

Masters Project (CSCI-6690-02-S15)

COMPUTING PAGERANKS OF WIKIPEDIA PAGES

Satyendra Gurjar¹

Advisor: Prof. Barun Chandra²

University of New Haven, West Haven, CT

Summer 2015

¹sgurj1@unh.newhaven.edu

²bchandra@newhaven.edu

Abstract

Objective of this project is to compute PageRanks of Wikipedia pages using distributed MapReduce framework, Apache Hadoop. Apache Hadoop is an open source project that implements a distributed filesystem (HDFS) and MapReduce framework. PageRank is a probability distribution over nodes in the graph quantifying the likelihood that a random walk over the edges will arrive at particular node. In this project we parse the Wikipedia pages and build directed graph structure. Each node in the graph represents a Wikipedia page and directed edges from that node represents all links to other Wikipedia pages. Graph structure is stored in the HDFS. PageRank algorithm is implemented in MapReduce programming model, that we run over Hadoop distributed cluster. Hadoop cluster is built using Amazon Elastic Compute Cloud (EC2).

Contents

1	Concepts	2
1.1	PageRank	2
1.2	MapReduce	4
1.3	Graph Representation	5
1.4	Hadoop	6
1.4.1	Hadoop Distributed FileSystem (HDFS)	6
1.4.2	YARN	11
2	Design	12
2.1	Wikipedia Dataset	12
2.1.1	Parsing Wikipedia Page XML	13
2.1.2	Types of Articles	13
2.2	Amazon Elastic Compute Cloud (EC2)	14
2.2.1	Types of Instances	14
2.2.2	Types of Storage	16
2.2.3	Which instance type to choose for Hadoop cluster	16
3	Implementation	17
3.1	Hadoop Cluster	17
3.1.1	Creating Server Instances	17
3.1.2	Installation	19
3.1.3	Configuration	19
3.1.4	Format HDFS	23
3.1.5	Starting all Hadoop processes	23
3.2	Wikipedia Dataset in HDFS.	24
3.3	Running Jobs	24
3.3.1	Job 1: Parsing Wikipedia XML to build Adjucency List Graph . . .	24
3.3.2	Job 2: Computing PageRank of Wikipedia Pages Graph	26
3.3.3	Job 3: Finding Top K pages	30
3.3.4	Source Code	30
4	Conclusion	30

List of Tables

1	Inbound ports in Security Group	19
3	Top 25 Pages of Wikipedia English Pages	30

List of Figures

1	Small Graph	3
2	HDFS Architecture, Source [6]	8
3	HDFS File Read, Source [7]	9
4	HDFS File Write, Source [7]	10
5	YARN MapReduce Job Flow, Source [7]	11
6	EC2 Instances	17

1 Concepts

1.1 PageRank

PageRank [1] is a link analysis algorithm that provides numerical value to nodes in a directed graph, for example graph of webpages where hyperlinks to other pages are directed edges. The higher the page rank is more “important” the node is in the graph.

Let consider a small graph in figure 1. It has 5 nodes, A, B, C, D and E. We assign each node equal page rank, $1/\text{Number_of_nodes}$ ($1/5$). In first iteration each nodes sends its pagerank divided by number of its outgoing links, for example, A sends pagerank of $1/5$ to B, and D sends pagerank of $1/10$ (since it has 2 outgoing links) to C and E. In this iteration each nodes sum up all the pageranks it has received and that would be pagerank of that node, for example, pagerank of node C is $1/10$ and of node B is $2/5$. In next iteration, each node sends its pagerank divided by its number of outgoing links to all the nodes it has outgoing links to. We can keep iterating until difference of pageranks become smaller than is fixed delta or we can run a fix number of iterations. In the original PageRank paper [1] convergence on a graph with 322 million edges was reached in 52 iterations.

At the end of each iteration sum of all the pageranks are 1, which makes it a probability distribution and its a stochastic process, can also be described as Markov chain, which describes that pageranks will converge and become stable after sufficient iterations.

This simlified view of pagerank algorithm has two issues:

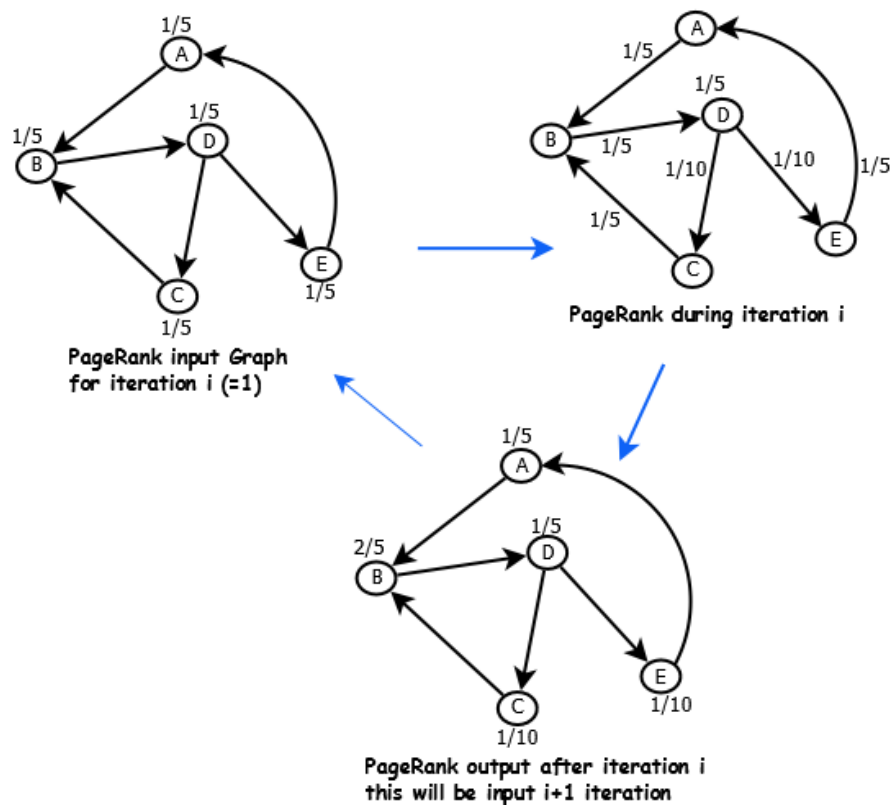


Figure 1: Small Graph

1. Dead-ends: dead-ends are the nodes in the graph that has incoming links from other nodes but no outgoing links from them. These types of nodes accumulate pageranks in every iteration but never distribute any pagerank to any other nodes in the graph. In long term, these nodes ends up getting all the pagerank and higher “importance” in the graph.
2. Spider traps: spider traps are group of nodes that have incoming links from the nodes outside of the group but outgoing links in within group only, and as result these nodes accumulate pageranks in each iteration and becomes “important” in long run.

To resolve these two issues, we consider small probability that “random surfer” can go to any node from a given node, (in terms webpage graph, this would means that surfer didnt click any of the hyperlinks on the page but went to random page). This would take us out from both situations- dead-ends and spider-traps.

Formally, PageRank P of a node n is defined as follows:

$$P(n) = \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

where $L(n)$ is the set of nodes that links to node n , in other words node n has incoming links from nodes in $L(n)$. $C(m)$ is number of outgoing links (out-degree) of node m .

Now, including small chance that a “random surfer” can jump to any page and not following links on the page, we get following formulation:

$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

where, α is random jump factor and $|G|$ is number of nodes in the graph. Since $1 - \alpha$ is probability that a random surfer will follow the links on the page, we would need to add both the probabilities.

1.2 MapReduce

MapReduce [2] is a programming model to process large data sets with a parallel, distributed algorithm on a cluster of commodity servers. It consist of three phases, Map phase, Shuffle phase and Reduce phase.

- Map Phase: Input data is split into chunks (splits) and each chunk is processed in parallel, possibly on different servers on the network.
- Shuffle Phase: Output of Map phase is sorted and partitioned. Each partition is send to Reduce phase. Reduce phase algorithm assumes that input data is sorted.
- Reduce Phase: Data is aggregated to desired output. Multiple Reducers runs each partition in parallel.

