantaka 1.0 nkÁsi
Ramahavaly 1.0 nkÁsi
Ratsimahalahy 1.0 nkÁsi
Scheidig 1.0 sÉ>tÉ>ImbÉ>,sÁ;nzÁ;_ya_libwÁ;
September 1.0 sÉ>tÉ>ImbÉ>,sÁ;nzÁ;_ya_libwÁ;,sÉ>tÉ>ImbÉ>,sÁ;nzÁ;_ya_libwÁ;
Wonnemond 1.0 sÁ;nzÁ;_ya_mÁtÁ;no,mÁ;yÁ
aba 1.0 tatÁ;
abrial 1.0 aprÁli
abril 1.0 aprÁli,aprÁli,aprÁli
abriu 1.0 aprÁli
afo 1.0 makÁ;la
agua 1.0 mÁ;i,mÁ;yi
aho 1.0 ngÁ;Á
ahy 1.0 ngÁ;Á
aina 1.0 bomÉ"i
akao 1.0 kÁ°nÁ;
akeo 1.0 kÁ°nÁ;
akese 1.0 kÁ°nÁ;
aketo 1.0 Á;wa

```

Fig 6.11 Mapping

As shown in fig. 6.11 All the pages are initially given rank 1, after parsing the xml file (that is mapping all the pages with their corresponding links and reducing the links to remove duplicates).



| File Name | Rank | Link/Path |
|--------------|------|--|
| dimy | 1.0 | mAtA;no |
| dionari | 1.0 | kaboni |
| dipoavatra | 1.0 | pilipilA |
| e | 1.0 | E>IÉ>I |
| echtgenote | 1.0 | mwasi |
| efitra | 1.0 | elA&A |
| eka | 1.0 | E>IÉ>I |
| ekany | 1.0 | E>IÉ>I |
| endry | 1.0 | mamA; |
| enero | 1.0 | sA;nzA;_ya_yambo, yanwA;li |
| eny | 1.0 | E>IÉ>I |
| eo | 1.0 | A;wa |
| eo_ho_eo | 1.0 | kA°nA; |
| esposa | 1.0 | mwÇZsA |
| esE'ngE'' | 1.0 | plaisir, vreugde, pleasure, plezier, geluk, nsA;A, bomE'ngE'', liyA°ki |
| etosa | 1.0 | kA°nA; |
| etsy | 1.0 | A;wa |
| etA | 1.0 | A;wa |
| etA_ho_etA | 1.0 | A;wa |
| ezanezana | 1.0 | bosA&ndA& |
| fahadimy | 1.0 | mAtA;no |
| fahafatesana | 1.0 | liwA&c |
| fahalaiana | 1.0 | bosA&ndA& |

Fig 6.12 Reducing

As shown in fig. 6.12 Each line consists of page name, initial rank and outgoing links. The links obtained from the mapping phase are reduced to remove any duplicates if present, so as to increase the accuracy.

| | | |
|----------------------|------------|---------------------------|
| Ineny | 0.14999998 | mamA; |
| Januar | 0.23358428 | yanwA;li,sA;nzA;_ya_yambo |
| January | 0.23358428 | sA;nzA;_ya_yambo,yanwA;li |
| June | 0.21822035 | yA*ni |
| Juni | 0.21822035 | yA*ni |
| JAmnner | 0.23358428 | yanwA;li,sA;nzA;_ya_yambo |
| Lenzing | 0.23470163 | sA;nzA;_ya_mAsA;to,mA;rsi |
| LonkA;sA;_ya_libosA; | 0.14999998 | LokA;sA;_ya_libosA; |
| Madagascar | 0.14999998 | Madagasikari |
| Madagasikara | 0.14999998 | Madagasikari |
| Mai | 0.22742413 | sA;nzA;_ya_mAtA;no,mA;yA |
| Main_Page | 0.14999998 | LokA;sA;_ya_libosA; |
| March | 0.23470163 | mA;rsi,sA;nzA;_ya_mAsA;to |
| May | 0.22742413 | mA;yA,sA;nzA;_ya_mAtA;no |
| Monday | 0.14999998 | mokE"lE" _ya_libosA; |
| Mutu | 0.14999998 | mutu |
| Mwasi | 0.14999998 | mwCZsA |
| MAMrz | 0.23470163 | sA;nzA;_ya_mAsA;to,mA;rsi |
| N | 0.14999998 | azoti |
| NikaragoA | 0.14999998 | , , Nikalagwa |

Fig 6.13 Calculating mapper

As shown in fig. 6.13 Each line consists of page name, initial rank and outgoing links. The rank is reduced by the iterations to increase accuracy. Based on the number of outgoing links to each of the existing links, ranks are calculated. This process of calculating and mapping occurs in several iterations to increase the accuracy and derive good results.

| | | |
|-----------------------|------------|--|
| Aeme | 0.14999998 | molAmo |
| Aene | 0.14999998 | mpA*nda |
| A@pouse | 0.42890626 | mwasi |
| E"ke"tE"ibE> | 0.16378637 | 1,1,october,1,octubre,1,outubro,1,1,oktober,1,October,iVi*i,iZi*iI*I,,iVi*i,iZi*iI |
| i'ieif-i'i-I, | 0.4155107 | aprAali |
| i'ieif-i'i-i;I, | 0.4155107 | aprAali |
| i'ipii-i;i;i@I, | 0.46212363 | yA*ni |
| i=i;iw;i;i-i-i;i;I, | 0.49591634 | yanwA;li,sA;nzA;_ya_yambo |
