INGI2145
Cloud Computing
Lab 3: Hadoop MapReduce

Nicolas LAURENT

Slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives

Goals

- Understand the Hadoop data flow and be able to write and run simple MapReduce programs on a local Hadoop.

- Understand how a distributed Hadoop works internally (HDFS, etc).

Goals

- Understand the Hadoop data flow and be able to write and run simple MapReduce programs on a local Hadoop.

- Understand how a distributed Hadoop works internally (HDFS, etc).

Hadoop Modes

- Local: Single Node, no distribution, no HDFS
- Pseudo Distributed
  (multiple nodes on a single machine)
- Fully Distributed

Distributed Modes

- Running Hadoop jobs on AWS Elastic MapReduce: setup done for you.

- Exercise today: local only. Ordinary Java program.

- Distributed setup (for you own enjoyment):

  - http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/
  - https://www.digitalocean.com/community/tutorials/how-to-install-hadoop-on-ubuntu-13-10
  - https://github.com/ericduq/hadoop-scripts/blob/master/make-single-node.sh

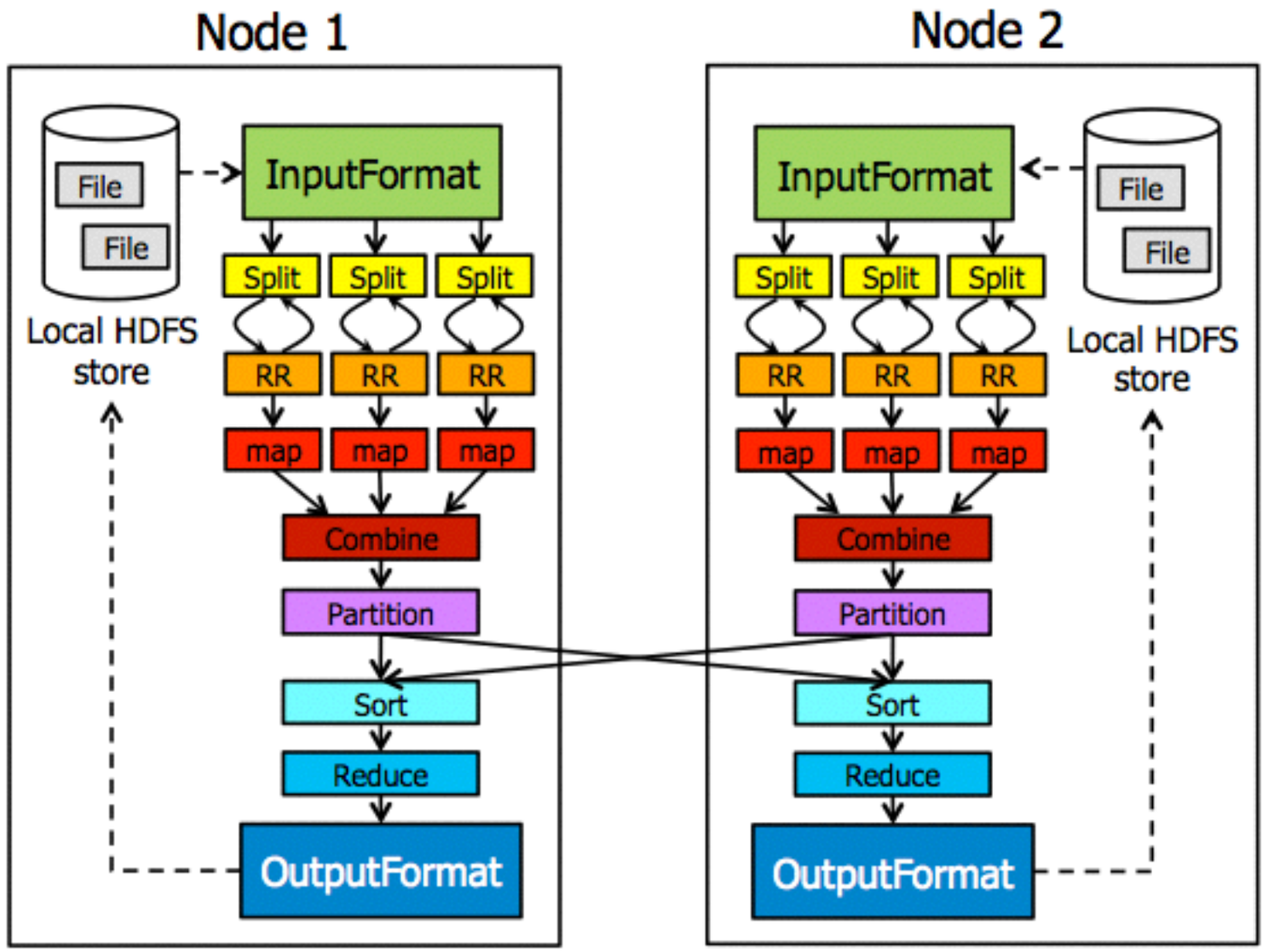
MapReduce

<key1, value1>* → map → <key2, value2>*

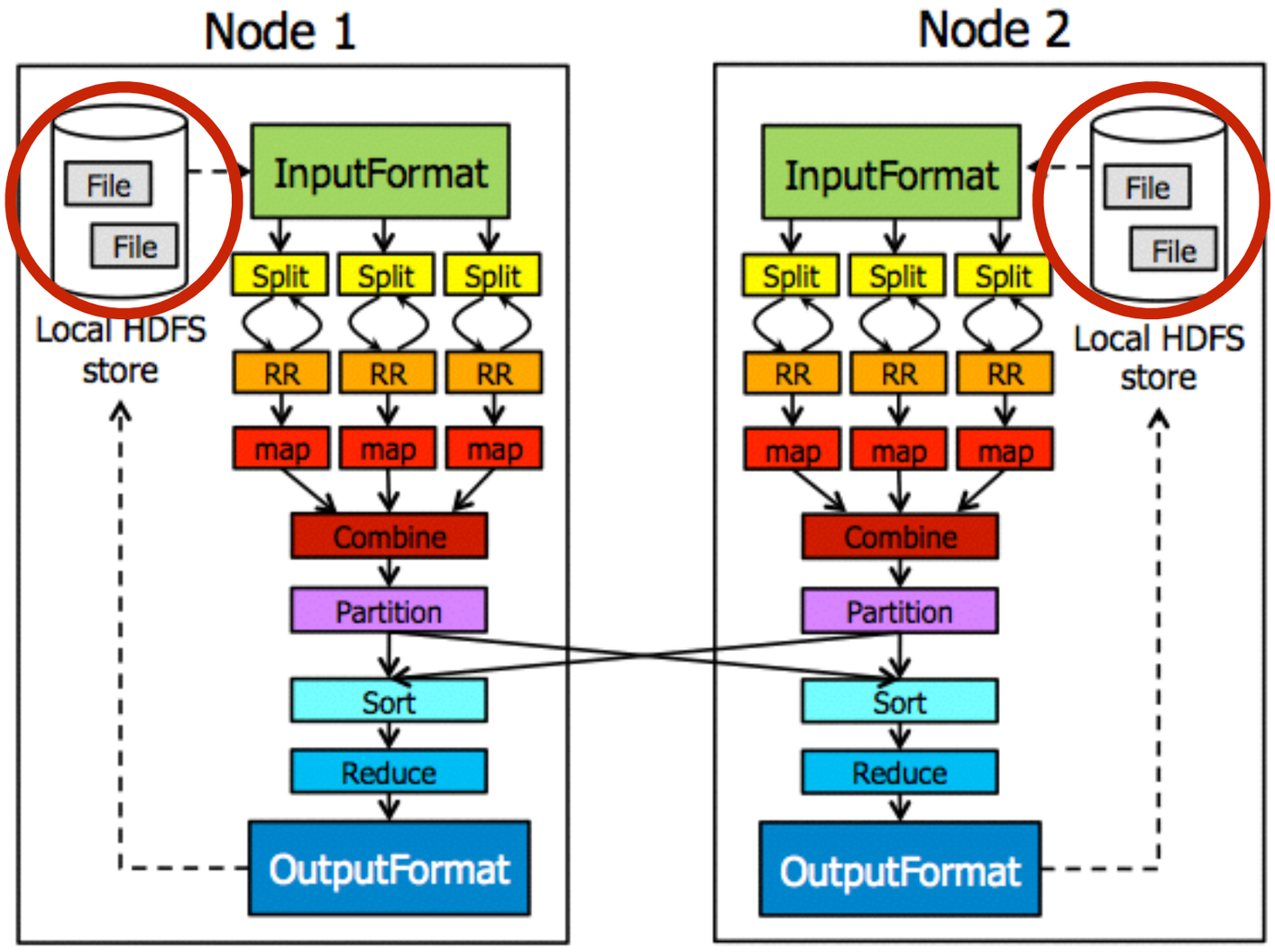
<key2, value2*>* → reduce → <key3, value3>*

- Usually, key1 = line number, value1 = line.
- <key3, value3> will be written on a single line in an output file.
- What really matters is key2: the reduce key.