PageRank algorithm can be described in MapReduce model as following:

MAPPER

```

INPUT: nodeid n, node N
PROCESS:
    p = N.PAGERANK / N.OUTDEGREE
    EMIT(n, N)          # Pass Graph Structure
    for all nodes m in N.OUTLINKS do
        EMIT(m, p)      # Pass PageRank to neighbors

REDUCER                                # Input to Reducer is sorted by nodeid
INPUT: nodeid m, [p1, p2, ....]
PROCESS:
    M = nil
    for all p in [p1, p2, ....] do
        if IsNode(p) then
            M = p        # Recover Graph Structure
        else
            s = s + p     # Sum PageRank contributions

    M.PAGERANK = s       # Update PageRank

    EMIT(m, M)

```

1.3 Graph Representation

Graph can be represented in two way, Matrix and Adjacency list. Matrix representation of Graph in iteration 1 in figure 1 is following:

	A	B	C	D	E
A	0	0	0	0	1/5
B	1/5	0	1/5	0	0
C	0	0	0	1/10	0
D	0	1/5	0	0	0
E	0	0	0	1/10	0

Where each cell contains PageRank a node going to send to its neighbors and 0 is there is no outgoing link exists. For example, column D has 1/10 in C and E and 0 every where else. This representation is good for mathematical computation, however it takes lots of memory specially if graph is sparse (it has more 0s than non-zero). For storing and processing such Graphs adjacency list representation, like following, is useful.

```

A -> [B]
B -> [D]
C -> [B]
D -> [D,E]
E -> [A]

```

In this implementation, we use adjacency list representation.

1.4 Hadoop

Hadoop is an open source implementation of MapReduce framework. It consists of two main components:

1. Hadoop Distributed Filesystem (HDFS)
2. Data Processing Framework (MapReduce)

In this project we have used Hadoop 2.6, which has HDFS [3] as distributed filesystem implementation and YARN [4] as cluster manager, MapReduce runs as an application over YARN.

1.4.1 Hadoop Distributed FileSystem (HDFS)

Hadoop was designed to process very large datasets, in size of multiple gigabytes and petabytes. In 2012, Facebook claimed that their Hadoop cluster stores 100 PB data and growing by half PB everyday [5]. Such datasets requires distributed filesystem, that can store data on multiple nodes in the cluster and provide tolerance to node and data failures. To addresses these HDFS was designed with following objectives:

1. Very large files: files that are hundreds of megabytes, gigabytes, or terabytes in size.
2. Streaming data access: It assumes most common data processing pattern, that write-once, read-many times. A dataset is typically copied from a source, and the various analysis creates new files and not update the existing files.
3. Commodity hardware: Doesnot require specialized hardware. Runs on commonly available servers (intel based x86) and disks. It doesn't need specialized storage system e.g. RAID, SAN etc.

1.4.1.1 The FileSystem

- Block Size: Disk's block size is the minimum amount of data it can read and write. Filesystems built for single disk uses this block size (or multiple of it) to read and write data. Filesystem block size are few kilobytes (e.g. ext4 has 4KB) while disk block size is few bytes, normally 512 bytes. HDFS has very large block size, 64MB by default, like single disk filesystems data is broken into chunk of blocks and stored in the disk, unlike single disk filesystem, in HDFS, if a chunk of block is smaller the filesystem block size it doesnt take full block worth of storage. Also, in HDFS block size can be changed per file.

- **NameNode and DataNode:** An HDFS cluster has two types of background process (daemon) operating in master-worker fashion. One NameNode (master) and many DataNodes (workers). The namenode manages filesystem metadata and namespaces, this information is persisted on the local disk in the form of two files- the namespace image (fsimage) and edit logs. Namenode also knows where all the datanodes are located (IP addresses), however it doesn't persist this information as whenever a datanode is started it let namenode knows about its location. Datanodes stores and retrieve blocks when they are asked by client or namenode, also they report to namenode about the data blocks they are storing. Figure 2 describes components of HDFS.

Client accesses to the HDFS is provided by interface similar to a Portable Operating System Interface (POSIX), and user code doesn't need to know about details of namenode and datanodes.

Namenode is critical component of HDFS as without it data stored in datanodes are just block of bytes. If namenode's metadata is lost whole filesystem is lost, therefore hadoop provides multiple strategies to avoid this single-point-of-failure.

- Configure namenode to store its metadata on multiple disk, including remote NFS mount.
- Run a Secondary namenode, despite of its name Secondary namenode is not backup namenode, its main role is to periodically merge namespace image (fsimage) with edit log to prevent editlog becoming too big. Usually Secondary namenode run on the separate physical machine because its tasks required plenty of CPU and as much memory as namenode. It keeps a copy of merged namespace which can be used to restart the namenode, however since secondary namenode lags in the merging editlog with fsimage, it won't be full recovery.

1.4.1.2 File Read

Figure 3 describes flow of reading a file from HDFS.

1. Client opens the file it want to read using `open()` on `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem`.
2. `DistributedFileSystem` calls namenode, using RPC, to determine the location of the first few blocks in the file. For each block, namenode returns the addresses of the datanodes that have copy of that block. datanodes are sorted according to their proximity to the client in topology of the cluster's network.
3. `FSDatInputStream` instance is returned by the `DistributedFileSystem` to the client, which manages the datanode and namenode I/O. Client call `read()` on `FSDatInputStream` instance.
4. Which connects client to close by datanodes to read data blocks from.

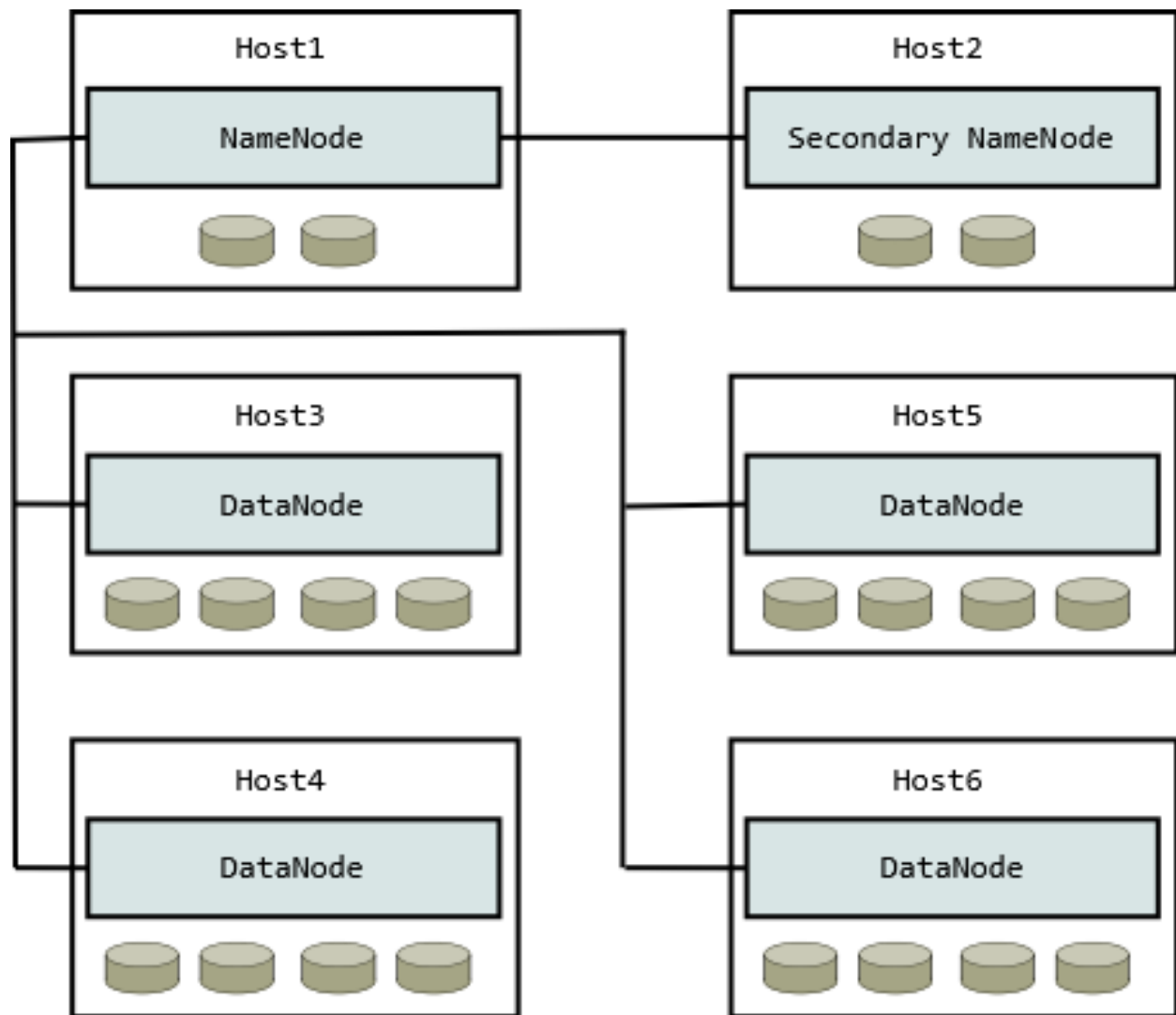


Figure 2: HDFS Architecture, Source [6]

