

INGI2145

Cloud Computing

Lab 3: Hadoop MapReduce

Thanh Nguyen

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.

# Use-case comparison

- Hadoop is used for data analysis purpose.
  - It is not strong in data query without statistic function
- Traditional database
  - Schema support
  - Indexing
- Hadoop (MapReduce)
  - Unstructured data
  - Programmable

# 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>

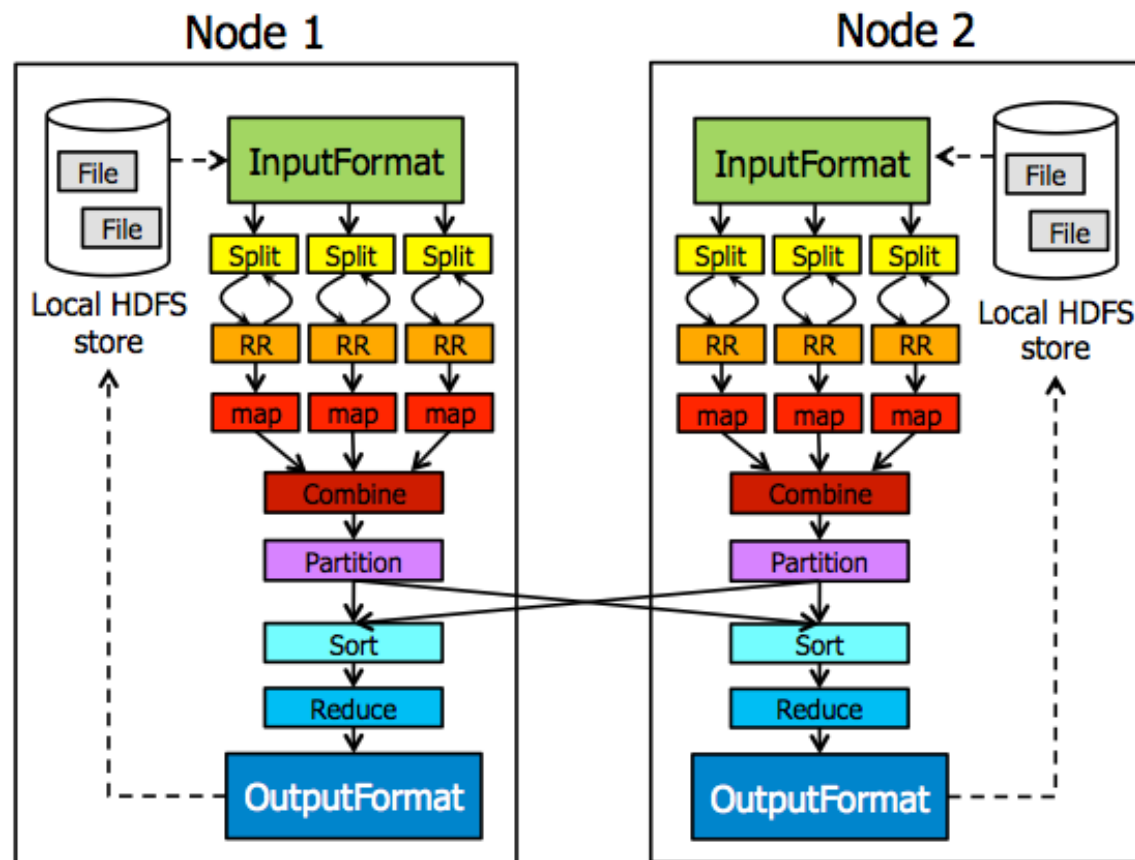
# MapReduce

$\langle \text{key1}, \text{value1} \rangle^* \rightarrow \text{map} \rightarrow \langle \text{key2}, \text{value2} \rangle^*$