| i=i;iw;i;i, | 0.46212363 | yA*ni |
| i=i;iw;i;i;I, | 0.46212363 | yA*ni |
| iei-i-i-I, | 0.46414408 | sA;nzA;_ya_mAtA;no,mA;yA |
| iei-i-i;i;I, | 0.46414408 | mA;yA,sA;nzA;_ya_mAtA;no |
| iei-ii"i-I, | 0.48439062 | sA;nzA;_ya_mAsA;to,mA;rsi |
| iei-ii"i;i;I, | 0.48439062 | sA;nzA;_ya_mAsA;to,mA;rsi |
| i;ipii-ii;i-i-ii-i;I, | 0.47007984 | sA;nzA;_ya_mAbalA@,febwa;li |
| i;iw;i;i-i-ii-i, | 0.47007984 | sA;nzA;_ya_mAbalA@,febwa;li |
| x"x x"x"x" | 0.14999998 | yanwA;li,sA;nzA;_ya_yambo |
| x"x"x" 0.14999998 | | mA;yA,sA;nzA;_ya_mAtA;no |
| x"x"x"x"x"x" | 0.14999998 | febwa;li,sA;nzA;_ya_mAbalA@ |
| @@*U\$@+@S@+ | 0.14999998 | yA*ni |
| ***** | | |

Fig 6.14 Calculating Reducer

As shown in fig. 6.14, Each line consists of page and its respective rank. After mapping the ranks to their respective pages, reducer reduces the pages by removing the pages which are not existing. Even the process of reducing occurs in several iterations. The number of iterations increases the accuracy.

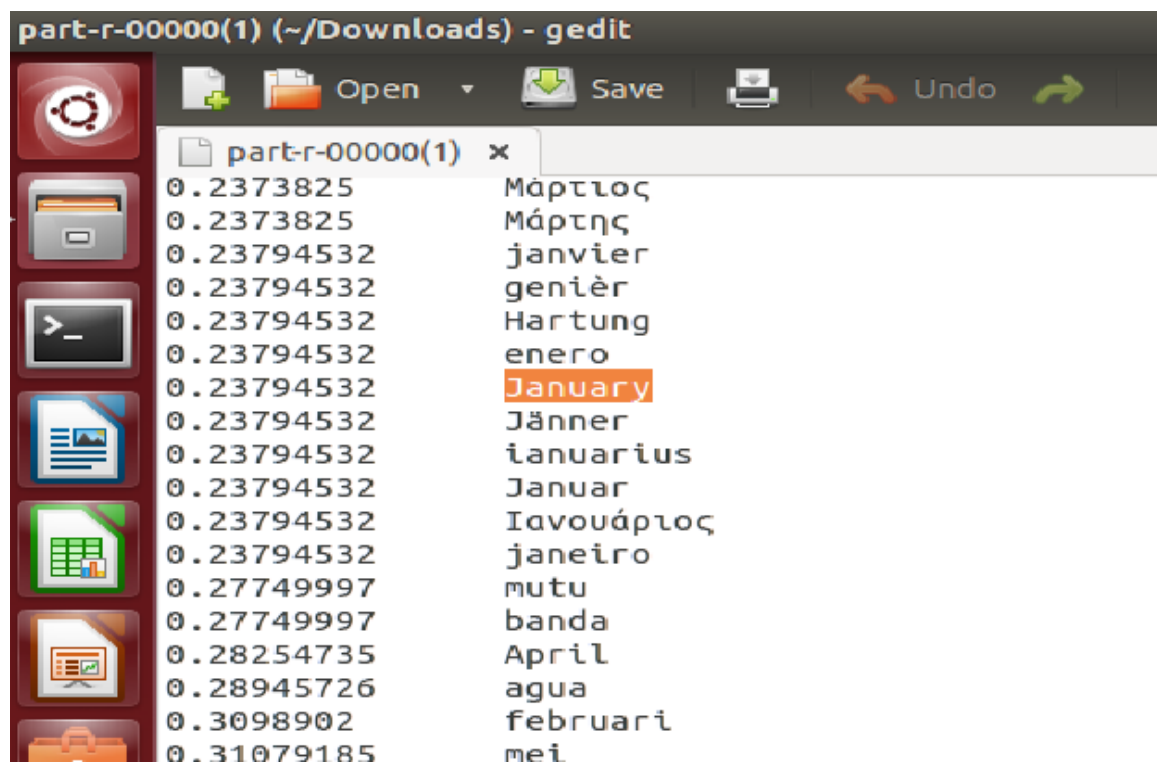


Fig 6.15 Rank mapping

As shown in fig. 6.15, Each line consists of page and its respective rank. The final output is obtained after several iterations where mapping and reducing has done. In this phase the pages with their corresponding ranks and their links are sorted based on their rank and the final result is obtained.

7. CONSLUSION AND FUTURE WORK

7.1 CONCLUSION

Huge amount of data is available on the internet and it is also growing rapidly year by year. If a user wants to search for particular content, he enters a query. From this vast amount of data, relevant information or pages should be given to the user. So, PageRank Algorithm is used by Google search engine for getting relevant pages. This algorithm is best solution for getting relevant pages when given a query by user.

The amount of data produced by us from the beginning of time till 2003 was 5 billion gigabytes. If you pile up the data in the form of disks, it may fill an entire football field. The same amount was created in every two days in 2011, and in every ten minutes in 2013. Scientists have concluded that more data has been generated in the past two years than in the entire history of mankind, and the rate of data generation is intensifying. This rate is still growing enormously. Though all this information produced is meaningful and can be useful when processed, it is being neglected. A recent Cisco Systems report projects the global Internet traffic will exceed the zettabyte (1,000 Exabyte) threshold by the end of 2016.

This huge amount of data is called as BigData which cannot be processed using traditional computing. Hadoop framework is capable enough to develop applications capable of running on clusters of computers and they could perform complete statistical analysis for a huge amounts of data. Hadoop runs applications using the MapReduce algorithm, where the data is processed in parallel on different CPU nodes. In our project, MapReduce and HDFS file system are used for calculating Page Ranking for a dump of Wikipedia pages. The output of this project is the page name and its respective rank. By using Hadoop, we can store and manage this huge amount of data by dividing the jobs on the clusters.

7.2 FUTURE WORK