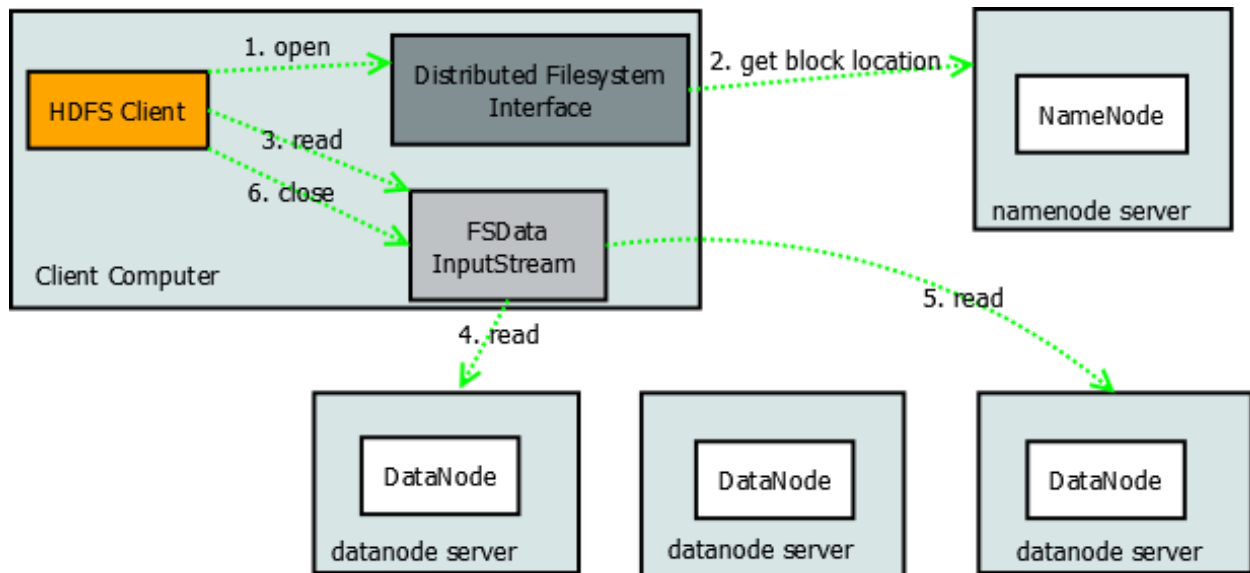


Figure 3: HDFS File Read, Source [7]

5. When end of block reach `FSDataInputStream` closes the connection to `DataNode` and find the closest datanode for the next block. From client's point view, it reads stream of data from the file and not aware of datanodes locations and opening/closing connections to the datanodes.
6. When client is done reading it calls `close()` which disconnectes it from namenode.

If there is an error reading data from a datanode, since data is replcated in multiple datanodes (by default 3), `FSDataInputStream` finds another datanode storing same data block.

1.4.1.3 File Write

Figure 4 describes flow of reading a file from HDFS.

1. Client create a file by calling `create()` on `DistributedFileSystem`
2. `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it. Namenode checks if file is not already exists and if client has permission to create a file. `DistributedFileSystem` returns an `FSDataOutputStream` to the client. `FSDataOutputStream` handles communication with datanodes and namenode.
3. As client writes data on to `FSDataOutputStream`, it splits data into packets, which it writes onto a internal queue, called data queue. The data queue is consumed by the `DataStreamer`, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas.
4. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the

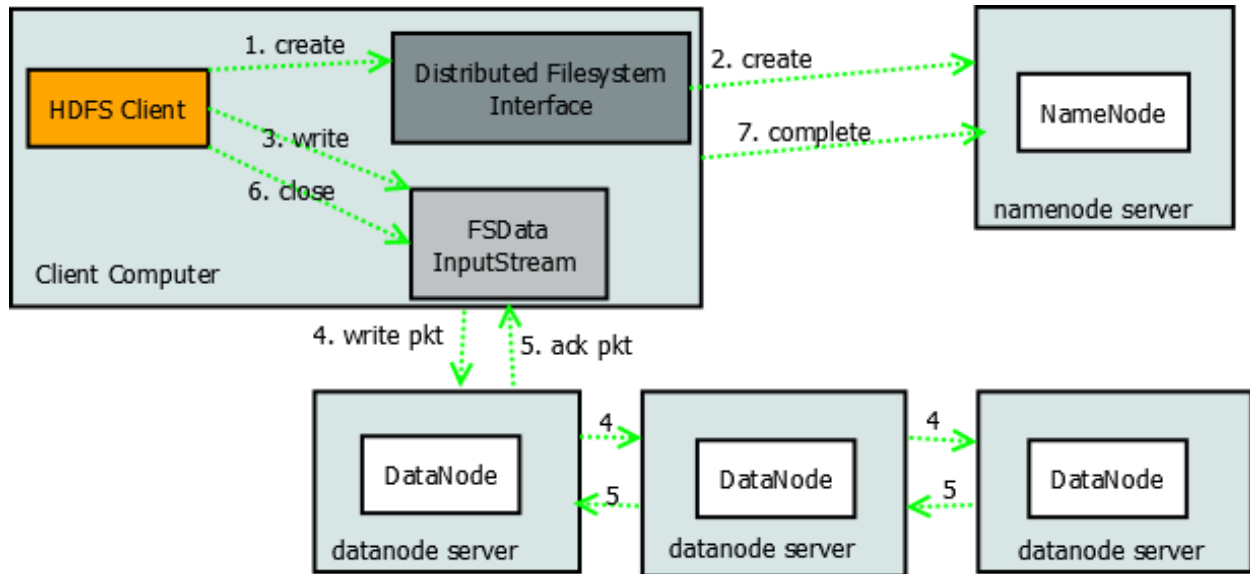


Figure 4: HDFS File Write, Source [7]

second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.

5. `FSOutputStream` also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline.
6. Client finished writing data and calls `close()` on `FSOutputStream`. This action flushes all remaining packets to the data pipeline and waits for the ack.
7. After all the remaining ack, `FSOutputStream` connects to namenode to signal that file is complete. Namenode already knows which blocks files are made of so it only waits for blocks to minimally replicated before returning successfully.

1.4.1.4 Coherency Model

Coherency model of a filesystem, describes the data visibility of reads and writes for a file. In HDFS, files are visible as soon as they are created, however data might not be visible even after file stream is flushed. Once more than a block of data is written it will be visible to the reader, and so on to the subsequent blocks. Only the block that is currently being written is not visible to other readers. HDFS provides method similar `fsync` system call in POSIX that commits the data. After a successful return from `sync` HDFS guarantees that all the data written to that point will be visible in all the datanodes in the datanode pipeline. However, `sync` call has performance overhead and in absence of this call applications should be prepare to recover from a block of data loss in case of failure.

1.4.2 YARN

YARN [4] provides resource management for Hadoop cluster. Currently it only supports memory resources, in future other resources like CPU, disk access and network bandwidth can be added. YARN has following background process (daemon):

- ResourceManager: This is one per cluster, and responsible for scheduling of all the available resource in the cluster.
- NodeManager: one per worker node, and responsible for scheduling of all the resources available in that node, it takes directions from Resource Manager.
- Application Master: One per User's submitted job (User Application running in the cluster)
- Container: User submitted Applications (Jobs) runs in Containers. Containers are limited by the resources requirement specified at the time of Job submission.

1.4.2.1 Hadoop Job Flow

Sequence of events from when User submits a job to when job is finished is described in figure 5.

