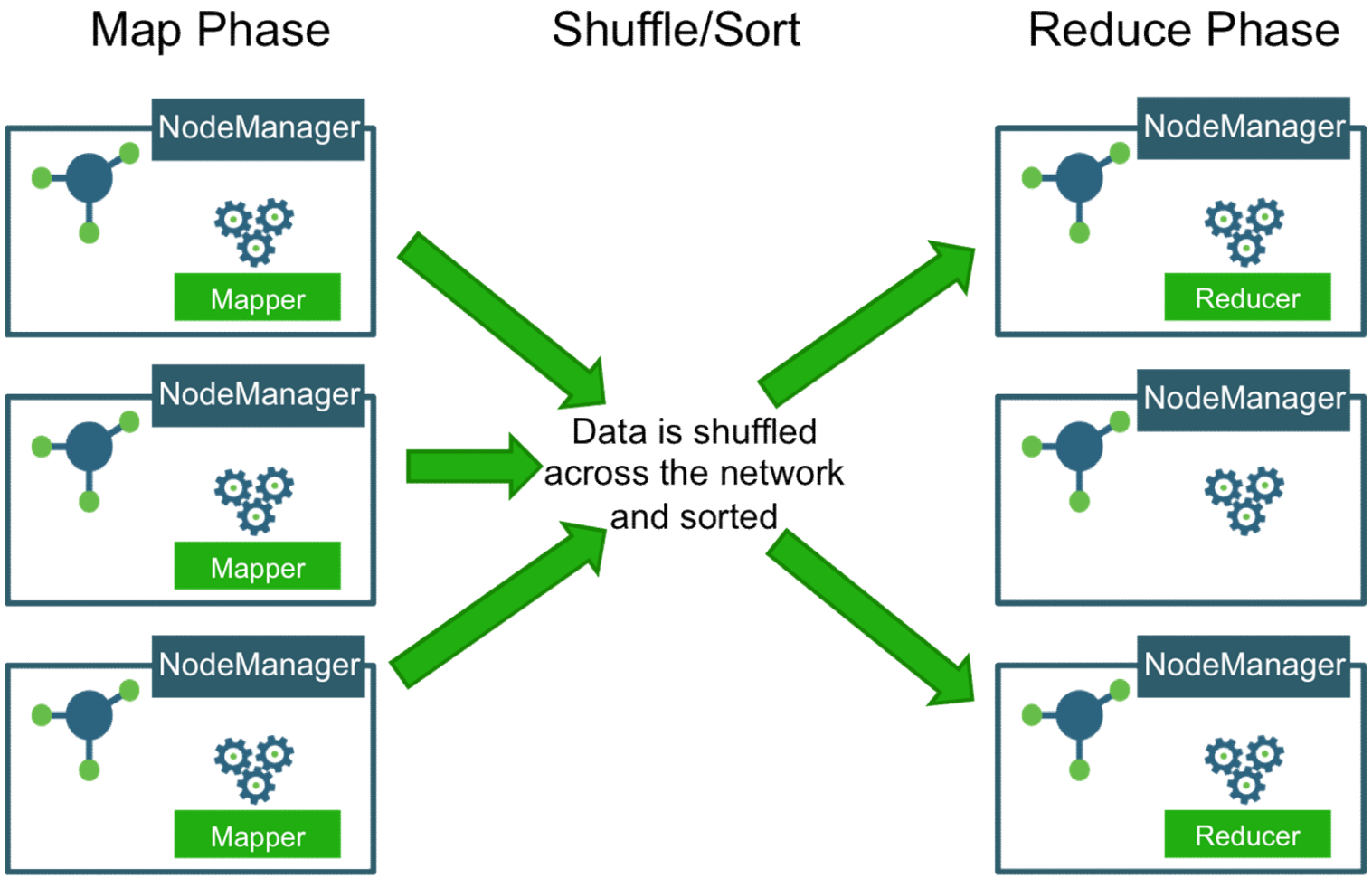
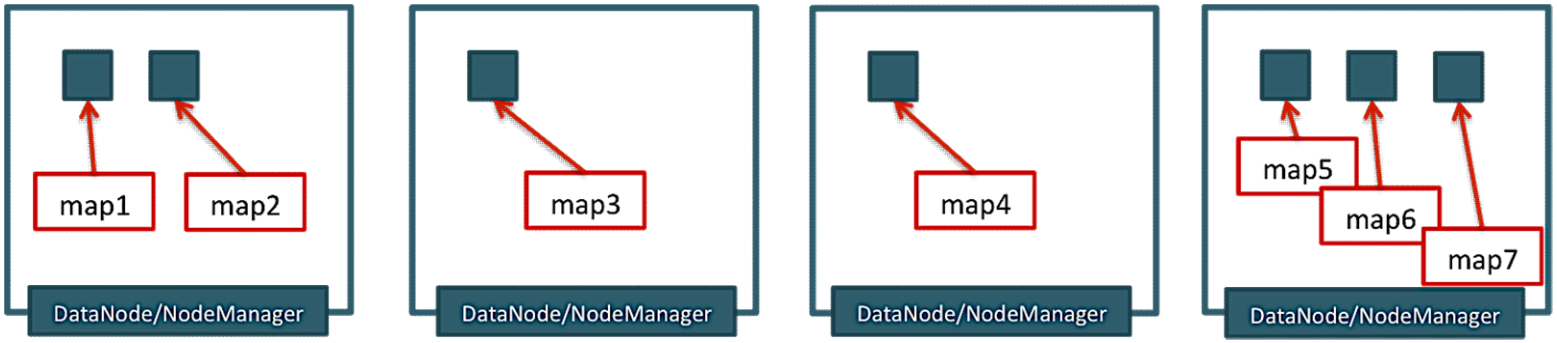
Writing MapReduce Applications


