Hadoop & Spark

Dr. Villanes

Overview for today…

1. Introduction to MapReduce and HDFS (Hadoop)
2. Introduction to Spark
3. After the class → Hadoop + Spark Quiz (due at midnight – 1 hour to complete)

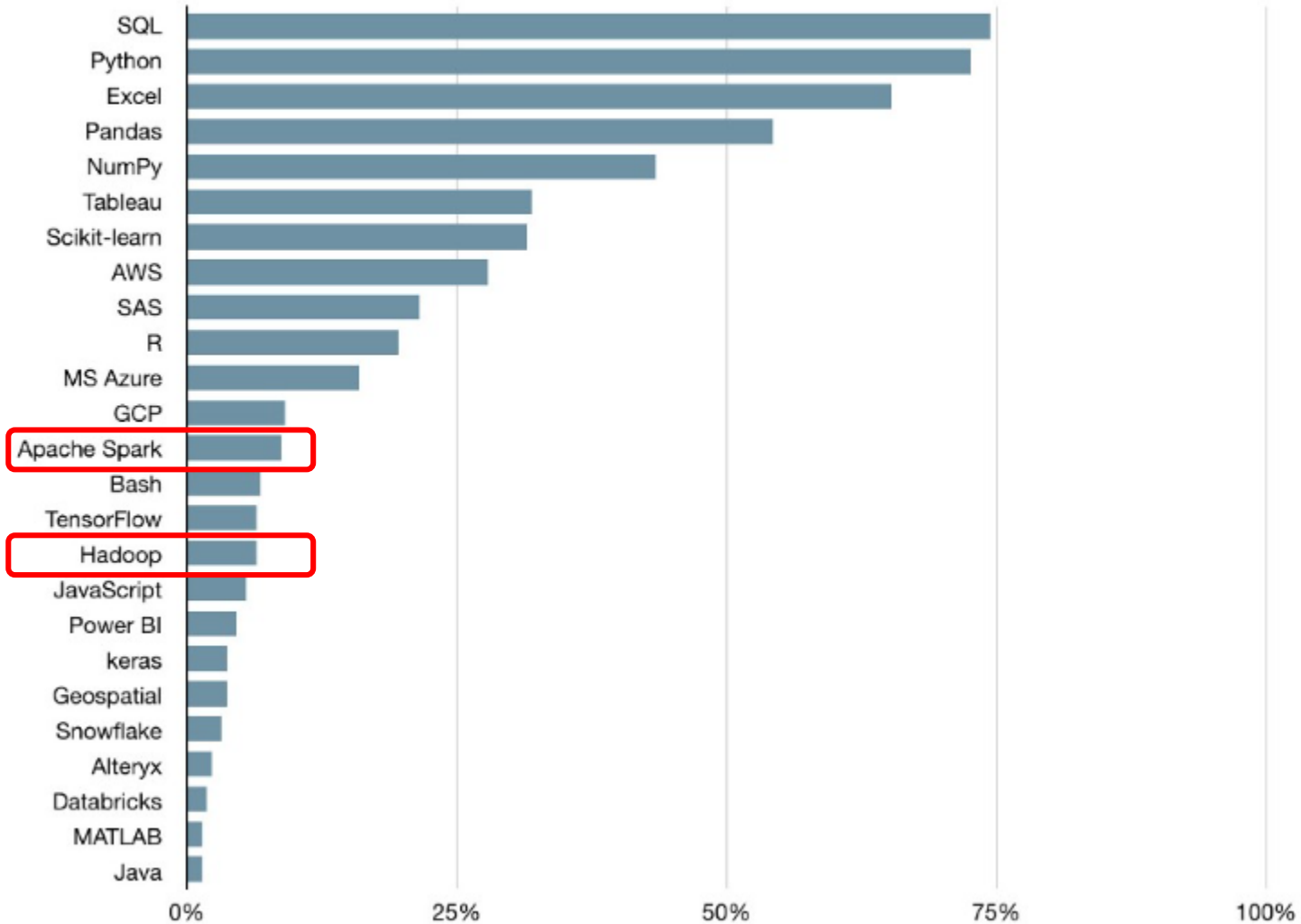
What we are going to see today?

Theory! Not a single line of code or hands-on lab

No coding? ☹ Next class!

Coding languages and tools you normally use when doing your work

(percent of respondents, n = 219)



Introduction to HDFS and MapReduce

The Motivation for Hadoop

- Velocity
- Variety
- Volume
- Data Has Value
- Two key problems to address:
  - How can we reliably store large amounts of data at a reasonable cost?
  - How can we analyze all the data we have stored?

What is Apache Hadoop ?

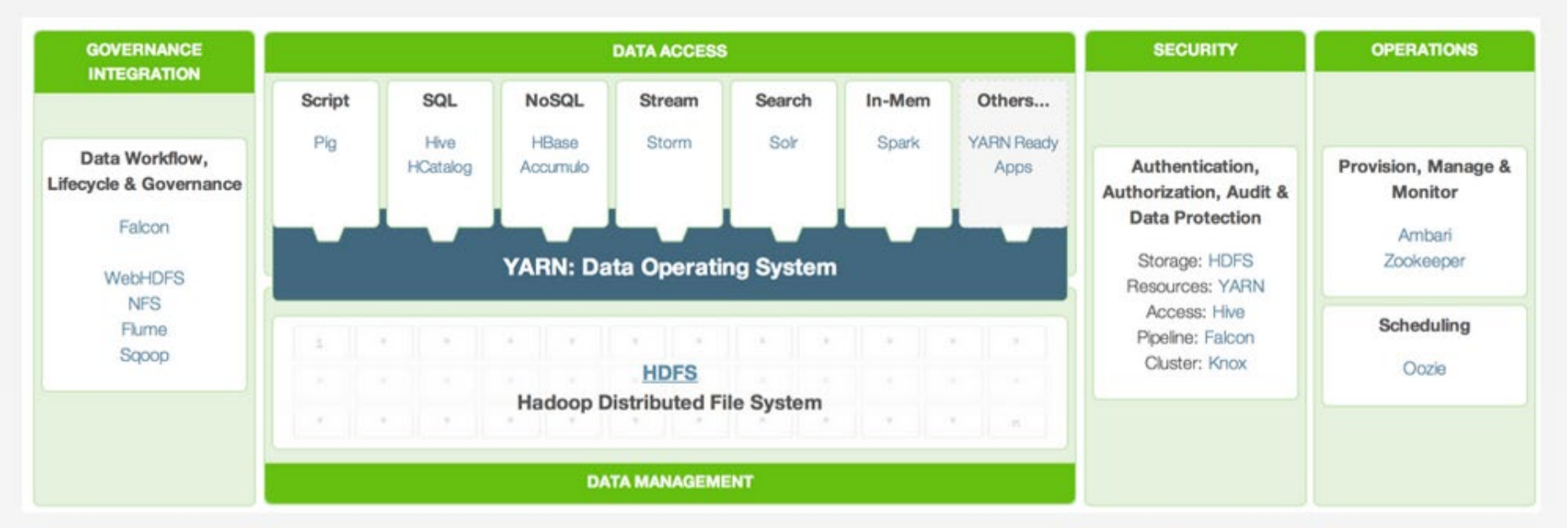
- Apache Hadoop is a community driven **open-source project** governed by the Apache Software Foundation.
- It was originally implemented at Yahoo based on **papers published by Google** in 2003 and 2004.
- **Scalable and economical** data storage and processing:
  - Distributed and fault tolerant
  - Harnesses the power of industry standard hardware

Core of Apache Hadoop

- ‘Core’ Hadoop consists of two main components:
  1. **Storage: the Hadoop Distributed File System (HDFS)**
  2. **Processing: MapReduce**
  - Plus the infrastructure needed to make them work, including:
    - Filesystem and administration utilities
    - Job scheduling and monitoring

What is Apache Hadoop?

- Apache Hadoop matured and developed to become a **data platform**



YARN: *MapReduce 2.0 (MRv2)*

Benefit 1: Scalability

Definitions:

- Individual machines are known as nodes
  - Each node both stores and processes data
- A cluster: a set of nodes
- A cluster can have as few as one node to as many as several thousands.
- A set of machines running HDFS and Map Reduce is known as a *Hadoop Cluster*.