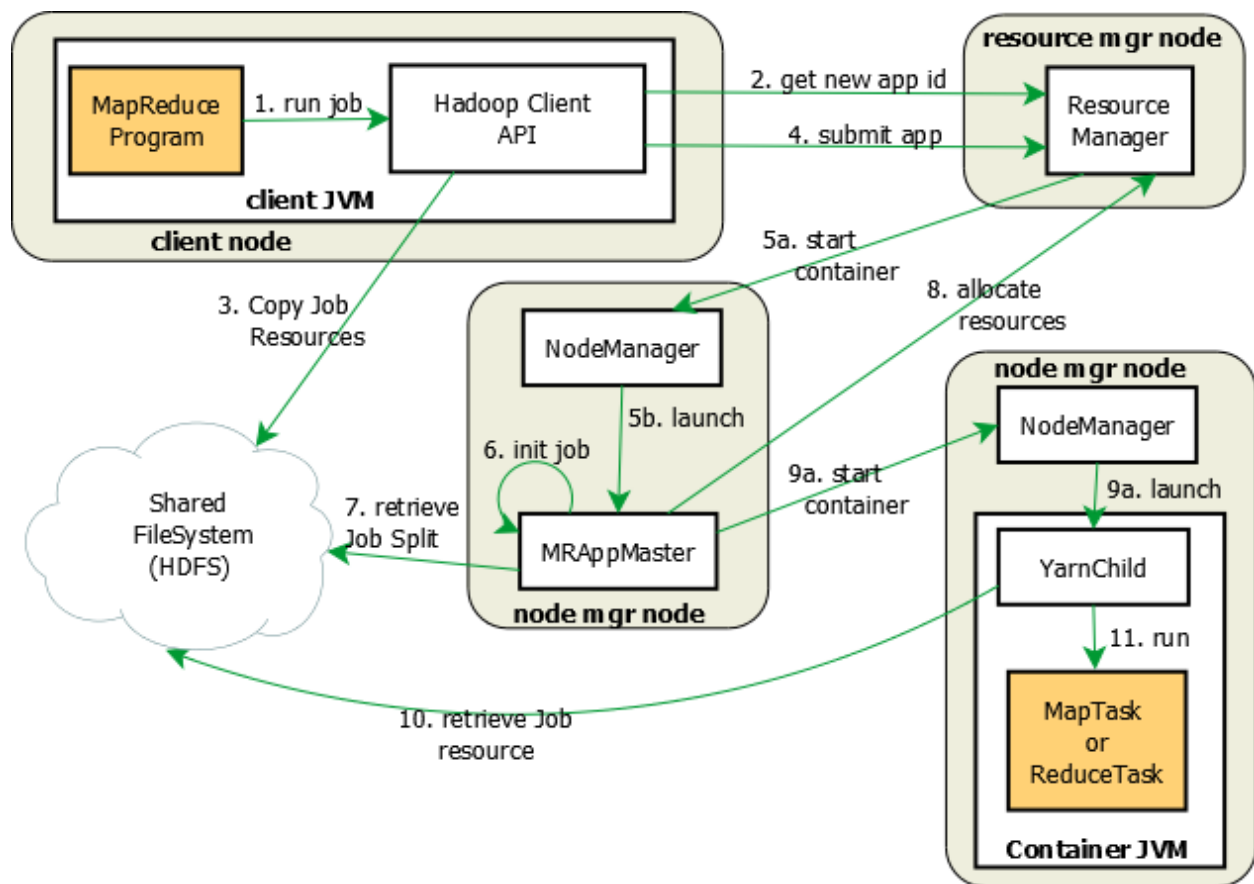


Figure 5: YARN MapReduce Job Flow, Source [7]

1. Client submits a “MapReduce Program” (Application or Job) to the Resource Manager using Hadoop Client API. Application specification includes input files and output directory, config files, and memory requirements.
2. Hadoop Client API communicates with Resource Manager to get new “ApplicationId” via `ApplicationClientProtocol#getNewApplication`.
3. Hadoop Client API copies “Application” resources- code (jar), config files, to HDFS.
4. Hadoop Client API then submits the “Application” to be run via `ApplicationClientProtocol#submit`.
5. Resource Manager hands off request to the scheduler, which finds a NodeManager node and ask it to launch a Container (5a) according to resource constraint specified. NodeManager launches the Container and lunch (MapReduce)ApplicationMaster (MRAppMaster) (5b).
6. MRAppMaster initialize the job by creating number of bookkeeping objects to keep track of job.
7. MRAppMaster retrieve input split from HDFS that was computed in the client. Then for each split it creates MapTask, that is if there are n split, it run n MapTasks as well as ReduceTasks specified by the Application (using `mapreduce.job.reduces` property).
8. MRAppMaster request for containers from ResourceManager to run Map(Reduce)Tasks. Request includes information about each map task’s data locality (hosts and corresponding racks that the input split resides in), ResourceManager scheduler uses this information to allocate containers that run MapTask close to the data (input split) its going to operate on, if it cant find enough resources on the same host as input split resides on, it prefers same rack.
9. Once ResourceManager assigns containers for tasks, MRAppMaster starts those containers by connecting to NodeManagers of respective hosts. NodeManager launches containers on their hosts (9a)
10. Each container starts with a class called YarnChild which retrieve any config files code needs by this MapTask from HDFS.
11. YarnChild run the MapTask in the same JVM.

When Application is runing they report their status and progress to MRAppMaster which aggregates it and send back to Client API waiting for Application to be completed. Job status and progress can also be queried from MRAppMaster at run time using YARN API.

2 Design

2.1 Wikipedia Dataset

Wikipedia english pages are publically available as one xml file `enwiki-latest-pages-articles.xml.bz2`¹. Size of the compressed file is about 12GB, while uncompressed size is 49GB. This one xml file contains all the english pages of Wikipedia in following format:

```
<mediawiki>
```

¹<http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2>

```

<siteinfo>
</siteinfo>
<page>
  <title>University of New Haven</title>
  <ns>0</ns>
  <id>497586</id>
  <revision>
    <text xml:space="preserve" bytes="28937">
      </text>
    </revision>
  </page>
</mediawiki>

```

to see the complete format we can use `Special:Export` link, for example to see xml page for University of New Haven Wikipedia link would be `Special:Export/University_of_New_Haven`²

2.1.1 Parsing Wikipedia Page XML

We want to parse this xml, (not using a real xml parser because that would take too long) however treating this as a text. Article id that we will use as nodeid is in `<id>` tag, and outgoing links from this article are in `<text>` tag, we also need article title which is in `<title>` tag. For a given article we use its id as source node in the graph and all id of all the outgoing links as destination node. We didnt consider external links to web. Internal Links are enclosed in double square brackets, for example, `[[Private University|Private]]`, `[[West Haven, Connecticut|West Haven]]`. Part after `|` is the text that is displayed for this link. Using this data we build adjacency graph structure as following:

```

id1  -> [id2, id3...]
id2  -> [id1, id4...]
....

```

We also build a mapping between page id and its title, so that we find title of that article for a given id, we would need it to generate list of articles.

2.1.2 Types of Articles

- Disambiguation: When one word represents multiple pages. for example, page for Mercury³ points to page for

– Mercury (element), a metallic chemical element

²http://en.wikipedia.org/wiki/Special:Export/University_of_New_Haven

³<http://en.wikipedia.org/wiki/Mercury>

- Mercury (planet), the planet closest to the Sun
- Mercury (mythology), a Roman god
- Redirect: Not a page itself, but redirects to other pages. For example, page UK⁴ redirects to United_Kingdom⁵. Text #REDIRECT is added to the redirect pages.
- Stub: Stubs are too short to be page, they might become page in future. Text {{stub}} is added to make a page stub.

During parsing of XML we count number of types of pages using a MapReduce job as described later. Following is the counts we have found:

ARTICLE	=	4,855,471
DISAMBIGUATION	=	132,156
EMPTY	=	1,379
OTHER	=	3,548,186
REDIRECT	=	6,901,158
STUB	=	1,905,746
TOTAL	=	15,438,350

Wikipedia also provides these counts on Wikipedia:Size_of_Wikipedia⁶. After building the graph we have found following counts:

EDGES	=	148,136,628
TOTAL_VERTICES	=	15,438,342
VERTICES_WITH_OUTLINKS	=	12,779,080

2.2 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2)⁷ provides resizable compute capacity in the cloud. Users only pays for the capacity that they used. EC2 uses Xen⁸ hypervisor to provide virtual instances of servers.

2.2.1 Types of Instances

Instances are servers that we use to build cluster. Each instance types is a specification that defines the memory, CPU, storage capacity, and hourly cost for that instance. Computing unit is measured in vCPU, according to EC2 documentation⁹, “Each vCPU is a hyperthread

⁴<http://en.wikipedia.org/wiki/UK>

⁵http://en.wikipedia.org/wiki/United_Kingdom

⁶http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

⁷<http://aws.amazon.com/ec2/>

⁸<http://www.xenproject.org/>

⁹<http://aws.amazon.com/ec2/instance-types>

of an Intel Xeon core for M3, C4, C3, R3, HS1, G2, I2, and D2.”, however some tests [8] has shown vCPU represents half of a physical core, so if we are looking for 8-core server we would need an instance with 16 vCPUs.