Hadoop Data Flow



Input / Output

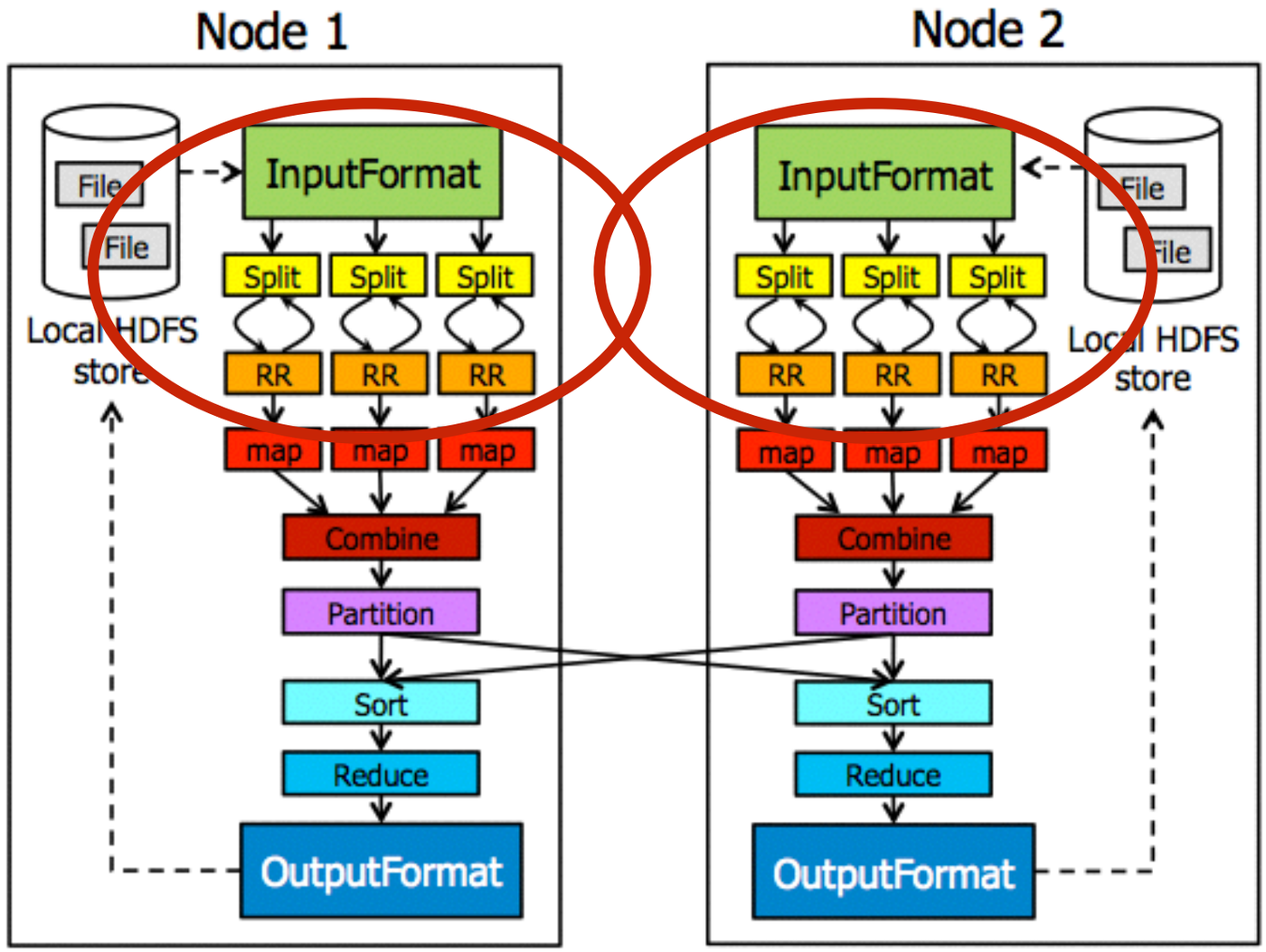


Input / Output

- The input is a set of files in an input directory.

- The output directory must not exist (will be created by Hadoop).