- How do we increase performance?: Add more nodes to the cluster to increase scalability
  - You can expect better performance by simply adding more nodes.
  - However, if we keep adding nodes, we are guaranteed that some of them will fail…

Benefit 2: Fault Tolerance

- Problem: adding nodes increases chances that any one of them will fail
  - **Solution: build redundancy into the system and handle it automatically**
- Files loaded into HDFS are replicated across nodes in the cluster
  - **If a node fails, its data is re-replicated using one of the other copies**
- Data processing jobs are broken into individual tasks
  - Each task takes a small amount of data as input
  - **Thousands of tasks often run in parallel**
  - If a node fails during processing, its tasks are rescheduled elsewhere
- Routine failures are handled automatically without any loss of data

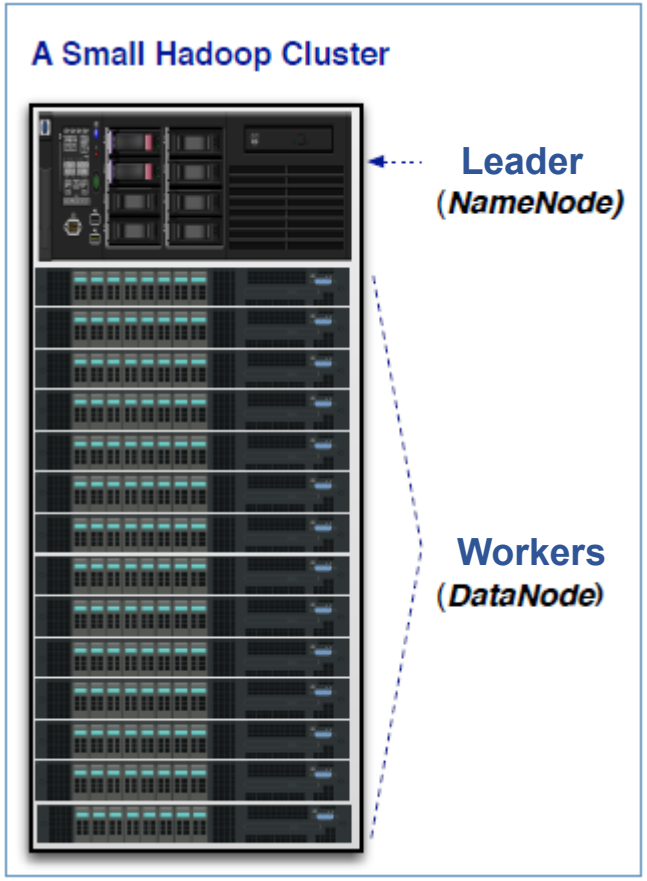
Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS)

- Provides **inexpensive and reliable** storage for massive amounts of data
- Optimized for sequential access to a relatively small number of large files
- Must use Hadoop-specific utilities or custom code to access HDFS
- Not built into the OS, so only specialized tools can access it
- End users typically access HDFS via the *hadoop fs* command

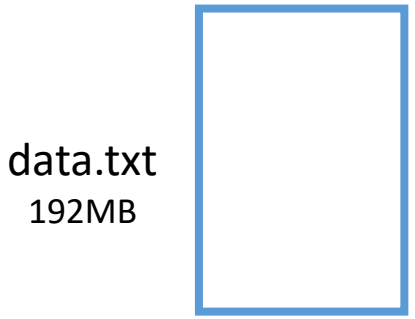
HDFS Architecture

- Leader/worker architecture (also called primary/secondary, leader/follower, main/worker)
- HDFS Leader:
  - Manages metadata and monitors workers
- HDFS Worker:
  - Reads and writes the actual data



HDFS - Review

- Files are stored in HDFS. What's going on behind the scenes?
- Imagine we are going to store a file called data.txt in HDFS

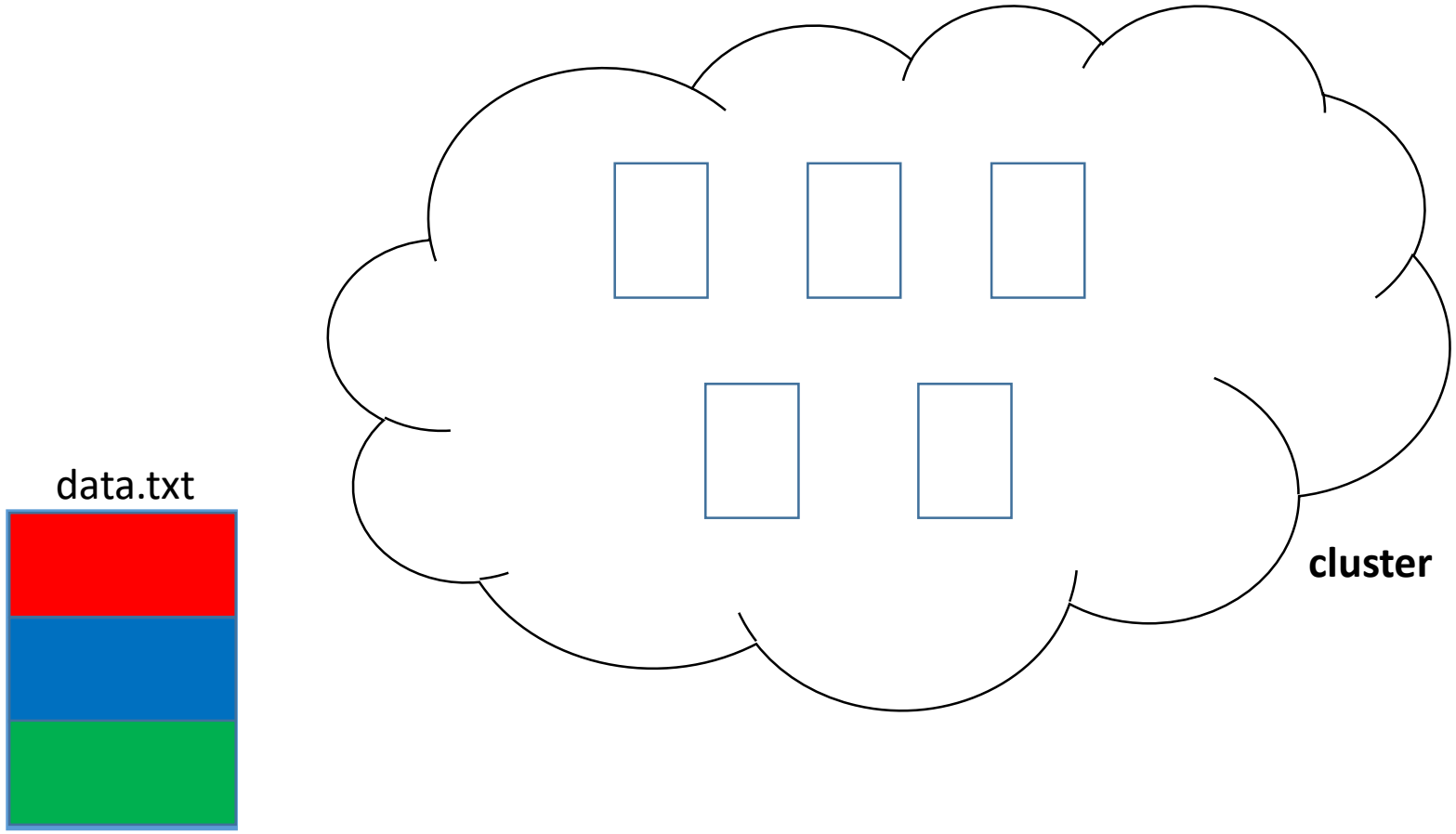


HDFS - Review

- When a file is loaded into HDFS, it is divided into blocks:

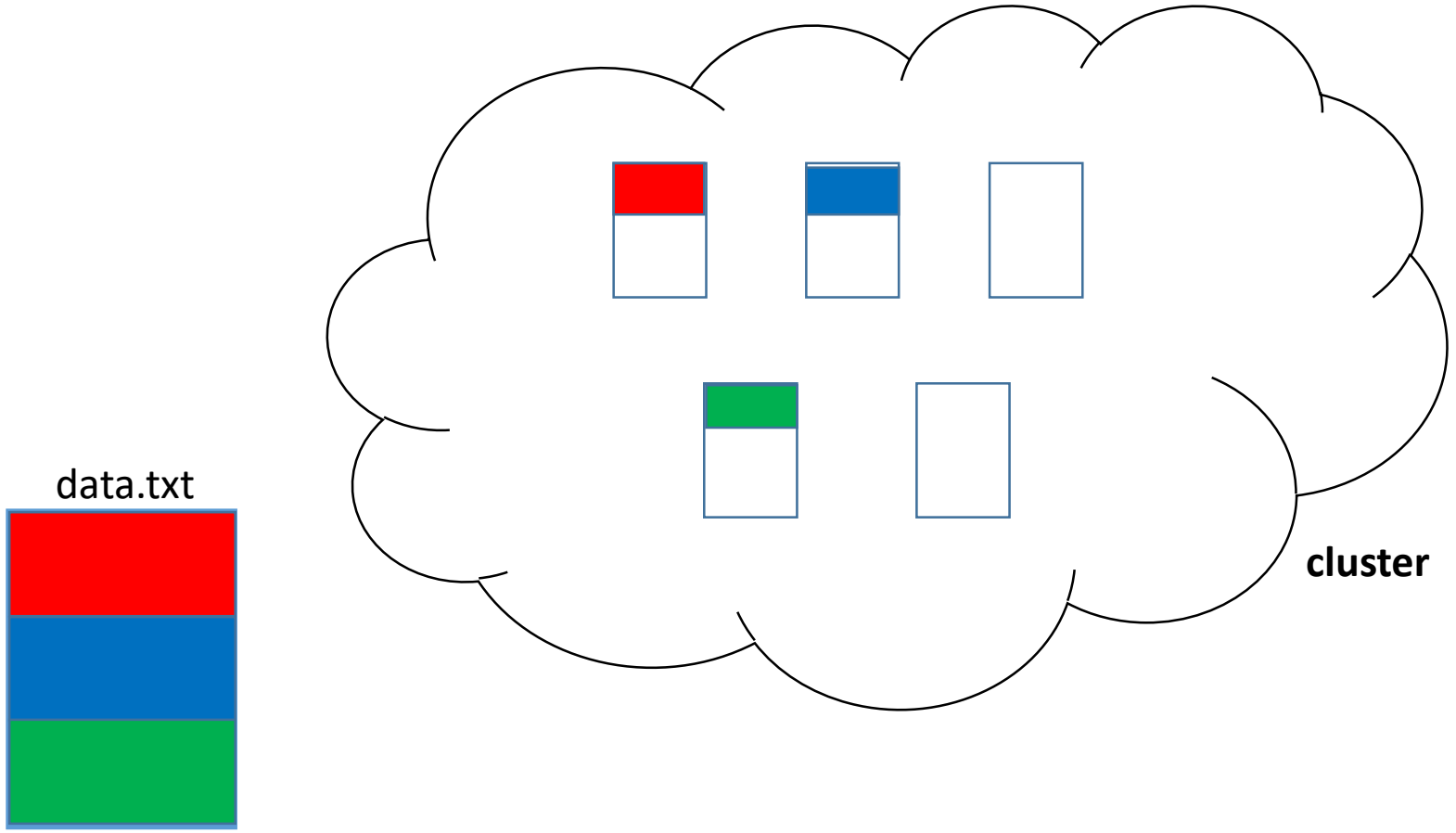
HDFS - Review

- The file is uploaded into HDFS:



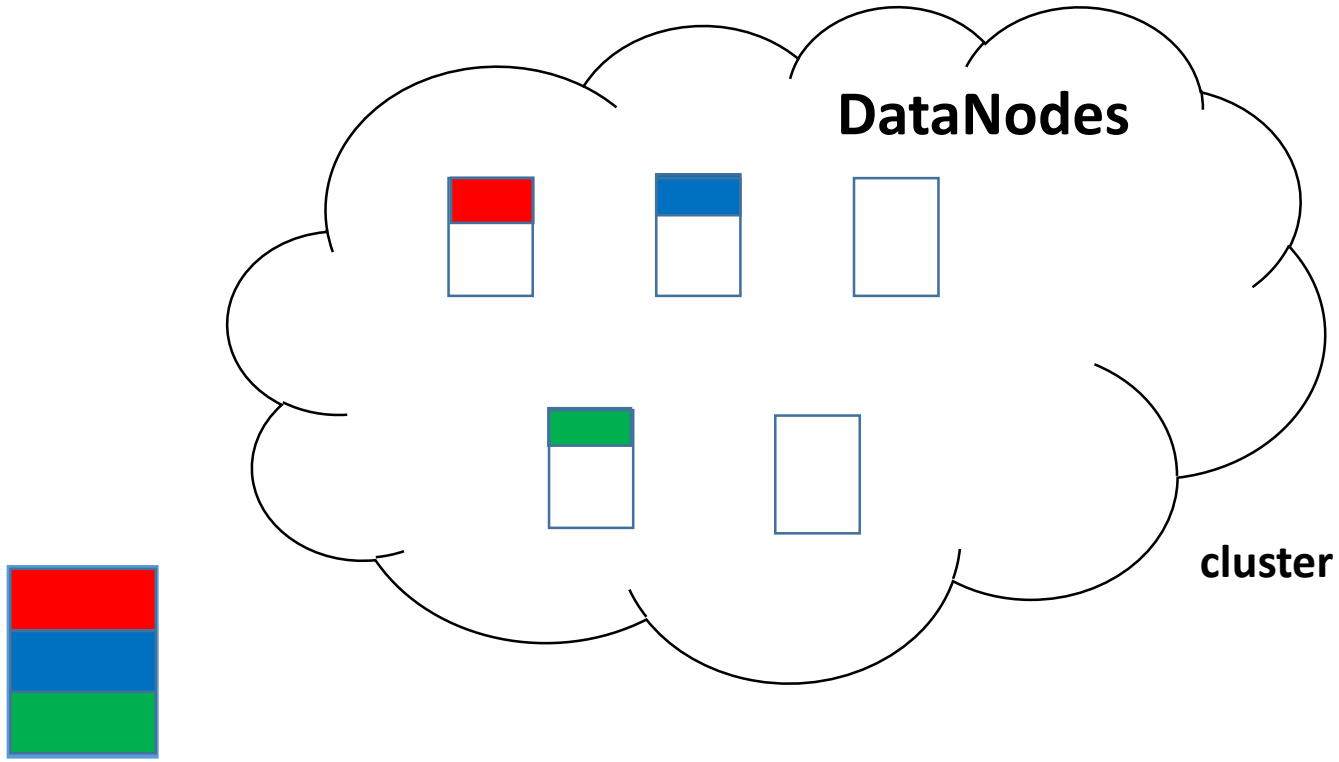
HDFS - Review

- Each block will get stored in one node in the cluster:



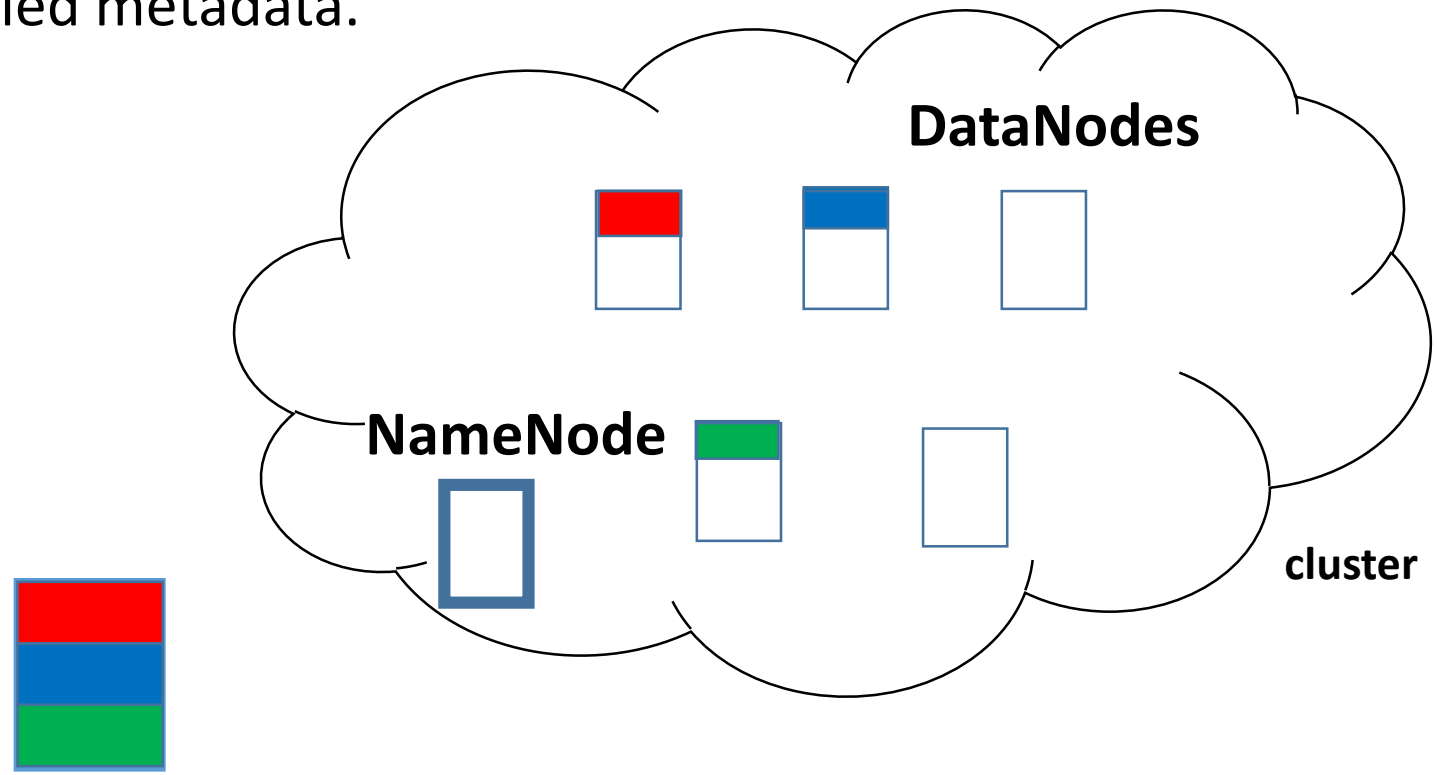
HDFS - Review

- DataNode is responsible for storing the actual data in HDFS:



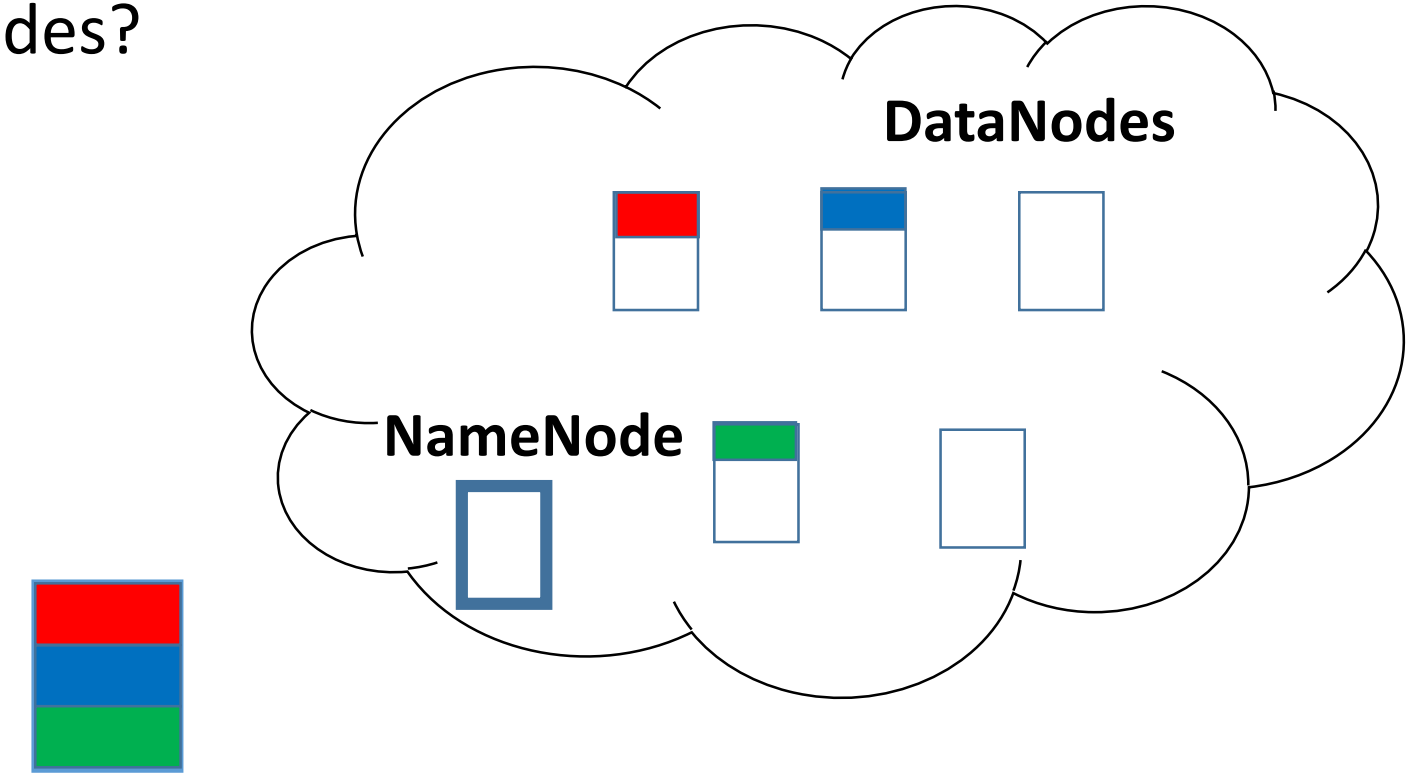
HDFS - Review

- We need to know which blocks make up the original file, and that is handled by a separate node called the NameNode. The information stored in the NameNode is called metadata.



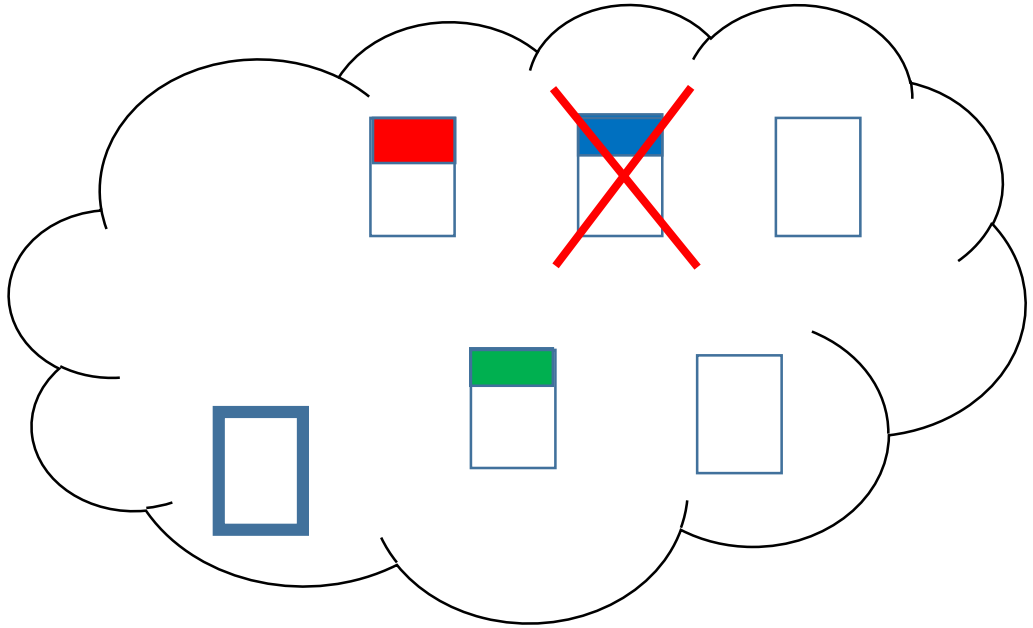
HDFS - Review

- Is there a problem? What happens if we have a disk failure in one of the nodes?