Generally, Page Rank of a web page is calculated using many input/output parameters. In this project, we considered only one parameter i.e., outgoing links of the page, but sometimes the user may not get the exact web page needed. There are different parameters like incoming links and content of the query given by the user and also the page. Thus, instead of considering only one parameter i.e., outgoing links alone, using more number of parameters gives good results. For future works, it would be necessary to include parameters like incoming links and the content without any performance degradation of the algorithm.

REFERENCCES

- [1] N. Duhan, A. K. Sharma and Bhatia K. K., “Page Ranking Algorithms: A Survey, Proceedings of the IEEE International Conference on Advance Computing, 2009.
- [2] International Journal of Innovative Research in Computer and Communication Engineering Vol. 2, Issue 2, February 2014.
- [3] A. Altman and M. Tennenholtz. Ranking systems: the PageRank axioms. In Proceedings of the 6th ACM conference on Electronic commerce, pages 1–8, 2005.
- [4] <http://search.iiit.ac.in/uploads/Efficient%20Ranking%20Using%20Hadoop.pdf>
- [5] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank, <http://arxiv.org/abs/1006.2880>, 2010.
- [6] <https://help.ubuntu.com/community/NetworkConfigurationCommandLine/Automatic>
- [7] Hadoop: The Definitive Guide, Third Edition by Tom White, Jan 2012.
- [8] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in PageRank computation: When one iteration is sufficient. SIAM J. Numer. Anal., 45(2):890–904, 2007.
- [9] Konstantin Shvachko, Yahoo! Sunnyvale, California USA, Hairong Kuang, Sanjay Radia and Robert Chansler :The Hadoop Distributed File System. In the proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), pages 1-100, 2010.

APPENDIX

WikiPageRanking.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

public class WikiPageRanking extends Configured implements Tool
{
    private static NumberFormat nf = new DecimalFormat("00");

    public static void main(String[] args) throws Exception
    {
        System.exit(ToolRunner.run(new Configuration(), new WikiPageRanking(), args));
    }

    @Override
    public int run(String[] args) throws Exception
    {
        boolean isCompleted = runXmlParsing(args[0], args[1]+"/iter00");
        if (!isCompleted)
```

```

        return 1;
String lastResultPath = null;
for (int runs = 0; runs < 5; runs++)
{
    String inPath = args[1]+"/iter" + nf.format(runs);
    lastResultPath = args[1]+"/iter" + nf.format(runs + 1);
    isCompleted = runRankCalculation(inPath, lastResultPath);
if (!isCompleted)
    return 1;
}

isCompleted = runRankOrdering(lastResultPath, args[1]+"/result");
if (!isCompleted) return 1;
return 0;
}

public boolean runXmlParsing(String inputPath, String outputPath) throws IOException,
ClassNotFoundException, InterruptedException
{
    Configuration conf = new Configuration();
    conf.set(XmlInputFormat.START_TAG_KEY, "<page>");
    conf.set(XmlInputFormat.END_TAG_KEY, "</page>");
    Job xmlHakker = Job.getInstance(conf, "xmlHakker");
    xmlHakker.setJarByClass(WikiPageRanking.class);
    // Input / Mapper
    FileInputFormat.addInputPath(xmlHakker, new Path(inputPath));
    xmlHakker.setInputFormatClass(XmlInputFormat.class);
    xmlHakker.setMapperClass(WikiPageLinksMapper.class);
    xmlHakker.setMapOutputKeyClass(Text.class);
    // Output / Reducer
    FileOutputFormat.setOutputPath(xmlHakker, new Path(outputPath));