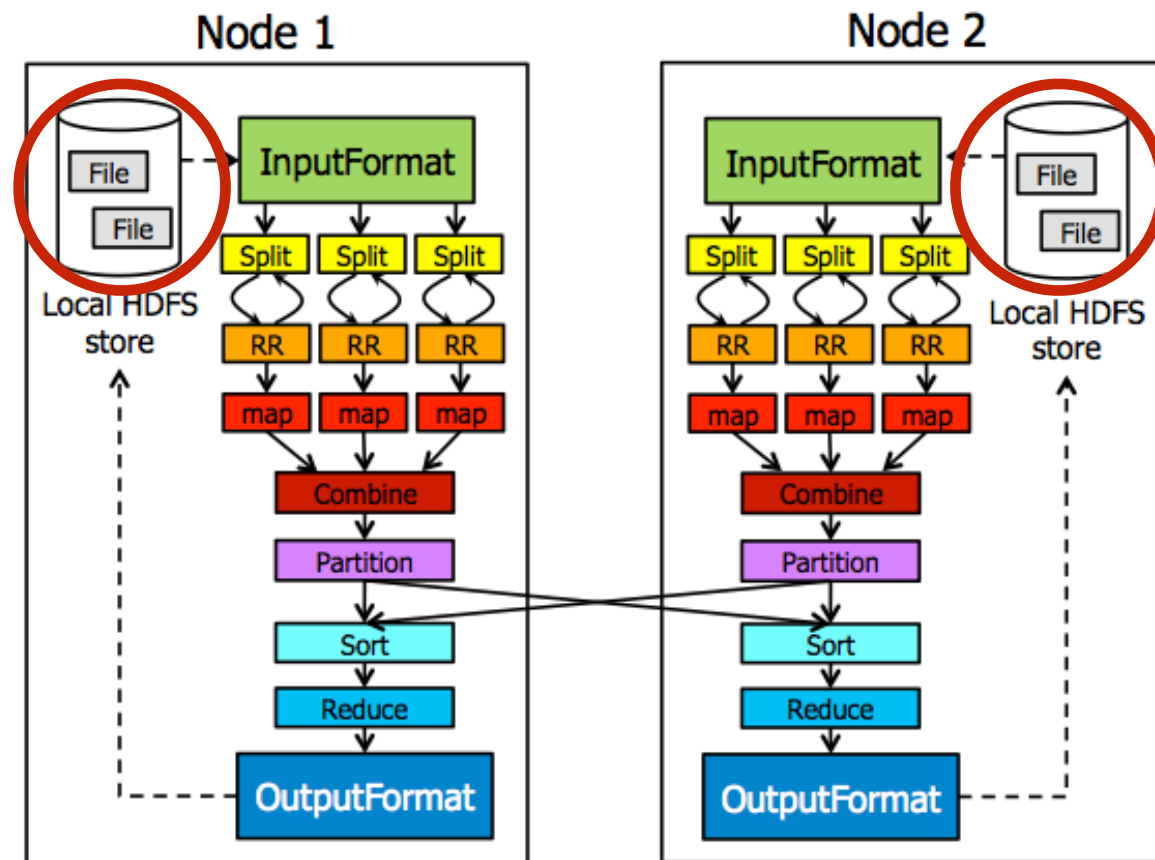
$\langle \text{key2}, \text{value2}^* \rangle^* \rightarrow \text{reduce} \rightarrow \langle \text{key3}, \text{value3} \rangle^*$

- Usually,  $\text{key1} = \text{line number}$ ,  $\text{value1} = \text{line}$ .
- $\langle \text{key3}, \text{value3} \rangle$  will be written on a single line in an output file.
- What really matters is  $\text{key2}$ : the reduce key.