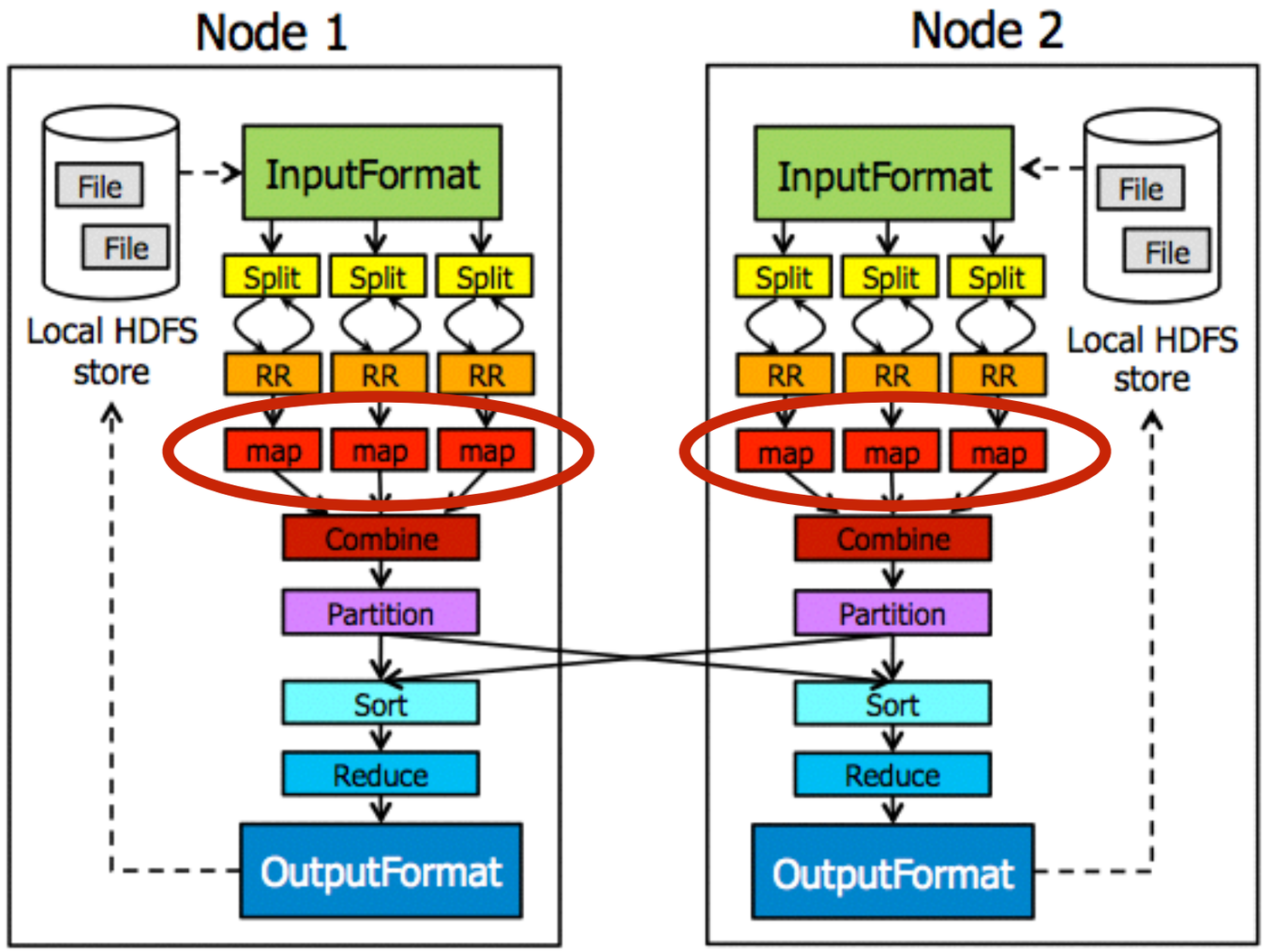
Input Split & Record Readers



Input Split & Record Readers

- You can configure how the input is split between different nodes for mapping.
- By default: fixed size chunks, split on line boundaries.
- Before mapping, convert each input split into a series of records, using a record reader.
- By default, one line = one record.
- Both aspects are defined by the “input format”.

Mapper


Node 1
File
File
Local HDFS store
InputFormat
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat
Node 2
InputFormat
File
File
Local HDFS store
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat

Mapper

```
class MyMapper
extends Mapper<LongWritable, Text, Text, Text>
{
    @Override protected
    void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException
    {
        . . .
        context.write(new Text(“foo”), value);
        . . .
    }
}
```

key1 value1 key2 value2

write a new <key2, value2> pair (can be called multiple times)

A note on Key/Value Types

- All types used for keys or values must implement the Writable interface.

- This is the Hadoop equivalent of Serializable, meant to be more performant.

Writable Pitfalls

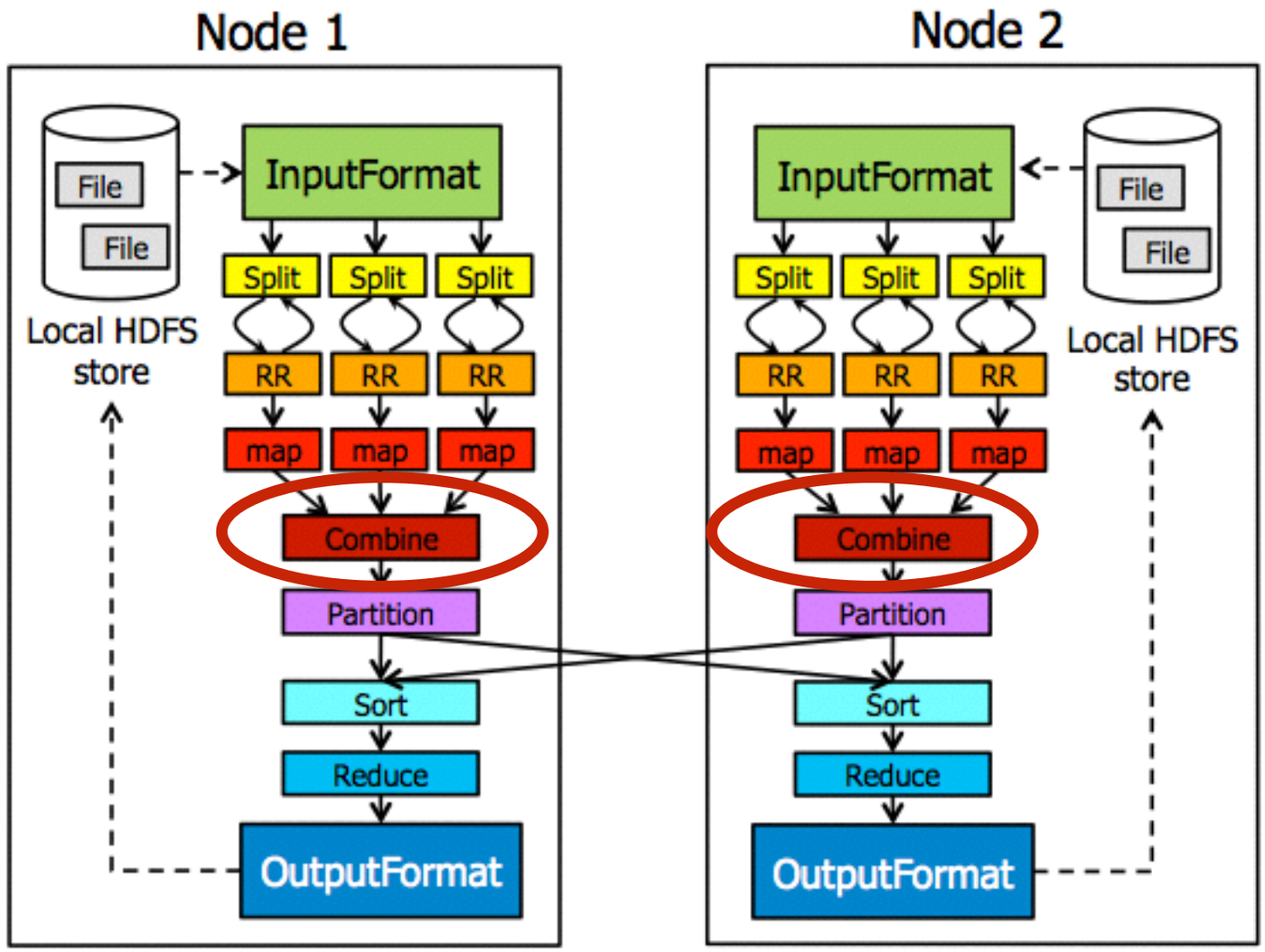
- Implementing Writable: most often you'll want to make a type which is a composite of other Writable types. All such child Writable types **must** be initialized in the mandatory no-argument constructor.