2.2.1.1 General Purpose

- T2: Most basic level instances, good for testing and prototypes. There are three types-
 1. t2.micro: 1 vCPU, 1GB RAM
 2. t2.small: 1 vCPU, 2GB RAM
 3. t2.medium: 1 vCPU, 4GB RAM All T2 storage are EBS only. EBS storage type is described later.
- M4: Better than T2, still only EBS storage and no instance storage. Following are M4 instance types-
 1. m4.large: 2 vCPU, 8GB RAM
 2. m4.xlarge: 4 vCPU, 16GB RAM
 3. m4.2xlarge: 8 vCPU, 32GB RAM
 4. m4.4xlarge: 16 vCPU, 64GB RAM
 5. m4.10xlarge: 40 vCPU, 160GB RAM
- M3: M3 provides local instance storage, SSD disks.
 1. m3.medium: 1 vCPU, 3.75GB RAM, 1x4GB SSD
 2. m3.large: 2 vCPU, 7.5GB RAM, 1x32GB SSD
 3. m3.xlarge: 4 vCPU, 15GB RAM, 2x40GB SSD
 4. m3.2xlarge: 8 vCPU, 30GB RAM, 2x80GB SSD

As for this project, we are using `m3.large` and `m3.xlarge` instances, We won't describe rest of the instance types in detail, and briefly mention it for completeness. For details see documentation¹⁰.

2.2.1.2 Compute Optimized

Provides high performance CPUs, useful in CPU bound computations.

- C4: c4.large, c4.xlarge, c4.2xlarge, c4.4xlarge, c4.8xlarge
- C3: c3.large, c3.xlarge, c3.2xlarge, c3.4xlarge, c3.8xlarge

2.2.1.3 Memory Optimized

Optimized for memory-intensive applications.

- R3: r3.large, r3.xlarge, r3.2xlarge, r3.4xlarge, r3.8xlarge

¹⁰<http://aws.amazon.com/ec2/instance-types>

2.2.1.4 GPU

Optimized for graphics and general purpose GPU computing, e.g. Deep learning and Video encoding.

- G2: g2.2xlarge, g2.8xlarge

2.2.1.5 Storage Optimized:

Optimized for high I/O bound computation.

- I2: i2.xlarge, i2.2xlarge, i2.4xlarge, i2.8xlarge
- D2: d2.xlarge, d2.2xlarge, d2.4xlarge, d2.8xlarge

2.2.2 Types of Storage

Amazon cloud services provides three types of storage options:

2.2.2.1 Elastic Block Store (EBS)

Block level storage volumes for use with EC2 instances. These are off-instance storage, that data stored here survive the instance restart. However this is similar to NFS mount and requires network communication to access data.

2.2.2.2 Simple Storage Service (S3)

Low cost redundant storage infrastructure, data is also available over internet. This has low IO performance compare to EBS, it can be used for application data, however its most useful as backup storage.

2.2.2.3 Instance Storage

Instance storage are local disk attached to the physical server, no network overhead to access data and high IO performance. Instance storage are SSD volumes that can be mounted to the EC2 instances.

2.2.3 Which instance type to choose for Hadoop cluster

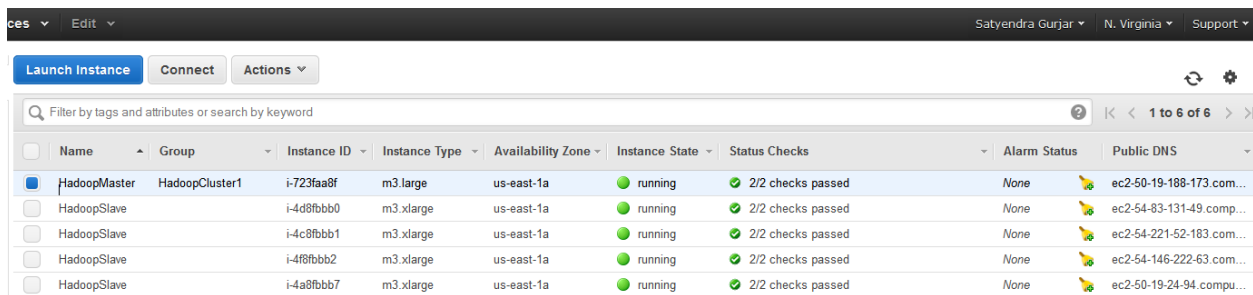
Hadoop split a Job in multiple Tasks and run each tasks in parallel. Data locality is processing data where it is stored, in HDFS data blocks are stored on DataNodes (workers), while NameNode (master) stores metadata. YARN provides MapReduce processing, where ResourceManager runs on master nodes and NodeManager runs on worker nodes. We run NodeManager and DataNodes on the same servers in our cluster. Hadoop goes great length to ensure data locality, that is a Task running on a Worker gets its data stored on the same

server. If we use EBS and/or S3 storage for Workers, there won't be any data locality as EBS and S3 storage are not attached to physical server and data is read/write over the network. Data stored in the Instance store does not survive server restart. Therefore, we use EBS storage to store configuration, code and libraries; and Instance storage to store data that needs to be processed. M3 types of instance provides general purpose computation efficiency with local SSD's, we use m3.xlarge instances for Worker nodes and m3.large instance for Master node.

3 Implementation

3.1 Hadoop Cluster

3.1.1 Creating Server Instances



	Name	Group	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
<input checked="" type="checkbox"/>	HadoopMaster	HadoopCluster1	i-723faa8f	m3.large	us-east-1a	running	2/2 checks passed	None	ec2-50-19-188-173.com...
<input type="checkbox"/>	HadoopSlave		i-4d8fbbb0	m3.xlarge	us-east-1a	running	2/2 checks passed	None	ec2-54-83-131-49.comp...
<input type="checkbox"/>	HadoopSlave		i-4c8fbbb1	m3.xlarge	us-east-1a	running	2/2 checks passed	None	ec2-54-221-52-183.com...
<input type="checkbox"/>	HadoopSlave		i-4f8fbbb2	m3.xlarge	us-east-1a	running	2/2 checks passed	None	ec2-54-146-222-63.com...
<input type="checkbox"/>	HadoopSlave		i-4a8fbbb7	m3.xlarge	us-east-1a	running	2/2 checks passed	None	ec2-50-19-24-94.compu...

Figure 6: EC2 Instances

- To build Hadoop cluster we choose one m3.large type instance to work as Master and four m3.xlarge instances to work as Workers. Figure 6 shows one instance with name **HadoopMaster** and four instances of type **HadoopSlave**.
- As discussed earlier, Master, m3.large, instance has 2 vCPU, 7.5GB RAM and 32GB SSD. Worker instances, m3.xlarge, have 4vCPU, 15GB RAM and two 40GB SSD. Both Master and Worker instances have 100GB EBS storage, that is used as root device, that OS is installed here and server boot from this device.
- On all instances, we create Red Hat Enterprise Linux 7.1 Amazon Machine Images (AMI), specifically its RHEL-7.1_HVM_GA-20150225-x86_64-1-Hourly2-GP2 (ami-12663b7a).
- Now we need to mount the additional SSD devices on each instance. `lsblk` command shows available disks on the server. For example, running `lsblk` on one worker node results in following

```
ec2-user@ip-10-233-128-100 ~$ lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
xvda	202:0	0	100G	0	disk	
xvda1	202:1	0	1M	0	part	
xvda2	202:2	0	100G	0	part	/
xvdb	202:16	0	37.5G	0	disk	/mnt
xvdc	202:32	0	37.5G	0	disk	/mnt2

Where `xvdb` and `xvdc` are two SSD, in this example its already mounted on directories `/mnt` and `/mnt2`. Process of mounting is following:

1. Check if device has a filesystem on it. If the output from this command shows simply data for the device, then there is no file system on the device and you need to create one. In following example it shows device has `ext3` filesystem and we don't need to create a filesystem.