```



```

xmlHakker.setOutputFormatClass(TextOutputFormat.class);
xmlHakker.setOutputKeyClass(Text.class);
xmlHakker.setOutputValueClass(Text.class);
xmlHakker.setReducerClass(WikiLinksReducer.class);
return xmlHakker.waitForCompletion(true);
}

private boolean runRankCalculation(String inputPath, String outputPath) throws
IOException, ClassNotFoundException, InterruptedException
{
    Configuration conf = new Configuration();
    Job rankCalculator = Job.getInstance(conf, "rankCalculator");
    rankCalculator.setJarByClass(WikiPageRanking.class);
    rankCalculator.setOutputKeyClass(Text.class);
    rankCalculator.setOutputValueClass(Text.class);
    FileInputFormat.setInputPaths(rankCalculator, new Path(inputPath));
    FileOutputFormat.setOutputPath(rankCalculator, new Path(outputPath));
    rankCalculator.setMapperClass(RankCalculateMapper.class);
    rankCalculator.setReducerClass(RankCalculateReduce.class);
    return rankCalculator.waitForCompletion(true);
}

private boolean runRankOrdering(String inputPath, String outputPath) throws
IOException, ClassNotFoundException, InterruptedException
{
    Configuration conf = new Configuration();
    Job rankOrdering = Job.getInstance(conf, "rankOrdering");
    rankOrdering.setJarByClass(WikiPageRanking.class);
    rankOrdering.setOutputKeyClass(FloatWritable.class);
    rankOrdering.setOutputValueClass(Text.class);
    rankOrdering.setMapperClass(RankingMapper.class);
    FileInputFormat.setInputPaths(rankOrdering, new Path(inputPath));