# Hadoop Data Flow



# Input / Output

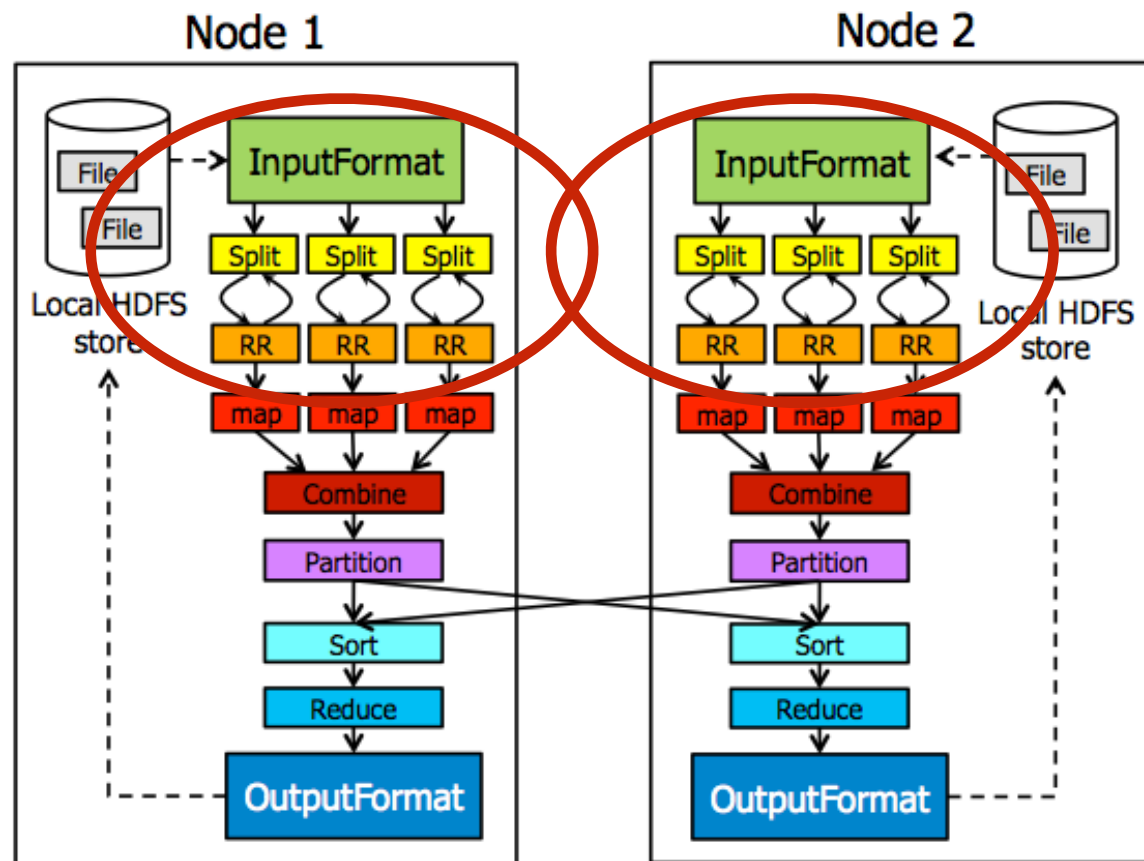




# Input / Output

- The input is a set of files in an input directory.
- The output directory must not exist (will be created by Hadoop).

