# AN INTRODUCTION TO HADOOP ARCHITECTURE

## Overview

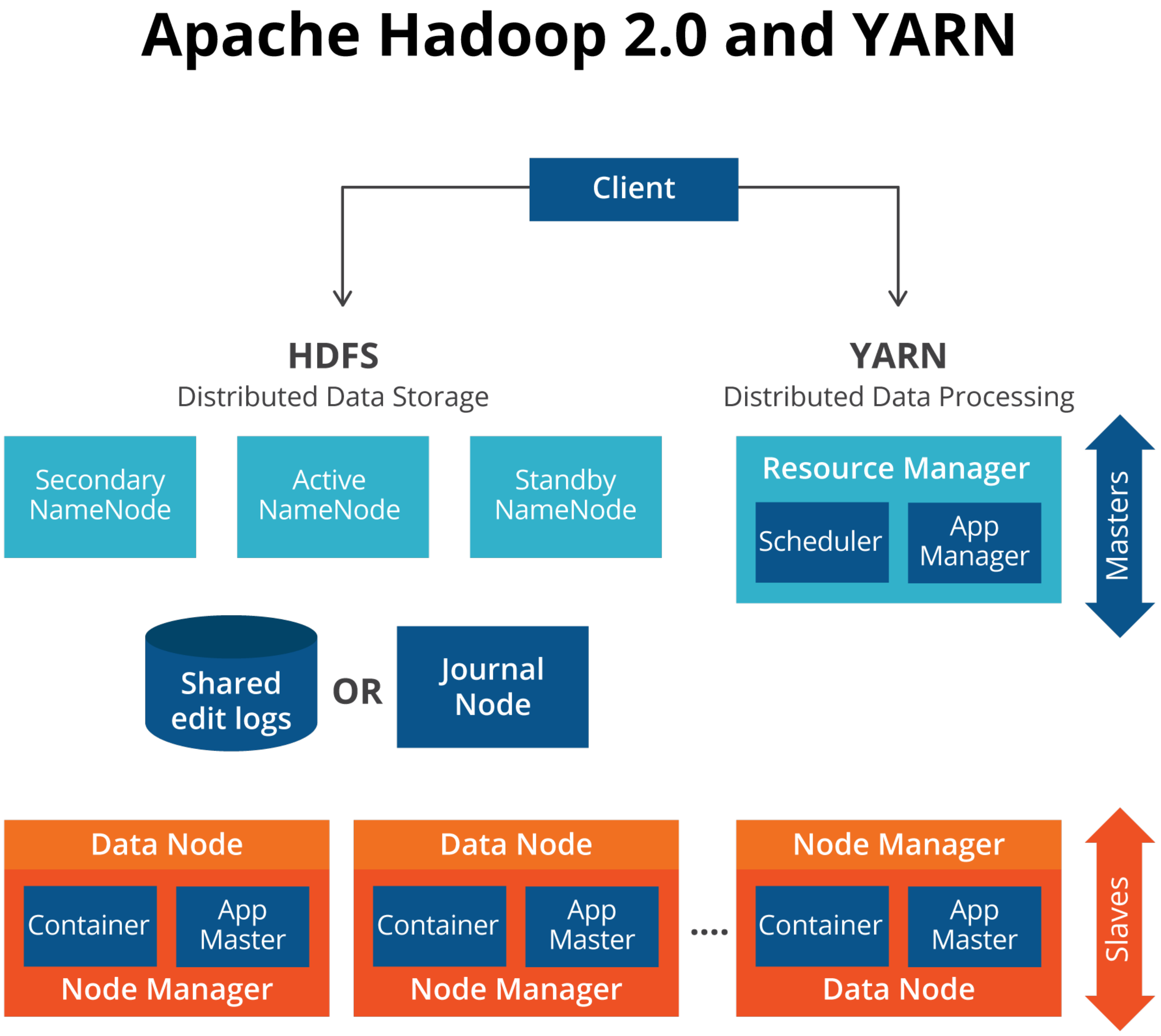
Hadoop is a distributed file system and batch processing system for running MapReduce jobs. That means it is designed to store data in local storage across a network of commodity machines, i.e., the same PCs used to run virtual machines in a data center. MapReduce is actually two programs. **Map** means to take items like a string from a csv file and run an operation over every line in the file, like to split it into a list of fields. Those become (key->value) pairs. **Reduce** groups these (key->value) pairs and runs an operation to, for example, concatenate them into one string or sum them like (key->sum).