- Using Writable: Hadoop reuses Writable objects, changing their content. When you get hold of a Writable object, save its content if you need to use it after calling another Hadoop method.

Example Writable Implementation

```
static class WordWithCount implements WritableComparable<WordWithCount> {
    Text word = new Text();
    IntWritable count = new IntWritable();

    @Override public void write(DataOutput out) throws IOException {
        word.write(out);
        count.write(out);
    }

    @Override public void readFields(DataInput in) throws IOException {
        word.readFields(in);
        count.readFields(in);
    }

    @Override public int compareTo(WordWithCount o) {
        int comp = -count.compareTo(o.count);
        return comp == 0 ? 1 : comp;
    }

    @Override public String toString() {
        return String.format("%s %d", word.toString(), count.get());
    }
}
```

Combiner

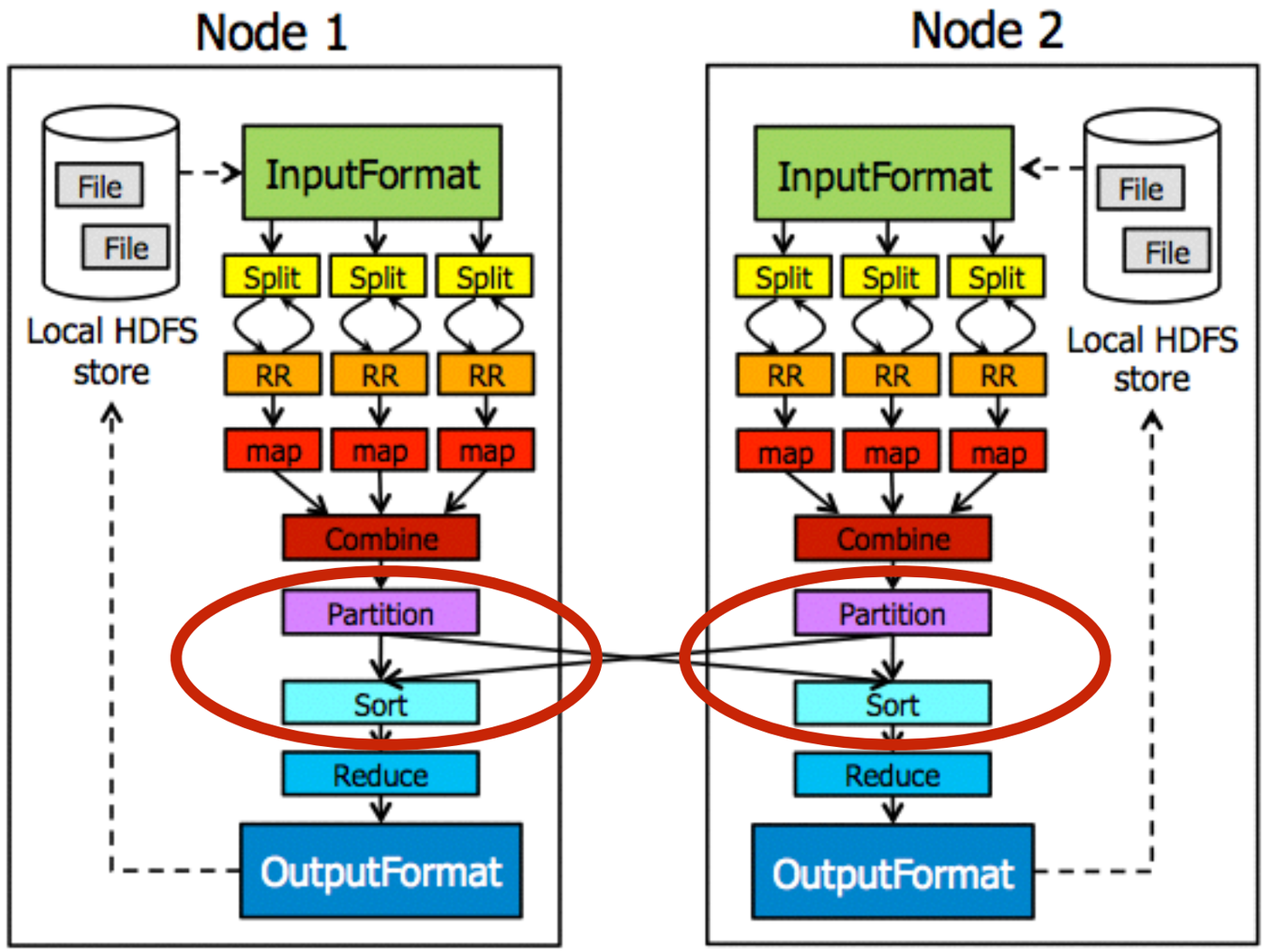

Node 1
File
File
Local HDFS store
InputFormat
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat
Node 2
InputFormat
File
File
Local HDFS store
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat

Combiner

- The combiner is a local reducer: it only runs on the subset of key-value pairs mapped by the node it runs on.

- Using a combiner is never necessary, but it is a very handy optimization for some use cases. It can reduce the data sent over the network significantly.