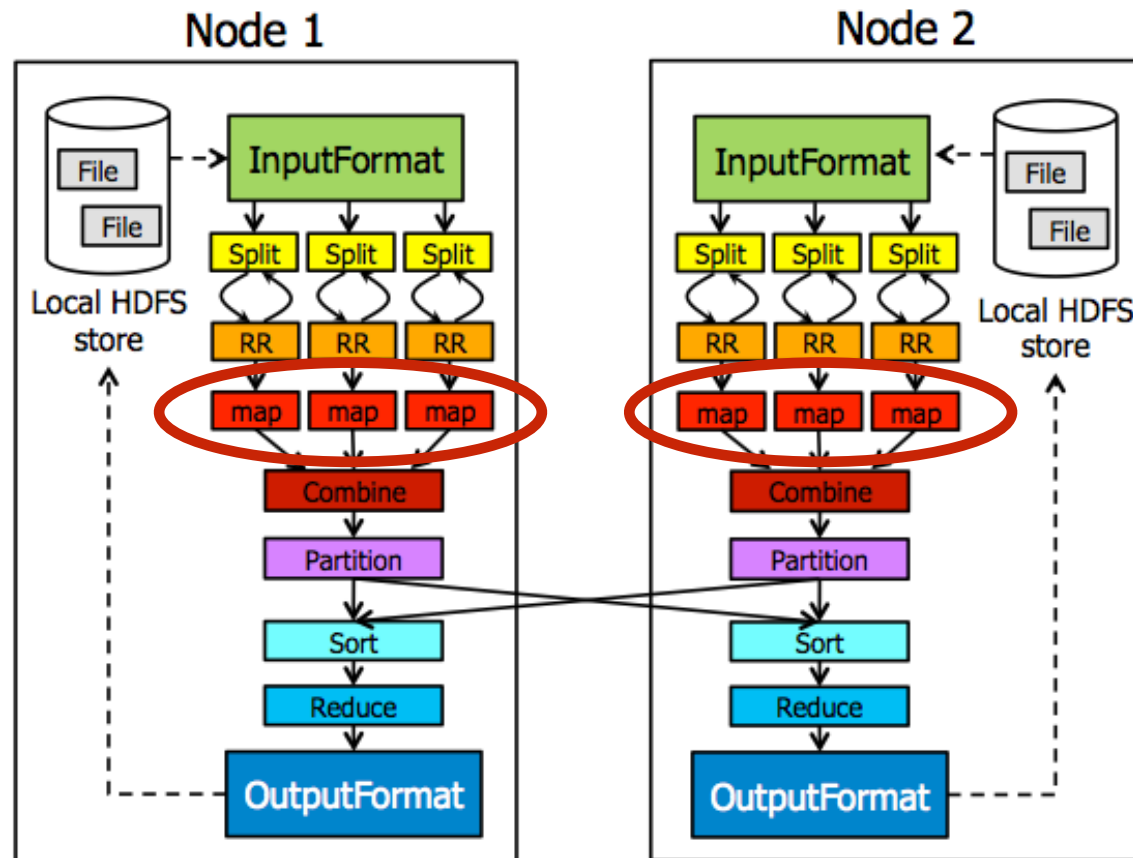
# Input Split & Record Readers



# 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



# Mapper

```
class MyMapper      key1      value1 key2 value2
extends Mapper<LongWritable, Text, Text, Text>
{
    @Override protected
    void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException
    {
        . . .      write a new <key2, value2> pair (can be called multiple times)
        context.write(new Text("foo"), value);
        . . .
    }
}
```