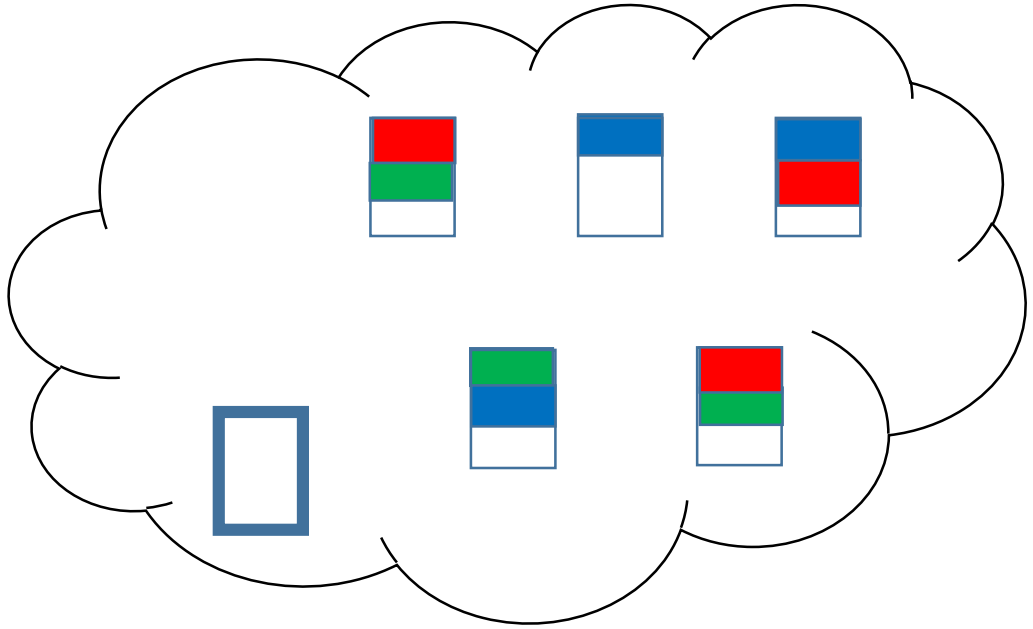
HDFS – Review – Data Redundancy

- The problem with things right now is that if one of our nodes fails, we are left with missing data for the file



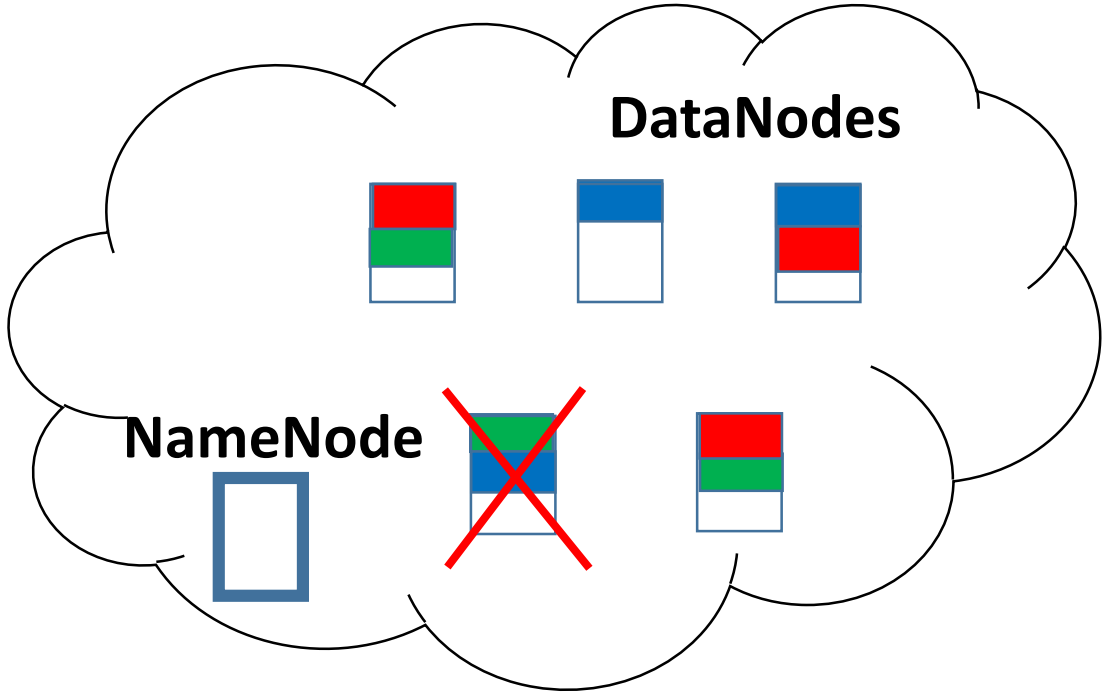
HDFS – Review – Data Redundancy

- To solve this problem, Hadoop replicates each block THREE times as it is stored in HDFS:



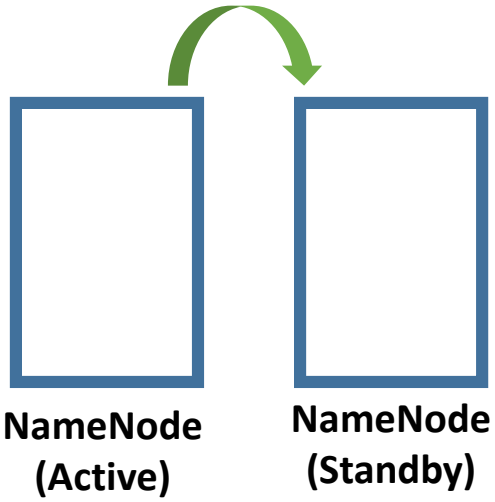
HDFS – Review – Data Redundancy

- Now, if a random node fails, it's not a problem because we have two more copies of the block in another node.



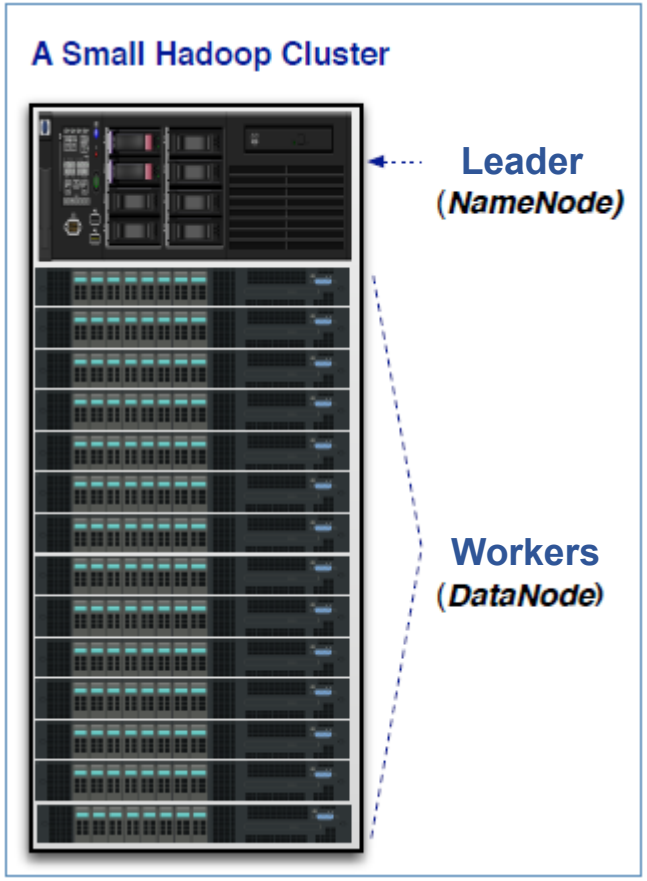
HDFS – Review – NameNode High Availability

- What happens if the NameNode has a hardware problem?
  - Data could be inaccessible
- To avoid the problem, the NameNode is not a single point of failure because they have configured two NameNodes: the Active NameNode works as before, but the Standby can be configured to take over if the Active one fails.