Map Phase
Shuffle/Sort
Reduce Phase
NodeManager
Mapper
NodeManager
Mapper
NodeManager
Mapper
Data is shuffled
across the network
and sorted
NodeManager
Reducer
NodeManager
NodeManager
Reducer

Understanding MapReduce

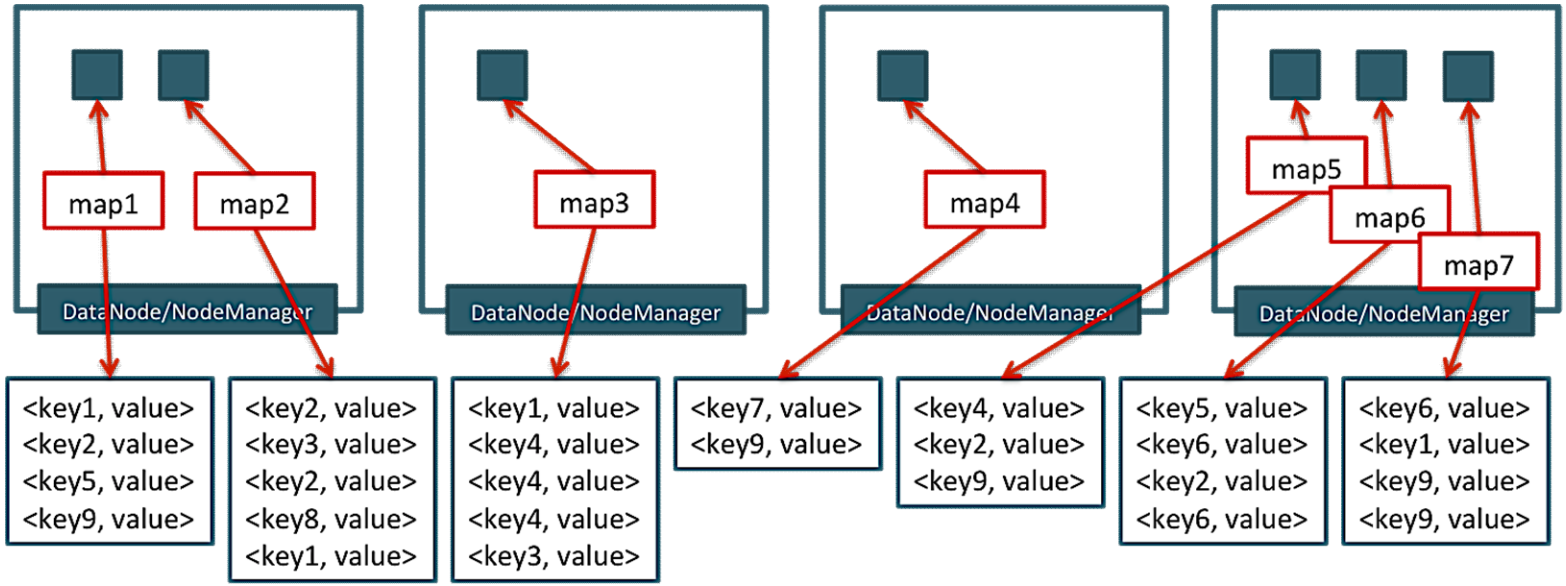
1. Suppose a file is the input to a MapReduce job. That file is broken down into blocks stored on DataNodes across the Hadoop cluster.