Partitioning & Sorting

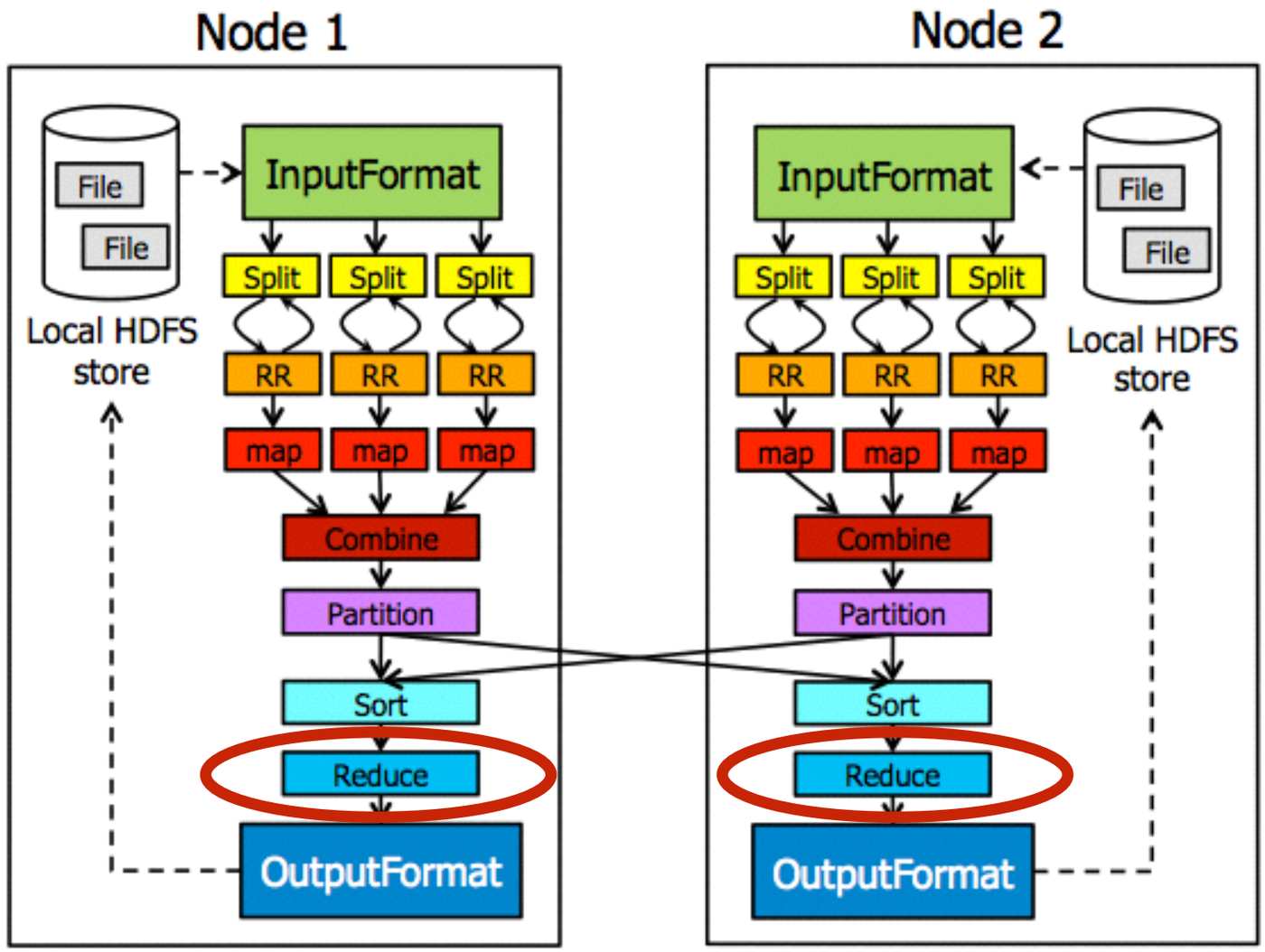


Partitioning & Sorting

- Partitioning: Hadoop assigns each key in the reduce key set to a single node which will run the reduce step for this key. Uses key hashing by default.

- Sorting: the order in which keys are reduced on each node is defined by an ordering on the set of keys. e.g., lexicographic order