# 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);
    }

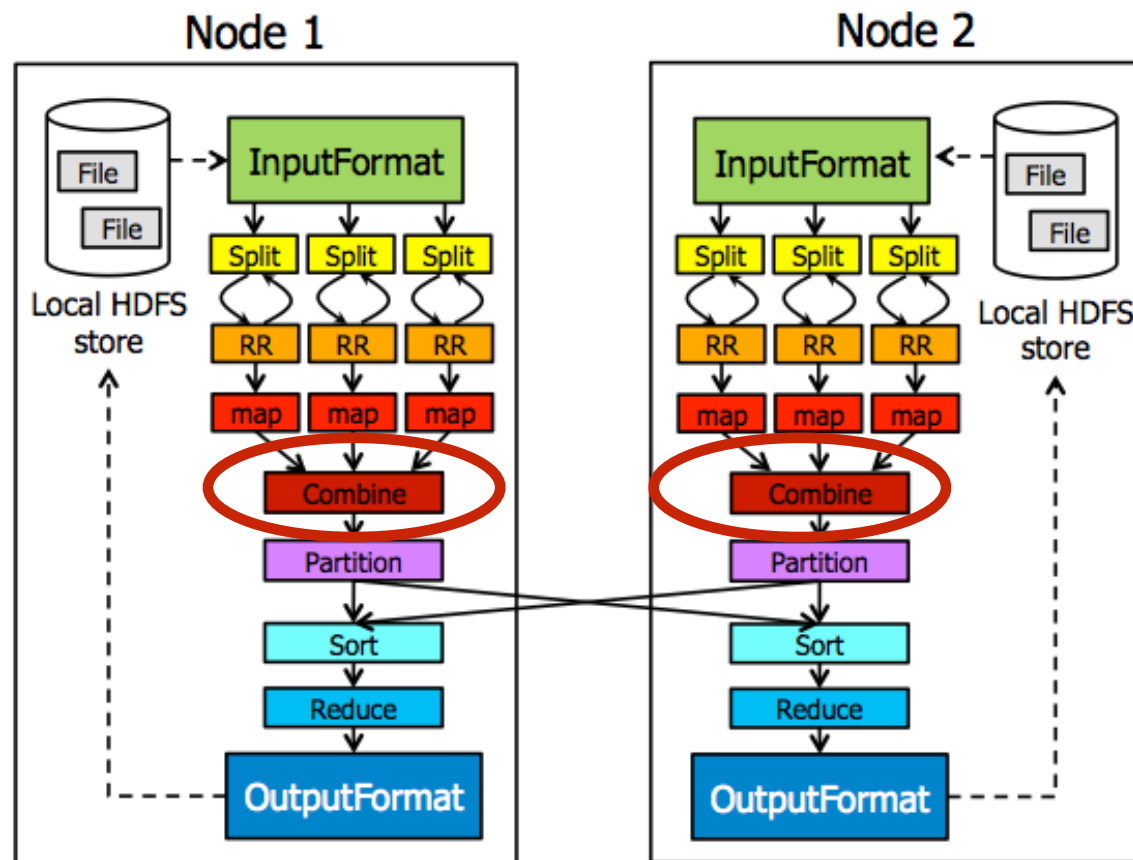
    @Override public void readFields(DataInput in) throws IOException {
        word.readFields(in);
        count.readFields(in);
    }