```
$sudo file -s /dev/xvdc
/dev/xvdc: Linux rev 1.0 ext3 filesystem data, \
          UUID=70a35edb-68d5-4132-864f-881822ae16f6 (large files)
```

2. If we were to create filesystem, we could use `mkfs` command. For example, following command will create a `ext4` filesystem-

```
$sudo mkfs -t ext4 /dev/xvdb
```

3. Once filesystem is created we can mount it using `mount` command, following mounts `/dev/xvdb` device on `/mnt` directory:

```
$sudo mount /dev/xvdb /mnt
```

4. Now we can create `/etc/fstab` entry so that we dont need to do this ech time server restart. We add `/dev/xvdb` as follwing-

```
$ cat /etc/fstab
UUID=6785eb86-c596-4229-85fb-4d30c848c6e8 / xfs defaults 0 0
/dev/xvdb /mnt auto defaults,nofail 0 2
```

We have to follow same steps on each server (instance), except on worker we mount two devices and master has only one device. As woker runs DataNodes, multiple disks increases I/O efficiency as Tasks can read/write in parallel. We will see this when we configure Hadoop cluster.

3.1.1.1 Security Policies

Instances also have “Security groups” assigned to them. Security groups, with other things, determines what port are open on the instance they are assigned to. We confiure security group for instances in the Hadoop cluster as following:

Table 1: Inbound ports in Security Group

Type	Protocol	Port Range	Source
SSH	TCP	22	0.0.0.0/0
All TCP	TCP	0 — 65535	10.0.0.0/7
All UDP	UDP	0 — 65535	10.0.0.0/7

Port 22 is open for all. However all other TCP and UDP ports are open for subnet 10.0.0.0/7. When an instances is started it is assigned two IP addresses, one is public and other is private. Each time server restarts new public and private IPs are assigned. Our private IP addresses are in 10.0.0.0/7 subnet, and we want servers in the cluster to be able to communicate each other using RPC. Hadoop starts multiple processes that needs to talk to each other, for example, NameNode needs to talk to DataNode, ResourceManager needs to talk to NodeManager. This policy is too wide, we could have open only specific ports.

3.1.1.2 SSH keys and shell access

Amazon EC2 supports shell access to server (instances) using SSH keys. SSH key pairs can be created using EC2 console UI or via EC2 commandline-interface (CLI) or we can use third-party tools like `ssh-keygen` and import key pairs in EC2. Once the SSH key pair is created it needs to be associated with EC2 instance, which in turn add public key to `~/.ssh/authorized_keys`, this allows holder of private key part of the key pair to login into EC2 instance and have shell access. We also add private key to the Master node, so that we can have password-less SSH access to all Worker nodes, which would help us to start and stop Hadoop processes on all nodes from Master node.

3.1.2 Installation

We didnt install pre-packaged Java and Hadoop, that is we didnt use `yum install`.

- Java SDK 7: Downloaded 64-bit Linux binaries `jdk-7u79-linux-x64.tar.gz`¹¹ and extracted in `$HOME/jdk` directory.
- Apache Hadoop 2.6.0: Downloaded Hadoop 2.6 binaries `hadoop-2.6.0.tar.gz`¹² and extracted in `$HOME/hadoop-2.6.0` directory.

3.1.3 Configuration

Hadoop configuration resides in `$HADOOP_HOME/etc/hadoop`, where `$HADOOP_HOME` is directory of Hadoop installation, in our case its `$HOME/hadoop-2.6.0`. There are configuration files for each main component of Hadoop.

¹¹<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

¹²<http://hadoop.apache.org/releases.html>

3.1.3.1 Core

- `hadoop-env.sh`: This is where we can set Hadoop-specific environment variables. In this file we only set `JAVA_HOME` and `HADOOP_HEAPSIZE` variables. `JAVA_HOME` points to Java installation. `HADOOP_HEAPSIZE` defines maximum heap size for Java process runs tasks.

```
export JAVA_HOME=${HOME}/jdk
export HADOOP_HEAPSIZE=4096
```

- `core-site.xml`: This is main configuration file, that contain variables that affects all the Hadoop components. We set following variables:

- `fs.defaultFS=hdfs://10.142.242.104:8020`
 - * The name of the default file system. A URI whose scheme and authority determine the FileSystem implementation. The uri's scheme determines the config property (`fs.SCHEME.impl`) naming the FileSystem implementation class. The uri's authority is used to determine the host, port, etc. for a filesystem.
- `file.stream-buffer-size=65536`
 - * The size of buffer to stream files. The size of this buffer should probably be a multiple of hardware page size (4096 on Intel x86), and it determines how much data is buffered during read and write operations.
- `hadoop.tmp.dir=/tmp`
 - * A base for other temporary directories.

3.1.3.2 HDFS

- `hdfs-site.xml`: This file contains HDFS configuration.

- `dfs.replication=2`
 - * Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time. default: 3
- `dfs.hosts=/home/ec2-user/hadoop/etc/hadoop/slaves`
 - * A file that contains a list of hosts that are permitted to connect to the namenode. The full pathname of the file must be specified. If the value is empty, all hosts are permitted.
- `dfs.blocksize=268435456`
 - * The default block size for new files, in bytes. You can use the following suffix (case insensitive): k(kilo), m(mega), g(giga), t(tera), p(peta), e(exa) to specify the size (such as 128k, 512m, 1g, etc.), Or provide complete size in bytes (such as 134217728 for 128 MB). default: 134217728

- `dfs.namenode.name.dir=file:///mnt/dfs/name`
 - * Determines where on the local filesystem the DFS name node should store the name table(fsimage). If this is a comma-delimited list of directories then the name table is replicated in all of the directories, for redundancy. default: `file://${hadoop.tmp.dir}/dfs/name`
- `dfs.datanode.data.dir=file:///mnt/dfs/data,file:///mnt2/dfs/data`
 - * Determines where on the local filesystem an DFS data node should store its blocks. If this is a comma-delimited list of directories, then data will be stored in all named directories, typically on different devices. Directories that do not exist are ignored. default: `file://${hadoop.tmp.dir}/dfs/data`
- `slaves`: Contains IP addresses of DataNodes that are allowed to connect to NameNode. This file is used to set `dfs.hosts` variable.

```
$cat hadoop-2.6.0\etc\hadoop\slaves
10.231.171.25
10.79.142.248
10.142.179.84
10.79.142.41
```

3.1.3.3 MapReduce

- `mapred-site.xml`: Contains configuration for MapReduce jobs.
 - `mapreduce.framework.name=yarn`
 - * The runtime framework for executing MapReduce jobs. Can be one of local, classic or yarn. default is local
 - `mapred.child.java.opts=-Xmx1G`
 - * Java opts for the task processes. The following symbol, if present, will be interpolated: `???@` is replaced by current TaskID. Any other occurrences of '@' will go unchanged. For example, to enable verbose gc logging to a file named for the taskid in /tmp and to set the heap maximum to be a gigabyte, pass a 'value' of: `-Xmx1024m -verbose:gc -Xloggc:/tmp/???@.gc`
Usage of `-Djava.library.path` can cause programs to no longer function if hadoop native libraries are used. These values should instead be set as part of `LD_LIBRARY_PATH` in the map / reduce JVM env using the `mapreduce.map.env` and `mapreduce.reduce.env` config settings.
 - `mapreduce.map.memory.mb=2048`
 - * The amount of memory to request from the scheduler for each map task.
 - `mapreduce.reduce.memory.mb=2048`
 - * The amount of memory to request from the scheduler for each reduce task.
 - `mapreduce.task.io.sort.mb=256`

- * The total amount of buffer memory to use while sorting files, in megabytes. By default, gives each merge stream 1MB, which should minimize seeks.
- `mapreduce.task.io.sort.factor=64`
 - * The number of streams to merge at once while sorting files. This determines the number of open file handles.
- `mapreduce.job.maps=8`
 - * The default number of map tasks per job. Ignored when `mapreduce.jobtracker.address` is “local”.
- `mapreduce.job.reduces=4`
 - * The default number of reduce tasks per job. Typically set to 99% of the cluster’s reduce capacity, so that if a node fails the reduces can still be executed in a single wave. Ignored when `mapreduce.jobtracker.address` is “local”.

3.1.3.4 YARN

- `yarn-env.sh`: Contains environment variables for YARN processes. Similar to `hadoop-env.sh` we set `JAVA_HOME` variable here.