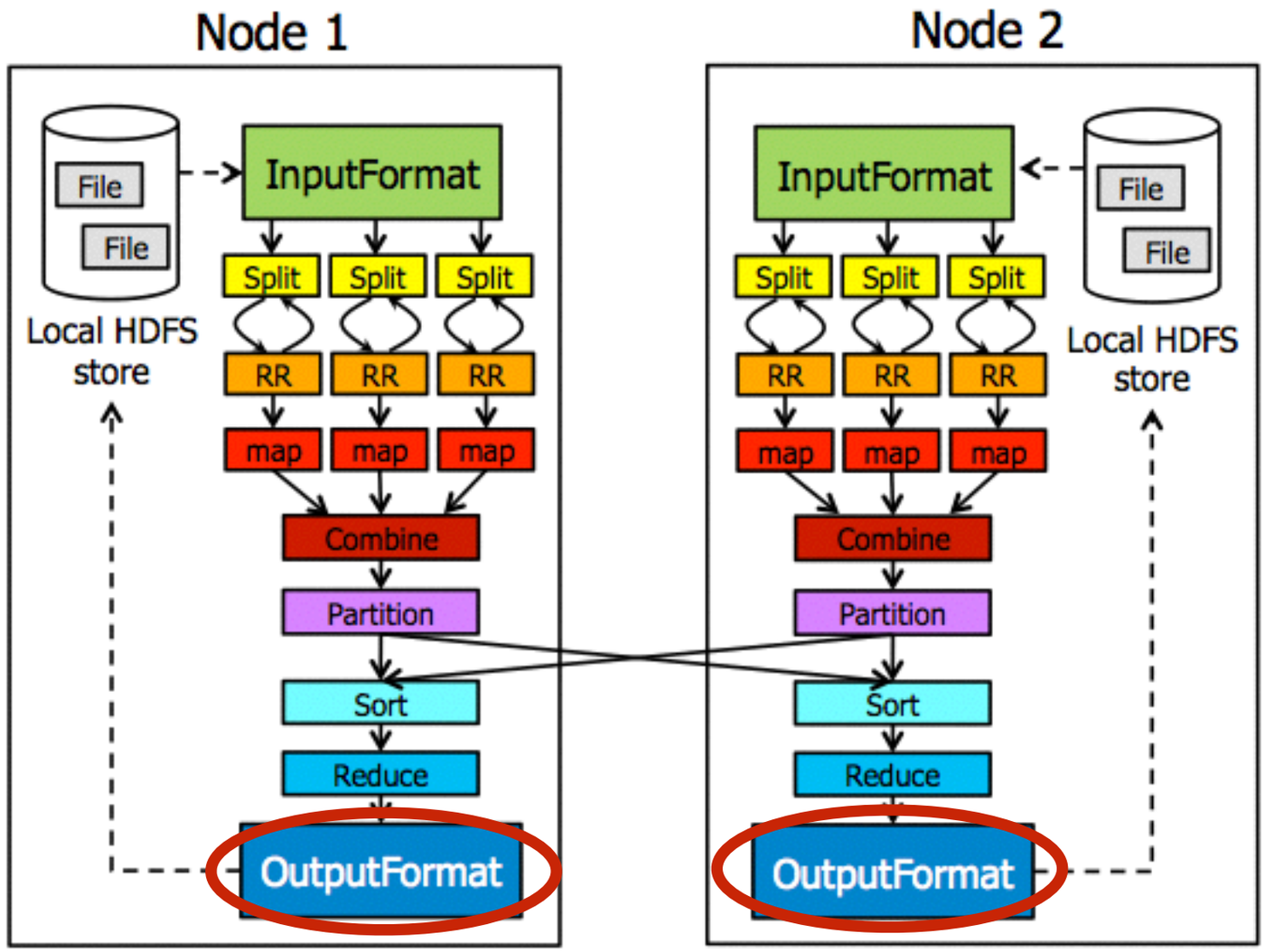
Reducer


Node 1
File
File
Local HDFS store
InputFormat
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat
Node 2
InputFormat
File
File
Local HDFS store
Split
Split
Split
RR
RR
RR
map
map
map
Combine
Partition
Sort
Reduce
OutputFormat

Reducer

```
class MyReducer key2 value2   key3     value3
extends Reducer<Text, Text, IntWritable, Text>
{
    @Override protected
    void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException
    {
        ...      write a new <key3, value3> pair (can be called multiple times)
        context.write(new IntWritable(42), values.iterator().get());
        ...
    }
}
```

Output Format



Output Format

- An output format supplies “record writers” to write a <key3, value3> pair to a file.