    @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());
    }
}
```



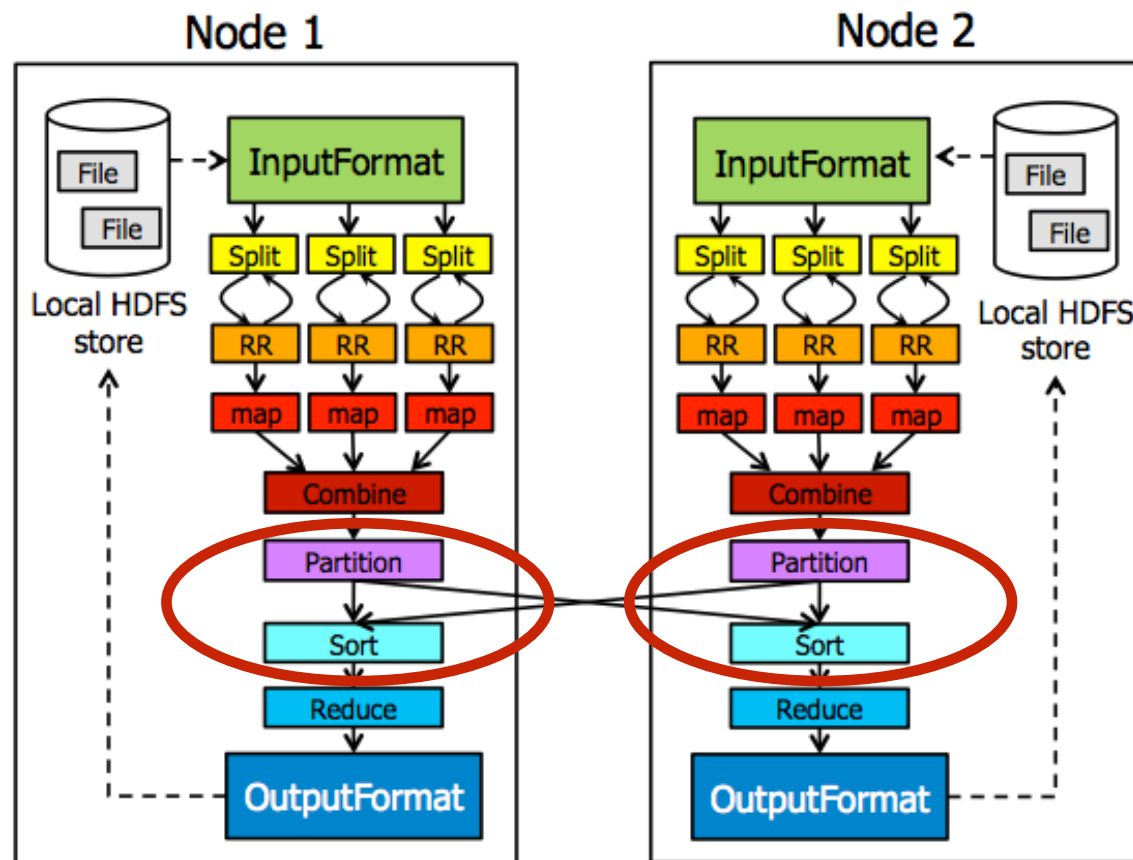
# Combiner



# 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.

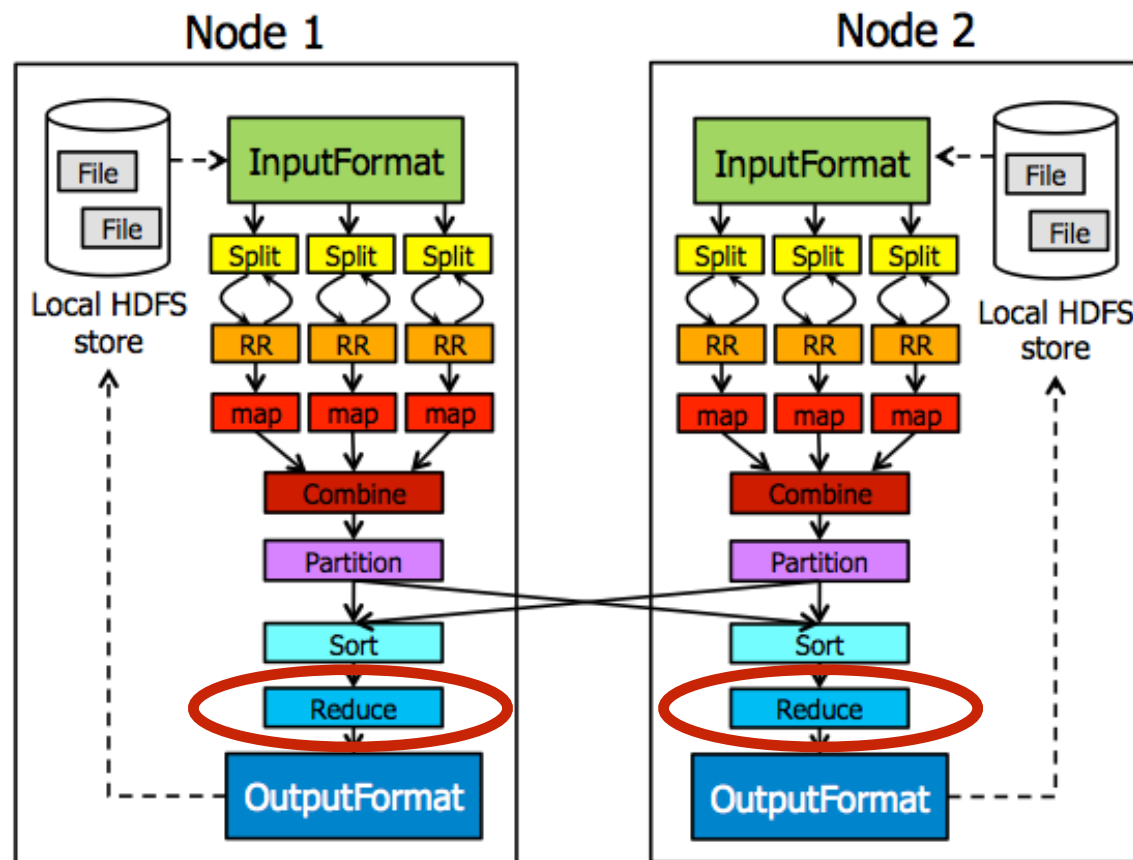