```
export JAVA_HOME=${HOME}/jdk
```

- `yarn-site.xml`

- `yarn.resourcemanager.hostname=10.142.242.104`
 - * The hostname of the Resource Manager.
- `yarn.resourcemanager.nodes.include-path=/home/ec2-user/hadoop/etc/hadoop/slaves`
 - * Path to file with nodes to include.
- `yarn.nodemanager.aux-services=mapreduce_shuffle`
 - * This variable tells NodeManagers that there will be an auxiliary service called `mapreduce_shuffle` that they need to implement. After we tell the NodeManagers to implement that service, we give it a class name as the means to implement that service, default implementation is `org.apache.hadoop.mapred.ShuffleHandler` and `org.apache.hadoop.mapreduce.task.reduce.Shuffle`. This particular configuration tells MapReduce how to do its shuffle. Because NodeManagers won’t shuffle data for a non-MapReduce job by default, we need to configure such a service for MapReduce.
- `yarn.resourcemanager.scheduler.class=org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler`
 - * The class to use as the resource scheduler.

After configuring files on the Master node, we need same configuration on all the Worker nodes. We use `rsync` command to keep configuration in sync on all the nodes.

3.1.4 Format HDFS

`hdfs-site.xml` configuration has 2 variables, `dfs.namenode.name.dir` and `dfs.datanode.data.dir` that define directories where NameNode stores its metadata and directories where DataNodes stores its datablocks, respectively. We have set, `dfs.namenode.name.dir=file:///mnt/dfs/name` and `dfs.datanode.data.dir=file:///mnt/dfs/data,file:///mnt2/dfs/data`. We have to make sure `dfs.namenode.name.dir` directory exists on the Namenode and the user that run Namenode daemon has permissions to read/write to this directory. Similarly, `dfs.datanode.data.dir` directories must exists on all the workers where DataNode runs, and the user that runs DataNode daemon has permissions to read/write to this directory. After this we are ready to format HDFS using following command-

```
~/hadoop-2.6.0/bin/hdfs -format
```

If this command return successfully, we have create HDFS filesystem and we are ready start HDFS processes.

3.1.5 Starting all Hadoop processes

3.1.5.1 Start HDFS daemons

- Start HDFS on all the servers in the cluster. This starts NameNode and SecondaryNameNode on the master and DataNode on all the worker servers.

```
$hadoop-2.6.0/sbin/start-dfs.sh
```

After HDFS is started we can use `hadoop-2.6.0/bin/hdfs dfsadmin -report` command to see the status of Name/DataNodes and filesystem capacity.

- Stop HDFS on all the servers in the cluster. This stops NameNode on the master and DataNode on all the worker servers.

```
$hadoop-2.6.0/sbin/stop-dfs.sh
```

3.1.5.2 Start YARN daemons

- Start YARN on all the servers in the cluster This starts ResourceManager on the master and NodeManager on all the worker servers.

```
$hadoop-2.6.0/sbin/start-yarn.sh
```

- Stop YARN on all the servers in the cluster This stops ResourceManager on the master and NodeManager on all the worker servers.

```
$hadoop-2.6.0/sbin/stop-yarn.sh
```

After both YARN and HDFS are started `NameNode` and `ResourceManager` should be running on the Master and `DataNode` and `NodeManager` should running on the Worker. We can verify this by using `ps` command or java specific `jps` command.

- On Master

```
$ jps
4619 NameNode
4754 ResourceManager
```

- On Worker

```
$jps
5499 DataNode
5612 NodeManager
```

3.2 Wikipedia Dataset in HDFS.

HDFS provides POSIX like filesystem interface. We are using HDFS commands to copy data to HDFS and to interact with HDFS. After downloading Wikipedia data dump for english pages, `enwiki-latest-pages-articles.xml.bz2`¹³, we decompressed it and copied on the HDFS using `hdfs dfs put` commands.

```
`$hdfs dfs -put enwiki-latest-pages-articles.xml /wiki/input/`
```

Where `enwiki-latest-pages-articles.xml` is file on local filesystem and `/wiki/input/` is directory in the HDFS filesystem.

3.3 Running Jobs

3.3.1 Job 1: Parsing Wikipedia XML to build Adjucency List Graph

Wikipedia XML parsing job is implemented in `edu.newhaven.sgurjar.wikipedia.ParseWikipediaDump` class that takes XML file as input and specify the output directory to write output files. It also specifiy InputFormatClass `edu.newhaven.sgurjar.wikipedia.XMLInputFormat` class and OutputFormatClass is `org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat`. The `SequenceFileOutputFormat` is compressed output file that can store java objects in binary form. We also set `MapSpeculativeExecution` to false. Hadoop provides speculative execution ability which means it runs multile map tasks for same input split in anicipation of some of them to fail or take too long, it use the output of the map task that is completed first, we don't speculative execution for this implementation. `ParseWikipediaDump` has two nested classes, `MyMapper` and `MyReducer` which implements map and reduce task functions, respectively.

¹³<http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2>

3.3.1.1 Computing Input splits of a MapReduce Job

Each Job that is submitted by the client specify a `InputFormat`, which is implementation of `org.apache.hadoop.mapreduce.InputFormat` class, and `InputFormat` has a `RecordReader` associated with which is implementation of `org.apache.hadoop.mapreduce.RecordReader` class. Job also specify input data files, either by specifying a directory in HDFS or by listing files that are stored in the HDFS (glob patterns can also be used). `InputFormat` is responsible to compute number of input split and their locations (locations of the datablocks) in the HDFS, which is IP addresses of `DataNodes` and datablock offsets. Input splits are computed at the client side (where the job was submitted) by calling `getSplits` method of `org.apache.hadoop.mapreduce.InputFormat` class, this returns a list of `InputSplit` object. Input splits are stored in the HDFS, input split only contains location of the split not actual data. When client call `submitApplication` on `ResourceManager`, `ResourceManager` hands off the request to scheduler. The scheduler allocate a `Container` and `ResourceManager` launches the application master's process here under the control of `NodeManager`. The application master for MapReduce jobs is a Java Application called `MRAppMaster` (short for "MapReduce Application Master"). `MRAppMaster` retrieves input splits for this job from HDFS and create map task for each split. The map task calls `createRecordReader` method on `InputFormat` for a given input split which returns a `RecordReader` object. Then map task runs a loop as following

```
while( RecordReader.nextKeyValue() ) {
    map( RecordReader.getCurrentKey(), RecordReader.getCurrentValue() )
}
```

Here, `RecordReader` is responsible to create keys and values from the input split, as long as `nextKeyValue` returns true, `getCurrentKey` returns next key and `getCurrentValue` returns next value. `map` function is implemented in User's code to process each key and value.

3.3.1.2 Parsing XML data

- For parsing Wikipedia, we used `InputFormat` class `edu.newhaven.sgurjar.wikipedia.XMLInputFormat` which is motivated from Apache Mahout¹⁴ project. `XMLInputFormat` computes splits using file's block size that is one split for each block, however it has custom `RecordReader` associated with it `edu.newhaven.sgurjar.wikipedia.XMLRecordReader` class. As discussed earlier map task initialized `RecordReader` by the input split (assigned to the map task) and calls `nextKeyValue` method on it to retrieve key and value from the split. `XMLRecordReader` reads split and returns the data between `<page>` and `</page>` tags as value and line offset as key. It is possible that beginning tag `<page>` exists in the split but ending tag `</page>` exists in the next split, `XMLRecordReader` takes care of that and ask filesystem to retrieve lines from next split until end tag `</page>` is found for a `<page>`, this might cause network communication, as only the split that map task for created for has to be data local.

¹⁴<http://github.com/apache/mahout>

- Now in `map` method of `MyMapper` class get text between `<page>` and `</page>` tags value. It parses the value, and find page id (between `<id>` and `</id>` tags) and all the outgoing links from this page (text between `[` and `]`). it outputs title of the source page and its id, and title of the all outgoing links and id of the source page. For example, if a page with title **A** and id 101 has outgoing links to pages **B**, **C** and **D**, mapper will output following key and value pairs-

key	value
(A,0)	101
(B,1)	101
(C,1)	101
(D,1)	101

Here key is the pair of title and an integer flag, where '0' means id is the source node in the edge and 1 means id is of **destination** node in the edge. When page with title **B** is processed by the mapper, it will out one key-value pair with `(B,0)` and B's id and other key-value pairs `(i,1)` and B's id, where **i** is an outgoing link from **B**.

- Since in Shuffle and Sort phase, all keys are sorted before each key and values pair sent to Reduce phase. And here values are list of values, as there could be multiple values for the key `(X,1)` where **X** is title of a page. The flag part also used in the sorting as second value, that is `(X,0)` comes first than `(X,1)` key. In `reduce` method we save the last id that came with `(X,0)` value and use it as source node in the adjacency list produced by all the values of `(X,1)` key. Reduer outputs page id as key and `edu.newhaven.sgurjar.wikipedia.Page` object as value. Output is stored as sequence file.

3.3.1.3 Counters

Hadoop Counters, are distributed version of global variable. We use counters to count number of nodes, edges etc. in graph. Counters can only be written without knowing their current value, that is incrementing them. Application code updating an counter can't use its current value to set new value as there are no guarantees that the current value read would be same as when a value being written. Counters are implemented in `org.apache.hadoop.mapreduce.Counters` class and has methods like `increment` and `get/setValue`.

3.3.2 Job 2: Computing PageRank of Wikipedia Pages Graph

Implementation of PageRank computation job motivated by Lin and Schatz's paper [9]. This job is implemented in `edu.newhaven.sgurjar.wikipedia.PageRankComputation` class. It takes following inputs- * **basepath**: Path in HDFS where input files for start iteration exists. * **nodes**: Number of Nodes in the graph * **start**: Start Iteration * **end**: End Iteration

There are two phases for each iteration, consists of two MapReduce jobs.

- Phase 1: This phase distributes PageRank mass of nodes along the edges.
- Phase 2: Being probability distribution, Pageranks of all nodes in the Graph must be one. We use this property to detect missing PageRank mass. PageRank could be missing if there are dead-ends in the graph, that is node that has no outgoing edge. This phase distributes the missing PageRank mass.

At the end of both phases, we get same graph structure that we started, with PageRanks of the nodes are updated. This will be used as the input to next iteration.

Before we get into details of these phases we need to describe two things, missing PageRank mass and floating point overflow.

3.3.2.1 Missing PageRank Mass

Since for the nodes that have no outgoing edges from it (dead-end), we do not distribute PageRanks. PageRanks of such graph will not converge and after each iteration sum of all PageRanks won't be one, which is the requirement for probability distribution. To deal with missing PageRank mass, in phase 1, last step of Reduce task (`cleanup` method) is to write total PageRank mass computed to file as "side-data". When the job in phase 1 is completed, client code (that submitted the job) reads all the files that were written by Reduce tasks and sum all the mass, that would give us all the mass distributed in Phase 1. Phase 1 returns this value which will be used by Phase 2 to distribute missing mass.

3.3.2.2 Floating point overflow

For large graphs, the probability of any particular node is often so small that it underflows standard floating point representation. For example, Wikipedia graph has about 15 million nodes, so initial probability of each node would be 15 millions over 1. To resolve this, we represent probabilities using their logarithms, and product of two probabilities is their sum. And addition of probabilities can be implemented with reasonable precision as following:

$$a \oplus b = \begin{cases} b + \log(1 + e^{a-b}) & a < b \\ a + \log(1 + e^{b-a}) & a \geq b \end{cases}$$

3.3.2.3 Iterations

We run multiple iterations of PageRank, and output of previous iteration goes as input to next iteration, we have setup directory structure of iterations as following-

- Phase 1
- Input directory: `basepath + "/iter" + start`
- Output directory: `basepath + "/iter" + start + "t"`
- Directory to write total mass computed in Reduce task: `basepath + "/iter" + start + "m"`

- Phase 2
- Input directory: `basepath + "/iter" + start + "t"`
- Output directory: `basepath + "/iter" + end`

Each phase is job, with its mapper and reducer functions. Phase 1, takes basedir, number of nodes and start/end iteration index as input, and it outputs same graph structure with updates PageRank of each node, it also returns total PageRank mass distributed. Phase2 takes start/end iteration and missing mass as input, and distribute missing to the graph structure. Following is the pseudo code of this job, at high level-