```

```

        FileOutputFormat.setOutputPath(rankOrdering, new Path(outputPath));
        rankOrdering.setInputFormatClass(TextInputFormat.class);
        rankOrdering.setOutputFormatClass(TextOutputFormat.class);
        return rankOrdering.waitForCompletion(true);
    }
}

```

WikiPageLinksMapper.java

```

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
import java.nio.charset.CharacterCodingException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class WikiPageLinksMapper extends Mapper<LongWritable, Text, Text, Text>
{
    private static final Pattern wikiLinksPattern = Pattern.compile("\\[.+?\\]");
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
        InterruptedException
    {
        // Returns String[0] = <title>[TITLE]</title>
        //      String[1] = <text>[CONTENT]</text>
        // !! without the <tags>.
        String[] titleAndText = parseTitleAndText(value);
        String pageString = titleAndText[0];
        if(notValidPage(pageString))

```

```

        return;
    Text page = new Text(pageString.replace(' ', '_'));
    Matcher matcher = wikiLinksPattern.matcher(titleAndText[1]);
    //Loop through the matched links in [CONTENT]
    while (matcher.find())
    {
        String otherPage = matcher.group();
        //Filter only wiki pages.
        //- some have [[realPage|linkName]], some single [realPage]
        //- some link to files or external pages.
        //- some link to paragraphs into other pages.
        otherPage = getWikiPageFromLink(otherPage);
        if(otherPage == null || otherPage.isEmpty())
            continue;
        // add valid otherPages to the map.
        context.write(page, new Text(otherPage));
    }
}

private boolean notValidPage(String pageString)
{
    return pageString.contains(":");
}

private String getWikiPageFromLink(String aLink)
{
    if(isNotWikiLink(aLink))
        return null;
    int start = aLink.startsWith("[[") ? 2 : 1;
    int endLink = aLink.indexOf("]");
    int pipePosition = aLink.indexOf("|");
    if(pipePosition > 0)

```

```

{
    endLink = pipePosition;
}
int part = aLink.indexOf("#");
if(part > 0)
{
    endLink = part;
}
aLink = aLink.substring(start, endLink);
aLink = aLink.replaceAll("\\s", "_");
aLink = aLink.replaceAll(",", "");
aLink = sweetify(aLink);
return aLink;
}

private String sweetify(String aLinkText)
{
    if(aLinkText.contains("&"))
        return aLinkText.replace("&", "&");

    return aLinkText;
}

private String[] parseTitleAndText(Text value) throws CharacterCodingException
{
    String[] titleAndText = new String[2];
    int start = value.find("<title>");
    int end = value.find("</title>", start);
    start += 7; //add <title> length.
    titleAndText[0] = Text.decode(value.getBytes(), start, end-start);
    start = value.find("<text>");
    start = value.find(">", start);