# Partitioning & Sorting



# 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

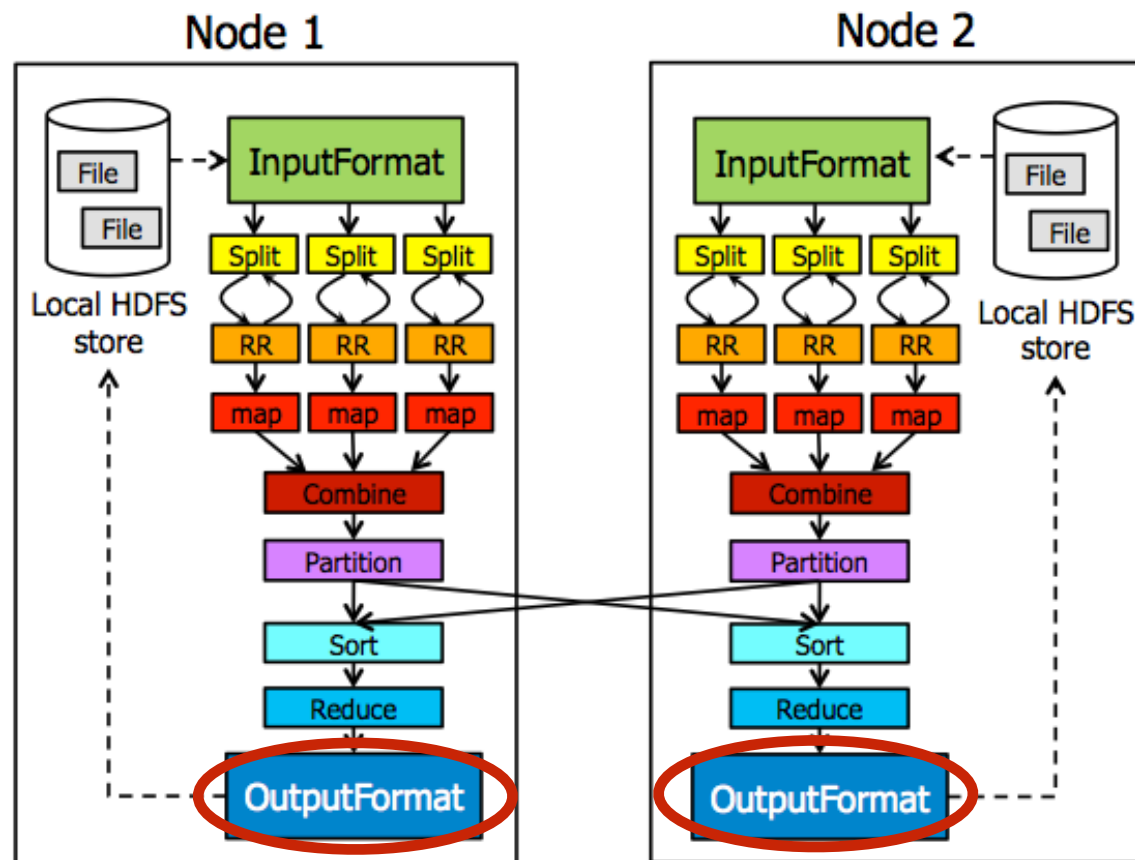
# Reducer



# 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**

# AWS Demo: Using Elastic MapReduce

# Exercise: Word Counting