```
for(i=start; i < end; i++) {

    // distribute PageRank mass along outgoing edges
    totalMass = phase1(i, i+1, numOfNodes, basepath)

    // how much PageRank mass got lost due to deadends
    missing = 1.0f - (float) exp(mass) // 1 - e^mass

    // distribute missing mass
    phase2(i, i+1, missing, numOfNodes, basepath)
}
```

- Phase1, job has its own map and reduce functions. `phase1` submits job, which caused map and then reduce functions to be called. The `map` function take node `srcid` (key) and Page object (value) as input and outputs two types of key-value pairs- one is same key-value that it has received to maintain graph structure, and second is `id` (key) for each outgoing link's from `srcid` node, value is PageRank mass contributed by `srcid` node.

```
run() {
    // submit job and wait for completion
    job.waitForCompletion

    totalmass = NEGATIVE_INFINITY

    for (file in mass-output-dir) {
        totalmass += read total mass from file
    }

    return totalmass
}

map(srcnodeid, page) {
```

```

intermediateStructureNode.nodeId      = page.nodeId
intermediateStructureNode.type        = Structure
intermediateStructureNode.adjacencyList = page.adjacencyList

OUTPUT key=srcnodeid, value=intermediateStructureNode

mass = page.pageRank - log(page.adjacencyList.size)

for( nodeId in page.adjacencyList ) {
    intermediateMassNode.nodeId  = nodeId
    intermediateMassNode.type    = Mass
    intermediateMassNode.pageRank = mass
    OUTPUT key=nodeId, value=intermediateMassNode
}
}

reduce(nodeid, list-of-page) {
    newnode = Page()
    newnode.nodeId = nodeid
    newnode.type    = Complete

    mass = Float.NEGATIVE_INFINITY

    for(page in list-of-page) {
        if(page.type == Structure) { // rcvd Structure node
            newnode.adjacencyList = page.adjacencyList
        } else (page.type == Mass) { // rcvd Mass node
            mass += page.pageRank
        } else {
            ERROR
        }
    }
}

newnode.pageRank = mass

totalMass += mass

// output structure with update PageRank
OUTPUT key=nodeid, value=newnode
}

// after all reduce are done
cleanup() {
    write totalMass to file
}

```

- Phase2, is map only job, that means it has no reduce function. Its primary function is to distribute missing PageRank mass equally among all the nodes in the graph. Following is the pseudo code of Phase2 job, at high level-

3.3.3 Job 3: Finding Top K pages

After we are done with PageRank iterations, we need to find Top 100 pages in descending sorted order of their PageRank. This job is implemented in class `edu.newhaven.sgurjar.wikipedia.TopKPageRank`. This has mapper and reducer functions. We configure this job to have exactly one reducer task, which is important to find top K elements.

Input to the map function is `key=nodeid` and `value=page object`. Each map task holds a priority queue of fixed size (`size=K`, where `K` is the input variable). The priority queue holds only top K objects in sorted order where the page on the top of the queue has the highest PageRank. A new Page object is added (in the sorted order) to the queue, if its PageRank is higher than the page at the bottom of the queue. If the queue is full, the page at the bottom of the queue is kicked-out. So at all times each Map task can have at most K items in its queue in sorted order of their PageRanks. In the end of the map task, each mapper outputs this queue. Since there is only one Reducer, it gets `K * number_of_map_task` pages as input, again the Reducer has a priority queue that holds K pages in sorted order, retains only K pages. In the end of the reducer task, it outputs all the K pages.

3.3.4 Source Code

Source code of the project is available at [github](https://github.com/sgurjar/wikipedia-pagerank)¹⁵.

4 Conclusion

Following are the top 20 pages of Wikipedia English pages and their PageRanks.

Table 3: Top 25 Pages of Wikipedia English Pages

Docid	Title	PageRank
3434750	United States	-6.270592
31717	United Kingdom	-7.238336
5843419	France	-7.290263
32927	World War II	-7.3456545
11867	Germany	-7.373622
5042916	Canada	-7.4675508
273285	Race and ethnicity in the United States Census	-7.5366764
11039790	Animal	-7.5616207

¹⁵<http://github.com/sgurjar/wikipedia-pagerank>

Docid	Title	PageRank
10568	Association football	-7.5616217
14533	India	-7.585159
68253	List of sovereign states	-7.5929527
9316	England	-7.640645
30680	The New York Times	-7.665605
4689264	Australia	-7.708271
8569916	English language	-7.7511253
14532	Italy	-7.7722635
5405	China	-7.777225
15573	Japan	-7.7942243
25391	Russia	-7.8326564
17867	London	-7.866274
645042	New York City	-7.8673935
14653	Iran	-7.880345
4764461	World War I	-7.9937897
606848	Catholic Church	-8.001275
26667	Spain	-8.016219

Here, PageRanks are negative because we used logarithm of probabilities.

Reference

- [1] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pageRank citation ranking: Bringing order to the web.” 1999.
- [2] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (mSST), 2010 IEEE 26th symposium on*, 2010, pp. 1–10.
- [4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, and others, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th annual symposium on cloud computing*, 2013, p. 5.
- [5] “Under the hood: Scheduling mapReduce jobs more efficiently with corona,” <https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920>. Facebook, 2012.
- [6] E. Sammer, *Hadoop operations*. “O’Reilly Media, Inc.”, 2012, p. 10.
- [7] T. White, *Hadoop: The definitive guide*. “O’Reilly Media, Inc.”, 2012.
- [8] “Virtual CPUs with amazon web services,” 2014-06-24 <http://www.pythian.com/blog/virtual-cpus-with-amazon-web-services>. Pythian, 2014.

- [9] J. Lin and M. Schatz, “Design patterns for efficient graph algorithms in mapReduce,” in *Proceedings of the eighth workshop on mining and learning with graphs*, 2010, pp. 78–85.