Reduce can work on any operation that is associative. That is the principle of mathematics a + b = b + a. And it works on more than numbers and any programming object can implement addition, multiplication, and subtraction methods. (Division is not associative since 1 /2 <> 2 /1). The associate property is a requirement of a parallel processing system as Hadoop will get items out of order as it divides them into separate chunks to work on them.

Hadoop is designed to be fault tolerant. You tell it how many times to replication data. Then when a datanode crashes data is not lost.

Hadoop uses a master-slave architecture. The basic premise of its design is to **Bring the computing to the data instead of the data to the computing**. That makes sense. It stores data files that are too large to fit on one server across multiple servers. Then when is does Map and Reduce operations it further divides those and lets each server in the node do the computing. So each node is a computer and not just a disk drive with no computing ability.

# Architecture diagram



Here are the main components of Hadoop.

- **Namenode**—controls operation of the data jobs.
- **Datanode**—this writes data in blocks to local storage. And it replicates data blocks to other datanodes. DataNodes are also rack-aware. You would not want to replicate all your data to the same rack of servers as an outage there would cause you to loose all your data.
- **SecondaryNameNode**—this one take over if the primary Namenode goes offline.
- **JobTracker**—sends MapReduce jobs to nodes in the cluster.
- **TaskTracker**—accepts tasks from the Job Tracker.
- **Yarn**—runs the Yarn components ResourceManager and NodeManager. This is a resource

manager that can also run as a stand-alone component to provide other applications the ability to run in a distributed architecture. For example you can use Apache Spark with Yarn. You could also write your own program to use Yarn. But that is complicated.

- **Client Application**—this is whatever program you have written or some other client like Apache Pig. Apache Pig is an easy-to-use shell that takes SQL-like commands and translates them to Java MapReduce programs and runs them on Hadoop.
- **Application Master**—runs shell commands in a container as directed by Yarn.

# Cluster versus single node

When you first install Hadoop, such as to learn it, it runs in single node. But in production you would set it up to run in cluster node, meaning assign data nodes to run on different machines. The whole set of machines is called the **cluster**. A Hadoop cluster can scale immensely to store petabytes of data.

You can see the status of your cluster here

http://localhost:50070/dfshealth.html#tab-datanode.

# MapReduce example: Calculate e

Here we show a sample Java MapReduce program that we run against a Hadoop cluster. We will calculate the value of the mathematical constant e.

e is the sum of the infinite series Σ i = 0 to n (1 + 1 /n!). Which is:

```
e = 1 + (1 / (1 * 2)) + (1 / (1 * 2 * 3*) + (1 / (1 * 2* 3* 4)) + …
```

The further out we calculate n the closer we get to the true value of e.
So we list these 9 values of n in a text file in.txt:

```
in.txt
1
2
3
4
5
6
7
8
9
```

In the map operation we will create these key value pairs:

```
(x, 1!)
(x, 2!)
…
(x,9!)
```

We use the string x as the key for each key so that the reduce step will collapse those to one key (x,e).

Then we compute sum the running sum $\Sigma\ i = (1 / n!)$ and in the reduce step add 1 to the value to yield our approximation of e.

First we copy the file in.txt to the Hadoop file system. (You first need to format than and then create a directory).

```
hadoop fs -put /home/walker/Downloads/in.txt /data/in.txt
```

Below is the source code for CalculateE.java: In order to compile and run it you need to:

```
export  HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
hadoop com.sun.tools.javac.Main CalculateE.java
jar cf e.jar CalculateE*.class
hadoop jar e.jar CalculateE /data/in.txt /data/out.txt
```

Code:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class CalculateE {

public static class Factorial extends Mapper {



public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

int n = Integer.parseInt(value.toString());

double f = factorial(n);



double x = (1 / f);
DoubleWritable y = new DoubleWritable(x);
Text ky = new Text("x");
System.out.println("key=" + ky + " y= " + y);
context.write(ky, y);
}
}


public static int factorial(int f) {
return ((f == 0) ? 1 : f * factorial(f - 1));
}

public static class DoubleSumReducer extends Reducer {

                private DoubleWritable x = new DoubleWritable(0);

                public void reduce(Text key, Iterable values, Context
context)throws IOException, InterruptedException {
                        double sum = 0;
                        for (DoubleWritable val : values) {
                                sum += val.get();
                                System.out.println("val=" + val + " sum=" +
sum);
                        }

                        sum += 1;

                        System.out.println("key=" + key.toString() + " e = "
+ sum);
                        x.set(sum);
                        context.write(key, x);