2. During the Map phase, map tasks process the input of the MapReduce job, with a map task assigned to each Input Split. The map tasks are Java processes that ideally run on the DataNodes where the blocks are stored.

Understanding MapReduce - cont.

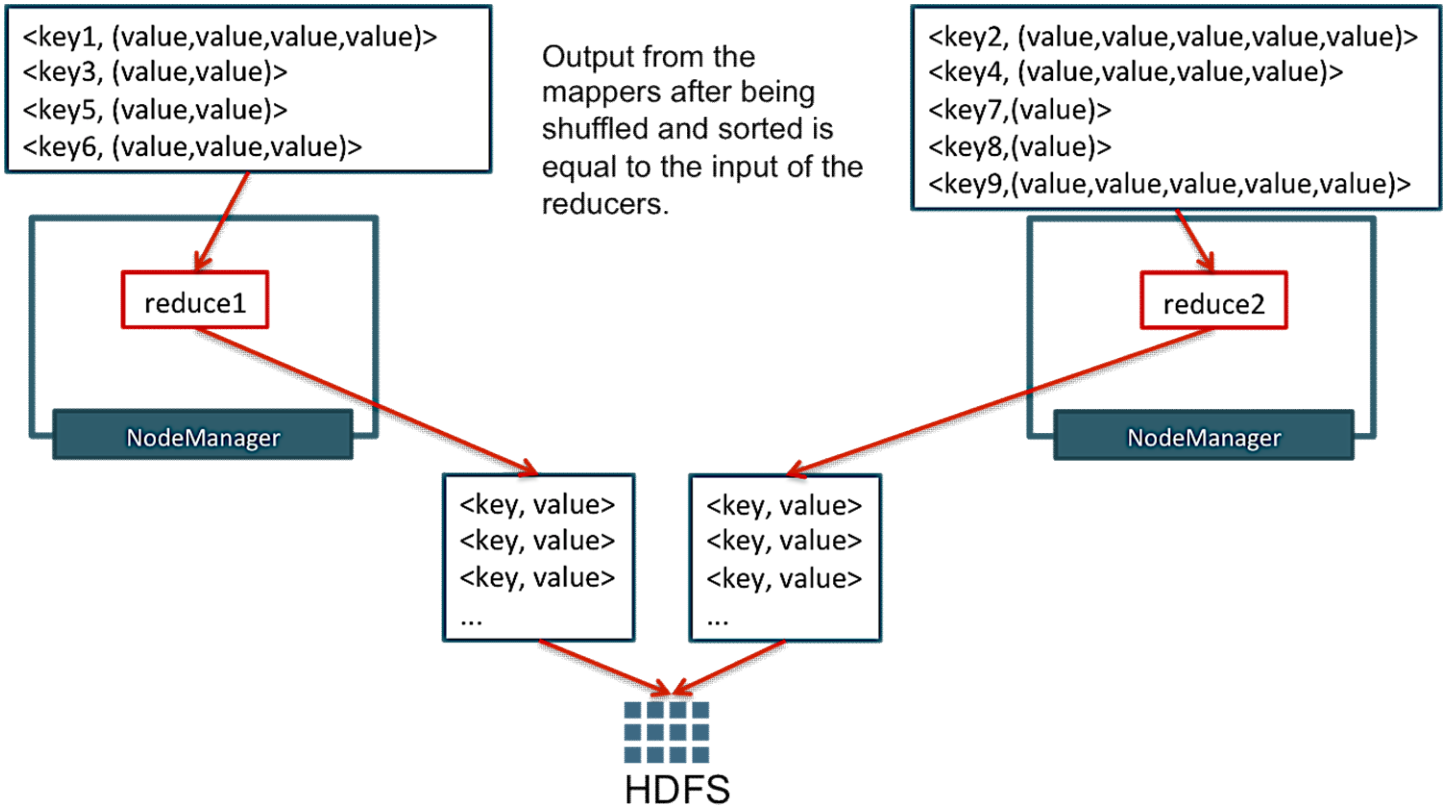
3. Each map task processes its Input Split and outputs records of <key, value> pairs.