```

```

        end = value.find("</text>", start);
        start += 1;
        if(start == -1 || end == -1)
        {
            return new String[]{"", ""};
        }
        titleAndText[1] = Text.decode(value.getBytes(), start, end-start);
        return titleAndText;
    }

    private boolean isNotWikiLink(String aLink) {
        int start = 1;
        if(aLink.startsWith("[[")){
            start = 2;
        }
        if( aLink.length() < start+2 || aLink.length() > 100) return true;
        char firstChar = aLink.charAt(start);
        if( firstChar == '#') return true;
        if( firstChar == ',') return true;
        if( firstChar == '.') return true;
        if( firstChar == '&') return true;
        if( firstChar == "\"") return true;
        if( firstChar == '-') return true;
        if( firstChar == '{') return true;
        if( aLink.contains(":")) return true; // Matches: external links and translations links
        if( aLink.contains(",")) return true; // Matches: external links and translations links
        if( aLink.contains("&")) return true;
        return false;
    }
}

```

WikiLinksReducer.java

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
public class WikiLinksReducer extends Reducer<Text, Text, Text, Text>
{
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException
    {
        String pagerank = "1.0\t";
        boolean first = true;
        for (Text value : values)
        {
            if(!first) pagerank += ",";
            pagerank += value.toString();
            first = false;
        }
        context.write(key, new Text(pagerank));
    }
}
```

RankCalculateMapper.java

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
public class RankCalculateMapper extends Mapper<LongWritable, Text, Text, Text>
```

```

{
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException
    {
        int pageTabIndex = value.find("\t");
        int rankTabIndex = value.find("\t", pageTabIndex+1);
        String page = Text.decode(value.getBytes(), 0, pageTabIndex);
        String pageWithRank = Text.decode(value.getBytes(), 0, rankTabIndex+1);
        // Mark page as an Existing page (ignore red wiki-links)
        context.write(new Text(page), new Text("!"));
        // Skip pages with no links.
        if(rankTabIndex == -1) return;
        String links = Text.decode(value.getBytes(), rankTabIndex+1, value.getLength()-
(rankTabIndex+1));
        String[] allOtherPages = links.split(",");
        int totalLinks = allOtherPages.length;
        for (String otherPage : allOtherPages)
        {
            Text pageRankTotalLinks = new Text(pageWithRank + totalLinks);
            context.write(new Text(otherPage), pageRankTotalLinks);
        }
        // Put the original links of the page for the reduce output
        context.write(new Text(page), new Text("|" + links));
    }
}

```

RankCalculateReduce.java

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class RankCalculateReduce extends Reducer<Text, Text, Text, Text>
{
    private static final float damping = 0.85F;

    @Override
    public void reduce(Text page, Iterable<Text> values, Context context) throws
IOException, InterruptedException
    {
        boolean isExistingWikiPage = false;
        String[] split;
        float sumShareOtherPageRanks = 0;
        String links = "";
        String pageWithRank;
        // For each otherPage:
        // - check control characters
        // - calculate pageRank share <rank> / count(<links>)
        // - add the share to sumShareOtherPageRanks
        for (Text value : values)
        {
            pageWithRank = value.toString();
            if(pageWithRank.equals("!")) {
                isExistingWikiPage = true;
                continue;
            }
            if(pageWithRank.startsWith("|")){
                links = "\t"+pageWithRank.substring(1);
            }
        }
    }
}
```



```

        continue;
    }
    split = pageWithRank.split("\\t");
    float pageRank = Float.valueOf(split[1]);
    int countOutLinks = Integer.valueOf(split[2]);
    sumShareOtherPageRanks += (pageRank/countOutLinks);
}
if(!isExistingWikiPage)
return;
    float newRank = damping * sumShareOtherPageRanks + (1-damping);
    context.write(page, new Text(newRank + links));
}
}

```

RankingMapper.java

```

import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
import java.nio.charset.CharacterCodingException;

public class RankingMapper extends Mapper<LongWritable, Text, FloatWritable, Text>
{
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException
    {
        String[] pageAndRank = getPageAndRank(key, value);
        float parseFloat = Float.parseFloat(pageAndRank[1]);
    }
}

```