                }
        }

        public static void main(String[] args) throws Exception {
                Path inputPath = new Path(args);
            Path outputPath = new Path(args);
                Configuration conf = new Configuration();
                Job job = Job.getInstance(conf, "calculate e");
                FileInputFormat.setInputPaths(job, inputPath);
            FileOutputFormat.setOutputPath(job, outputPath);
                job.setJarByClass(CalculateE.class);
                job.setMapperClass(Factorial.class);
                job.setReducerClass(DoubleSumReducer.class);
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(DoubleWritable.class);
                System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

When the program runs it saves the results in the folder /data/out.txt in the file part-r-00000. You can see the final results like this:

```
hadoop fs -ls /data/out.txt
hadoop fs -cat /data/out.txt/part-r-00000
 x 2.7182815255731922
```

That is pretty close to the actual value of e, 2.71828. To help debug the program as you write it you can look at the stdout log. Or you can turn on the job history tracking server and look at it with a browser: The stdout output includes many Hadoop messages including our debug printouts:

```
key=x y= 1.0
key=x y= 0.5
key=x y= 0.16666666666666666
key=x y= 0.041666666666666664
key=x y= 0.008333333333333333
key=x y= 0.001388888888888889
key=x y= 1.984126984126984E-4
key=x y= 2.48015873015873E-5
key=x y= 2.7557319223985893E-6


val=2.7557319223985893E-6 sum=2.7557319223985893E-6
val=2.48015873015873E-5 sum=2.755731922398589E-5
val=1.984126984126984E-4 sum=2.259700176366843E-4
val=0.001388888888888889 sum=0.0016148589065255732
val=0.008333333333333333 sum=0.009948192239858907
val=0.041666666666666664 sum=0.05161485890652557
val=0.16666666666666666 sum=0.21828152557319222
val=0.5 sum=0.7182815255731922
val=1.0 sum=1.7182815255731922
key=x e = 2.7182815255731922
```

## Hadoop storage mechanisms

Hadoop can store its data in multiple file formats, mainly so that it can work with different cloud vendors and products. Here are some:

**Hadoop**—these have the hdfs:// file prefix in their name.For example hdfs://masterServer:9000/folder/file.txt is a valid file name.

**S3**—Amazon S3.  The file prefix is s3n://

**file**—file:// causes Hadoop to use any other distributed file system.

Hadoop also supports Windows Azure Storage Blobs (WASB), MapR, FTP, and others.

# Writing to data files

Hadoop is not a database. That means there is no random access to data and you cannot insert rows into tables or lines in the middle of files. Instead Hadoop can only write and delete files, although you can truncate and append to them, but that is not commonly done.

# Hadoop as a platform for other products

There are plenty of systems that make Hadoop easier to use and to provide a SQL-like interface. For example, Pig, Hive, and HBase.

Many other products use Hadoop for part of their infrastructure too. This includes Pig, Hive, HBase, Phoenix, Spark, ZooKeeper, Cloudera Impala, Flume, Apache , Oozie, and Storm.

And then there are plenty of products that have written Hadoop connectors to let their product read and write data there, like ElasticSearch.

# Hadoop processes and properties

Type jps to see what processes are running. It should list NameNode and SecondaryNameNode on the master and the backup master. DataNodes run on the data nodes.

When you first install it there is no need to change any config. But there are many configuration options to tune it and set up a cluster. Here are a few.

| Process or property | Port commonly used | Config file | Name of property config item |
|---|---|---|---|
| NameNode | 50070 | hdfs-site.xml | dfs.namenode.http-address |
| SecondaryNameNode | 5090 | hdfs-site.xml | dfs.namenode.secondary.http-address |
| fs.defaultFS | 9000 | core-site.xml | The URL used for file access, like hdfs://master:9000. |
| dfs.namenode.name.dir | file:///usr/local/hadoop | | |
| dfs.datanode.name.dir | file:///usr/local/hadoop | | |
| dfs.replication | Set the replication value. 3 is commonly used. That means it will make 3 copie of each data it writes. | | |

# Hadoop analytics

Hadoop does not do analytics contrary to popular belief, meaning there is no clustering, linear regression, linear algebra, k-means clustering, decision trees, and other data science tools built into Hadoop. Instead it does the extract and transformation steps of analytics only.

For data science anaytics you need to use Spark ML (machine learning library) or scikit-learn for Python or Cran R for the R language. But only the first one runs on a distributed architecture. For the other two you would have to copy the input data to a single file on a single server. There is the Mahout analytics platform, but the authors of that say they are not going to develop it any more.