HDFS Architecture

- Leader/worker architecture (also called primary/secondary, leader/follower, main/worker)
- HDFS Leader:
  - Manages metadata and monitors workers
- HDFS Worker:
  - Reads and writes the actual data



MapReduce

MapReduce

- Method for distributing a task across multiple nodes
- We process data in Hadoop using MapReduce
- Each node processes data stored on that node to the extent possible
- The primary advantages of abstracting your jobs as MapReduce running over a distributed infrastructure are:
  - **Automatic parallelization and distribution of data in blocks across a distributed, scale-out infrastructure**
  - **Fault-tolerance** against failure of storage, compute and network infrastructure
  - **Deployment, monitoring and security capability**

MapReduce

- A running Map Reduce job consists of various phases such as:

Map → Sort → Shuffle → Reduce

- MapReduce is not a language, it's a programming model
- Most MapReduce programs are written in Java (can also be written in any scripting language using the Streaming API of Hadoop)

MapReduce: Terminology

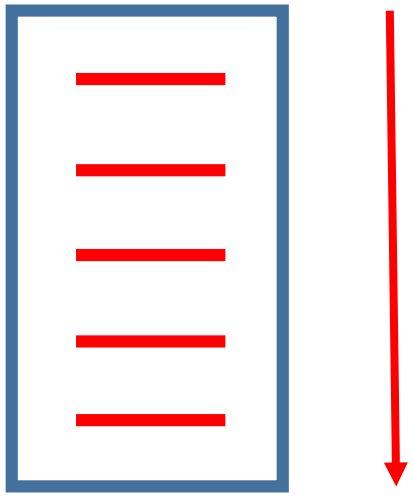
- JobTracker:
  - MapReduce jobs are controlled by a software daemon called JobTracker
  - Resides on a ‘leader node’
  - Clients submit MapReduce jobs to the JobTracker
  - It assigns Map and Reduce tasks to other nodes on the cluster
- TaskTracker:
  - Nodes in the cluster run a software daemon called TaskTracker
  - Responsible for initiating the Map or Reduce task
  - Reports progress to the JobTracker
- A job: program with the ability of complete execution of Mappers and Reducers over a dataset

MapReduce: Terminology

- The Mapper:
  - Mappers (attempts to) run on nodes which hold their portion of data locally, to minimize network traffic.
  - Multiple Mappers **run in parallel**, each processing a portion of the input data
  - After the Map phase is over, **all the intermediate values are given to the Reducer**
- The Reducer:
  - Intermediate values are passed to the **Reducer in sorted key order**
  - Values are **written to HDFS**

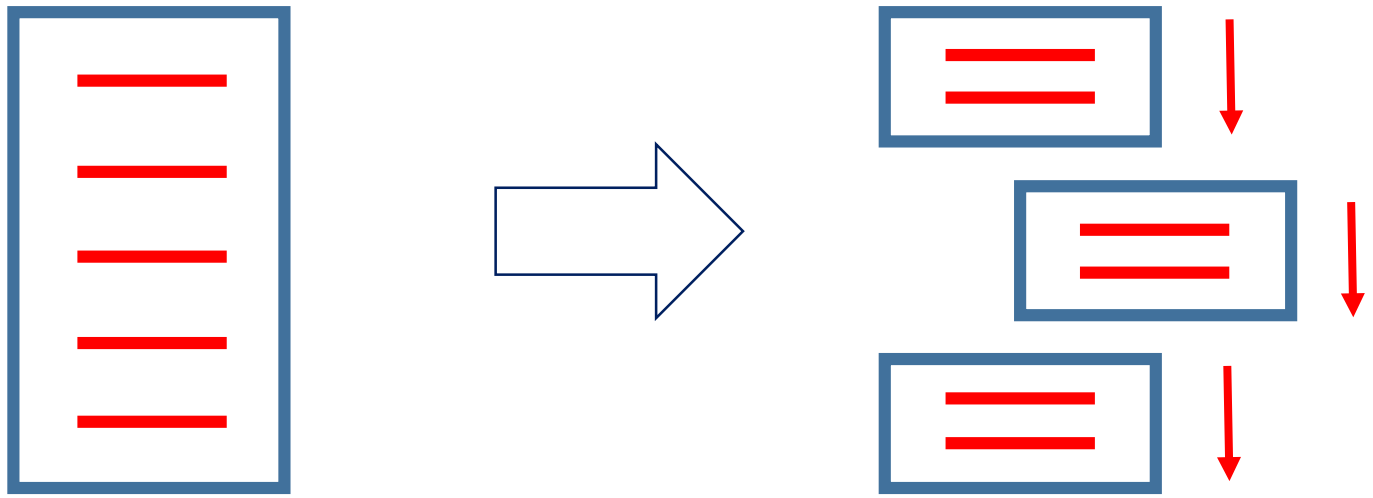
MapReduce example

- Scenario: you have a large file. Processing the file serially, from the top to the bottom, could take a long time.



MapReduce example

- Solution: MapReduce is designed to process data in parallel. Your file is broken into pieces, and then processed in parallels.



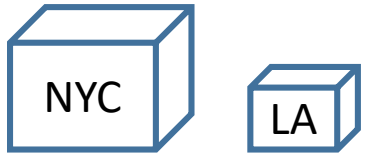
MapReduce example

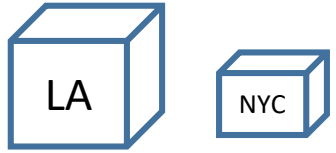
- Objective: given a file, calculate the total sales per city for the year.

Date	City	Category	Amount
01/02/21	NYC	Toys	4.10
01/02/21	Miami	Clothes	21.99
01/01/21	LA	Clothes	25.99
01/01/21	Miami	Music	12.15
01/02/21	NYC	Toys	3.10
01/02/21	Miami	Clothes	50.00
		.	
		.	