Configuring Hadoop: The Driver

```
public class FooDriver {
  public static void main(String[] args) throws Exception {
    Job job = new Job();
    job.setJarByClass(FooDriver.class);

    FileInputFormat.addInputPath(job, new Path("in"));
    FileOutputFormat.setOutputPath(job, new Path("out"));

    job.setMapperClass(FooMapper.class);
    job.setReducerClass(FooReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

input/output paths

mapper / reducer class

configure key3, value3

(key2, value2) can be inferred

Goals

- Understand the Hadoop data flow and be able to write and run simple MapReduce programs on a local Hadoop.

- Understand how a distributed Hadoop works internally (HDFS, etc).

HDFS

- The Hadoop Distributed File System (HDFS) is a virtual file system used internally by Hadoop.
- It solves the following issues:
  - How to efficiently store and transfer files in large data sets between nodes?
  - How to make the system failure-tolerant w.r.t. data loss.

HDFS
Strengths & Weaknesses

- HDFS targets large datasets: it is optimized for large read-only or append-only files.

- It works well with sequential access pattern: you can start reading a file even though you haven't received the whole of it yet.

- It does not work well with multiple writers, or with random access.

- Optimizes for throughput, not latency.

- HDFS is **distributed** and **replicated** !

- Shoddy permission handling.

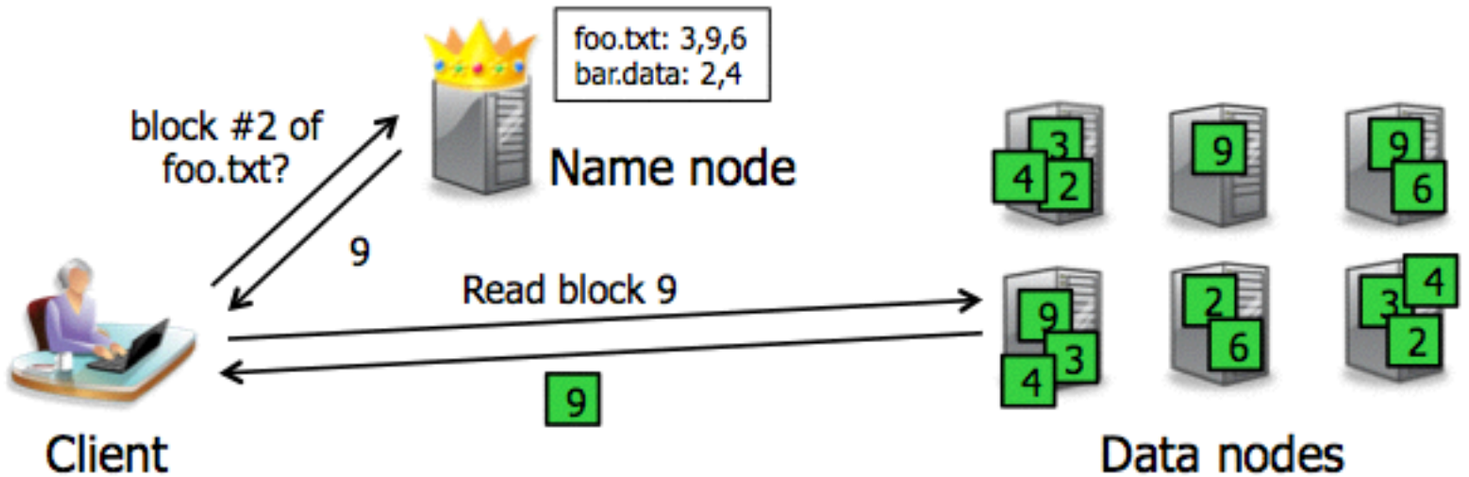
Accessing HDFS

- HDFS is backed by an ordinary file system, on which “blocks” are stored.
- HDFS is not meant to be mounted, but some projects make it possible.
- Use the command line interface to interact with HDFS.

HDFS CLI

hadoop fs -put [file] [hdfsPath]	Stores a file in HDFS
hadoop fs -ls [hdfsPath]	List a directory
hadoop fs -get [hdfsPath] [file]	Retrieves a file from HDFS
hadoop fs -rm [hdfsPath]	Deletes a file in HDFS
hadoop fs -mkdir [hdfsPath]	Makes a directory in HDFS

HDFS: How it Works



- Files are store as sets of large (default: 64MB) blocks. These blocks are replicated for durability and availability.

- The namespace is managed by a single name node.

Hadoop Daemons

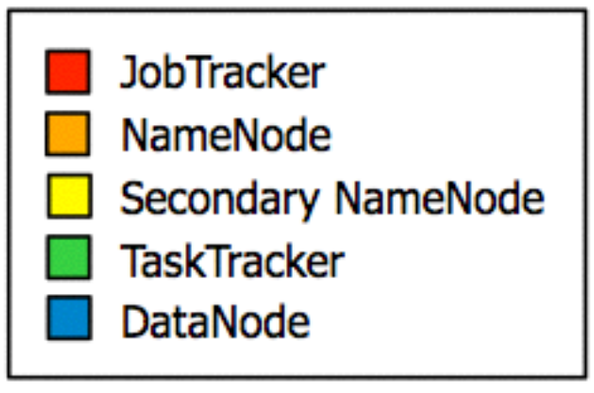
- **TaskTracker**
  Runs maps and reduces. One per node.
- **JobTracker**
  Accepts jobs; assigns tasks to TaskTrackers.
- **DataNode**
  Stores HDFS blocks.
- **NameNode**
  Stores HDFS metadata.
- **SecondaryNameNode**
  Merges edits file with snapshot;
  “backup" for NameNode.

A single node can run more than one of these!

Example Configurations

Small cluster

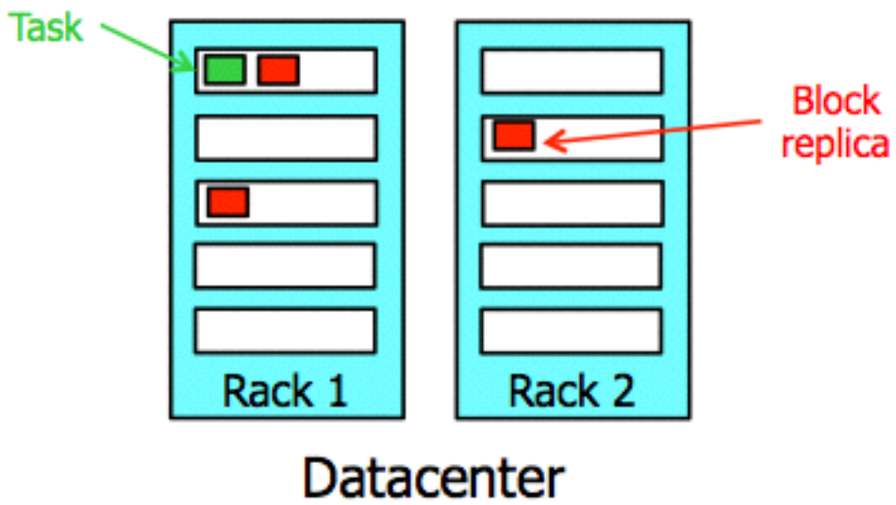
Medium cluster



Fault Tolerance

- If a node fails, the JobTracker notices. Need to re-run all map and reduce tasks assigned to the node.
- Speculative execution: if a node is faster than others, it can perform tasks assigned to other nodes. The first node to finish determines the definitive version, others are discarded.

Placement & Locality



- Read the block replica closest to where a task is running.

- Write replicas: trade-off between fault-tolerance and locality (performance).

AWS Demo:

Using Elastic MapReduce

Exercise:
Word Counting