```

        Text page = new Text(pageAndRank[0]);
        FloatWritable rank = new FloatWritable(parseFloat());
        context.write(rank, page);
    }

    private String[] getPageAndRank(LongWritable key, Text value) throws
CharacterCodingException {
        String[] pageAndRank = new String[2];
        int tabPageIndex = value.find("\t");
        int tabRankIndex = value.find("\t", tabPageIndex + 1);
        // no tab after rank (when there are no links)
        int end;
        if (tabRankIndex == -1) {
            end = value.getLength() - (tabPageIndex + 1);
        } else {
            end = tabRankIndex - (tabPageIndex + 1);
        }
        pageAndRank[0] = Text.decode(value.getBytes(), 0, tabPageIndex);
        pageAndRank[1] = Text.decode(value.getBytes(), tabPageIndex + 1, end);
        return pageAndRank;
    }
}

```

XmlInputFormat.java

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DataOutputBuffer;

```

```

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import java.io.IOException;

/**
 * Reads records that are delimited by a specific begin/end tag.
 */
public class XmlInputFormat extends TextInputFormat {
    public static final String START_TAG_KEY = "xmlinput.start";
    public static final String END_TAG_KEY = "xmlinput.end";

    @Override
    public RecordReader<LongWritable, Text> createRecordReader(InputSplit split,
TaskAttemptContext context) {
        try {
            return new XmlRecordReader((FileSplit) split, context);
        } catch (IOException e) {
            throw new RuntimeException("TODO : refactor this...");
        }
    }
}

/**
 * XMLRecordReader class to read through a given xml document to output xml
 * blocks as records as specified by the start tag and end tag
 */
public static class XmlRecordReader extends RecordReader<LongWritable, Text> {
    private final byte[] startTag;
    private final byte[] endTag;

```

```

private final long start;
private final long end;
private final FSDataInputStream fsin;
private final DataOutputBuffer buffer = new DataOutputBuffer();
private LongWritable key = new LongWritable();
private Text value = new Text();
public XmlRecordReader(FileSplit split, TaskAttemptContext context) throws
IOException {
    Configuration conf = context.getConfiguration();
    startTag = conf.get(START_TAG_KEY).getBytes("utf-8");
    endTag = conf.get(END_TAG_KEY).getBytes("utf-8");
    // open the file and seek to the start of the split
    start = split.getStart();
    end = start + split.getLength();
    Path file = split.getPath();
    FileSystem fs = file.getFileSystem(conf);
    fsin = fs.open(split.getPath());
    fsin.seek(start);
}
@Override
public void initialize(InputSplit split, TaskAttemptContext context) throws
IOException, InterruptedException {
}
@Override
public boolean nextKeyValue() throws IOException, InterruptedException {
    if (fsin.getPos() < end) {
        if (readUntilMatch(startTag, false)) {
            try {
                buffer.write(startTag);
                if (readUntilMatch(endTag, true)) {

```

```

        key.set(fsin.getPos());
        value.set(buffer.getData(), 0, buffer.getLength());
        return true;
    }
} finally {
    buffer.reset();
}
}
}
return false;
}

@Override
public LongWritable getCurrentKey() throws IOException, InterruptedException {
    return key;
}

@Override
public Text getCurrentValue() throws IOException, InterruptedException {
    return value;
}

@Override
public void close() throws IOException {
    fsin.close();
}

@Override
public float getProgress() throws IOException {
    return (fsin.getPos() - start) / (float) (end - start);
}

private boolean readUntilMatch(byte[] match, boolean withinBlock) throws
IOException {
    int i = 0;

```

```

while (true) {
    int b = fsin.read();
    // end of file:
    if (b == -1) return false;
    // save to buffer:
    if (withinBlock) buffer.write(b);
    // check if we're matching:
    if (b == match[i]) {
        i++;
        if (i >= match.length) return true;
    } else i = 0;
    // see if we've passed the stop point:
    if (!withinBlock && i == 0 && fsin.getPos() >= end) return false;
}
}
}
}

```