MapReduce example

Mappers



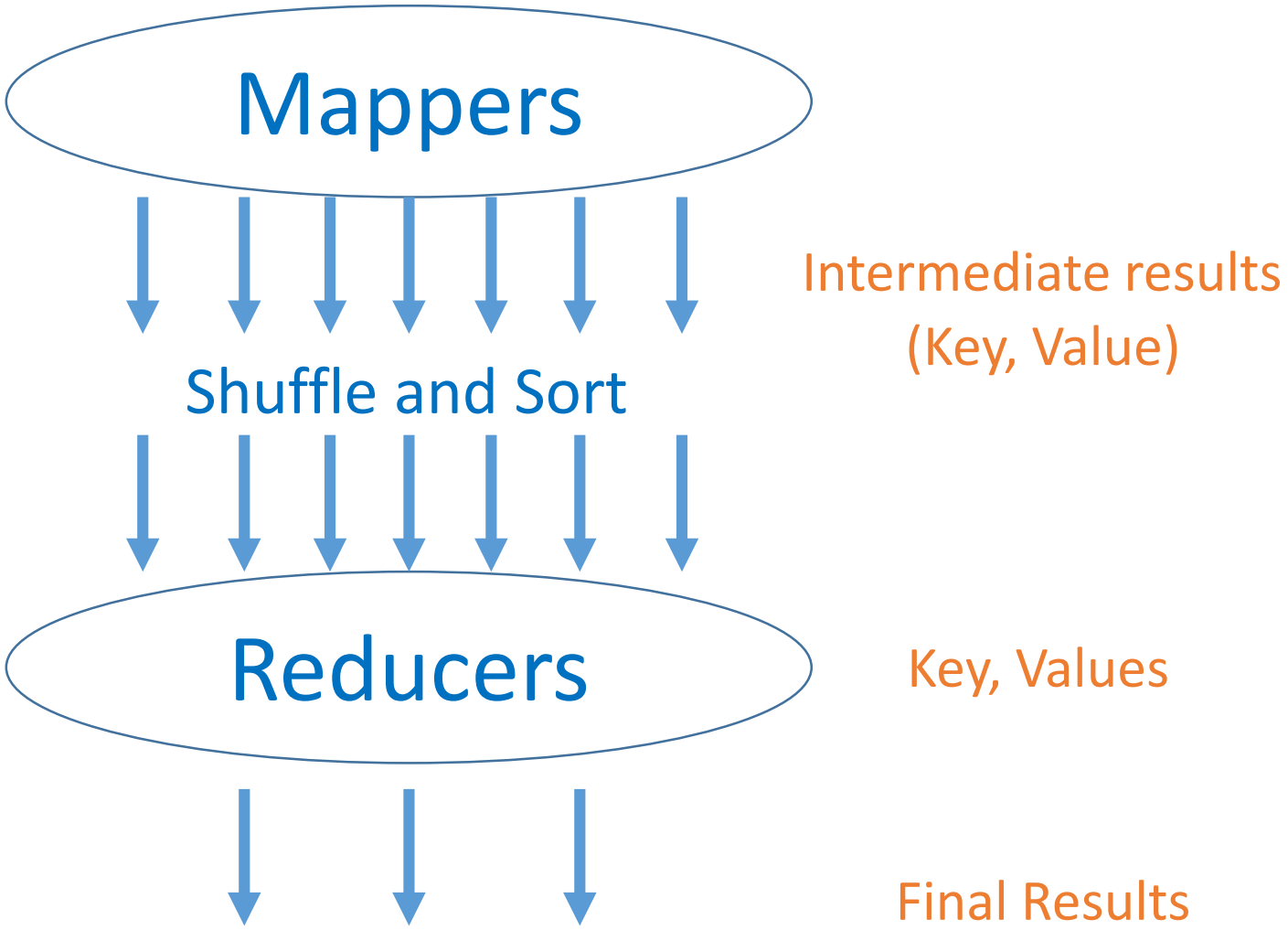


NYC

Miami, LA

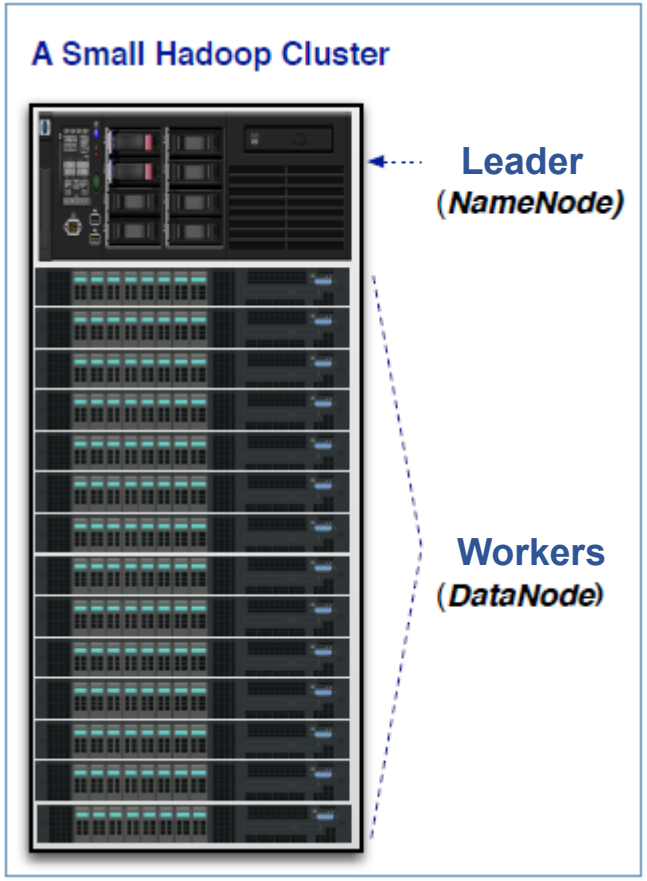
Reducers

MapReduce summary



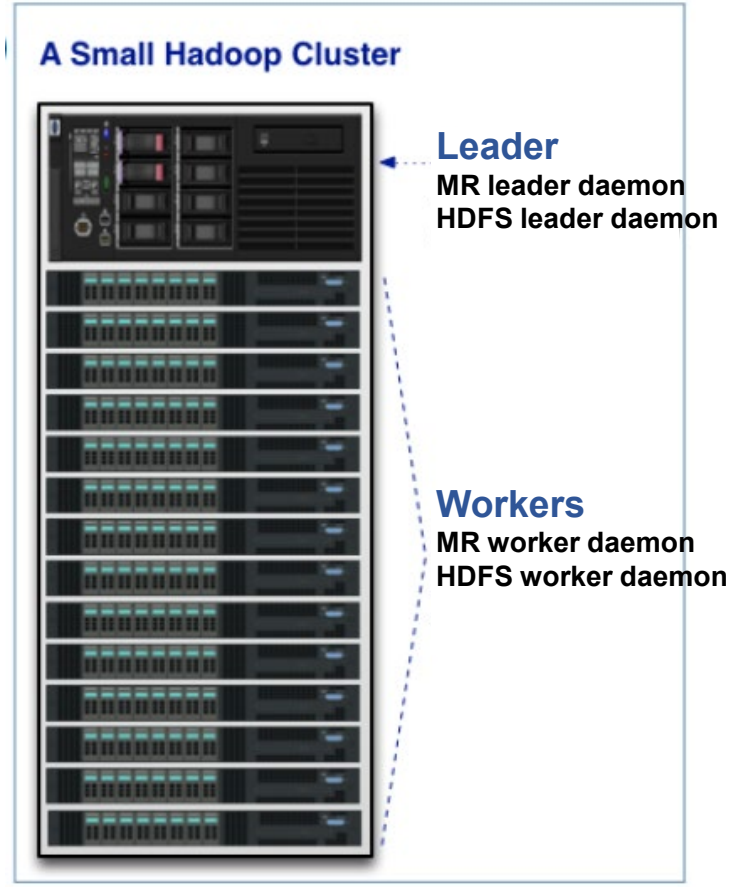
Remembering: HDFS Architecture

- Leader/worker architecture (also called primary/secondary, leader/follower, main/worker)
- HDFS Leader:
  - Manages metadata and monitors workers
- HDFS Worker:
  - Reads and writes the actual data



MapReduce Architecture

- Leader/worker architecture
- Leader node:
  - Run leader daemons to accept jobs, and monitor and distribute work
- Worker nodes:
  - Run worker daemons to start tasks
  - Do the actual work
  - Report status back to leader daemons
- HDFS and MapReduce are collocated
  - Worker nodes run both HDFS and MR worker daemons on the same machines



IT-Stammtisch DARMSTADT 07.11.13

Try that with Java...

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));

                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

Pig and Hive: Motivation

- MapReduce code is typically written in Java. Although it can be written in other languages using the Streaming API of Hadoop
- Streaming Requires a programmer who understands how to think in terms of MapReduce, who understands the problem they're trying to solve and who has enough time to write and test the code.
- Meanwhile, **many other people want to analyze data**
- **Needed:** a **higher-level abstraction on top of MapReduce providing the ability to query the data** without needing to know MapReduce intimately.
- **Solution:** Pig and Hive address these needs.

Where to go now?

- Introducing Apache Hadoop: The Modern Data Operating System
- Introduction to the Hadoop Ecosystem
- Hadoop: The Definitive Guide (O'Reilly)
- Big Data Now (O'Reilly)
- The Google File System (GFS)
- MapReduce: Simplified Data Processing on Large Clusters

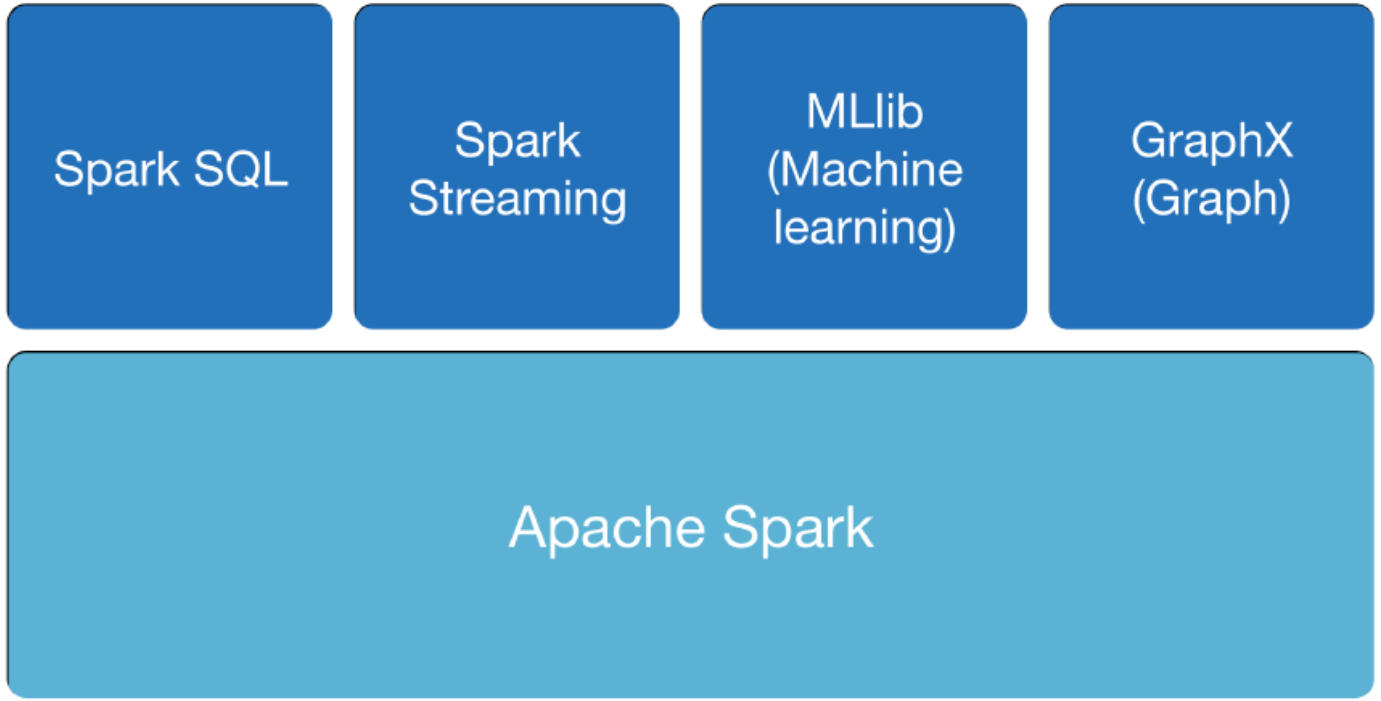
Introduction to Spark

What is Spark?

- Apache Spark is an open-source, cluster computing framework
- Developed with a focus on interactive, iterative computations
  - Utilizes *in-memory processing*
  - Makes it ideal for data science applications (data mining, machine learning)
- Extensive API support for Java, Scala, R and Python
  - You can use it *interactively* from the Scala, Python and R shells

Apache Spark Ecosystem Components

- Combine SQL, streaming, and complex analytics.



Why Use Spark for Data Science?

- **Iteration**
  - Spark was designed to facilitate iterative computation
- **Ease of use**
  - Multiple available APIs allow for familiar development environments
- **Speed**
  - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- **Runs Everywhere**
  - Spark runs on Hadoop, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

What is the **relationship** between Spark and Hadoop? (1)

1. Hadoop and Apache Spark are both **big-data frameworks**.
2. They don’t serve the same purposes:
   - Hadoop is essentially a distributed data infrastructure: It distributes massive data collections across multiple nodes within a cluster of commodity servers. It also indexes and keeps track of that data, enabling big-data processing and analytics far more effectively than was possible previously.
   - Spark is a data-processing tool that operates on those distributed data collections; it doesn't do distributed storage.
3. Spark *does not* provide storage
   - Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, etc.
4. You can use one without the other
   - Hadoop includes storage component (HDFS), but also the processing component (MapReduce): so you don't need Spark to get your processing done.
   - Conversely, you can also use Spark without Hadoop. **Spark does not come with its own file management system**, though, so it needs to be integrated with one -- if not HDFS, then another cloud-based data platform.

What is the **relationship** between Spark and Hadoop? (2)

5. Spark is speedier:
   - MapReduce operates in steps, Spark operates on the whole data set in one fell swoop.
   - MapReduce workflow: read data from the cluster, perform an operation, write results to the cluster, read updated data from the cluster, perform next operation, write next results to the cluster, etc.
   - Spark: read data from the cluster, perform all of the requisite analytic operations, write results to the cluster, done.
6. MapReduce and Spark both provide a framework for scalable data processing
   - Both platforms provide fault tolerance

What is the **relationship** between Spark and Hadoop? (3)

7. MapReduce uses only two fundamental data processing operations, while Spark supports over 80 distinct operations
   - Most machine-learning algorithms, for example, require multiple operations
8. Disk read/write vs. in-memory processing
   - MapReduce must write to and read from disk between operations
   - Spark utilizes in-memory processing to avoid intermediate reads/writes

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Can MapReduce and Spark live together?

- MapReduce is the original processing framework for Hadoop
- Spark is a great choice for large-scale data mining
- Spark is great for iterative processing (machine learning)
- Through resource managers like YARN, Hadoop can easily support shifting workloads of both MapReduce and Spark programs

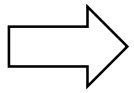
Can I use Spark without Hadoop?

Yes!

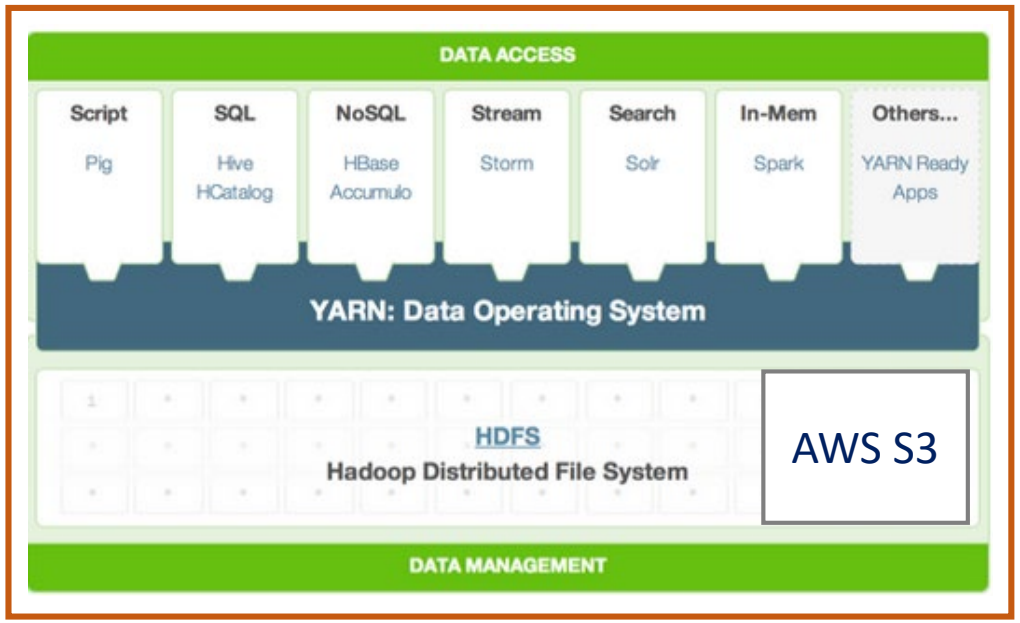
… you can also use Spark with Hadoop. Spark does not come with its own file management system, though, so it needs to be integrated with one -- if not HDFS, then another cloud-based data platform.

AWS Hadoop Distribution: Amazon Elastic MapReduce (EMR)

AWS EMR



AWS EMR



- Amazon EMR includes EMRFS, a connector allowing Hadoop to use **S3 as a storage layer**.
- **HDFS** is automatically installed with Hadoop on your EMR cluster, and **you can use HDFS along with Amazon S3 to store your input and output data.**
- Amazon EMR configures Hadoop to use **HDFS for intermediate data** created during MapReduce jobs, even if your input data is located in Amazon S3.

Amazon EMR programmatically installs and configures applications in the Hadoop project, including Hadoop MapReduce (YARN), and HDFS, across the nodes in your cluster.

Where now?

- Hadoop + Spark Quiz (due today at midnight – 1 hour to complete)
- Quizzes are open notes
- You will receive an AWS Academy email invitation - AWS Academy Learner Lab