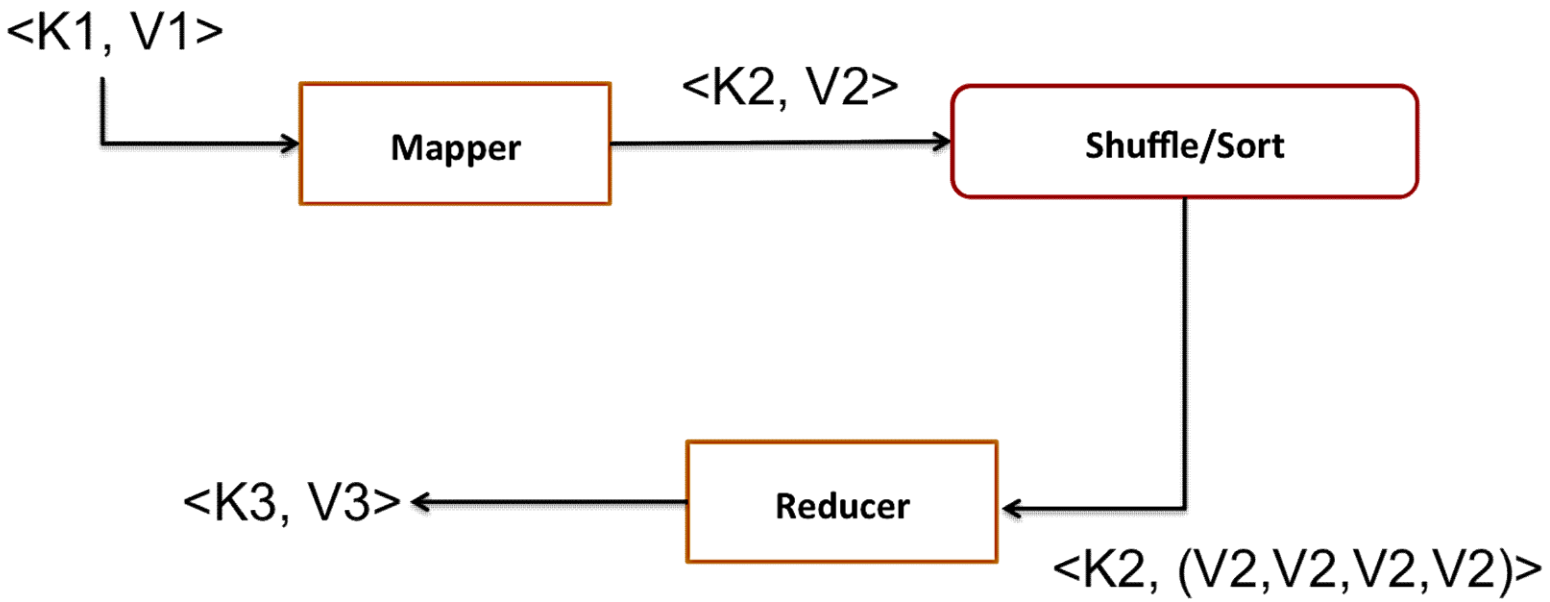
4. The <key,value> pairs go through a shuffle/sort phase, where records with the same key end up at the same reducer. The specific pairs sent to a reducer are sorted by key, and the values are aggregated into a collection.

Understanding MapReduce - cont.

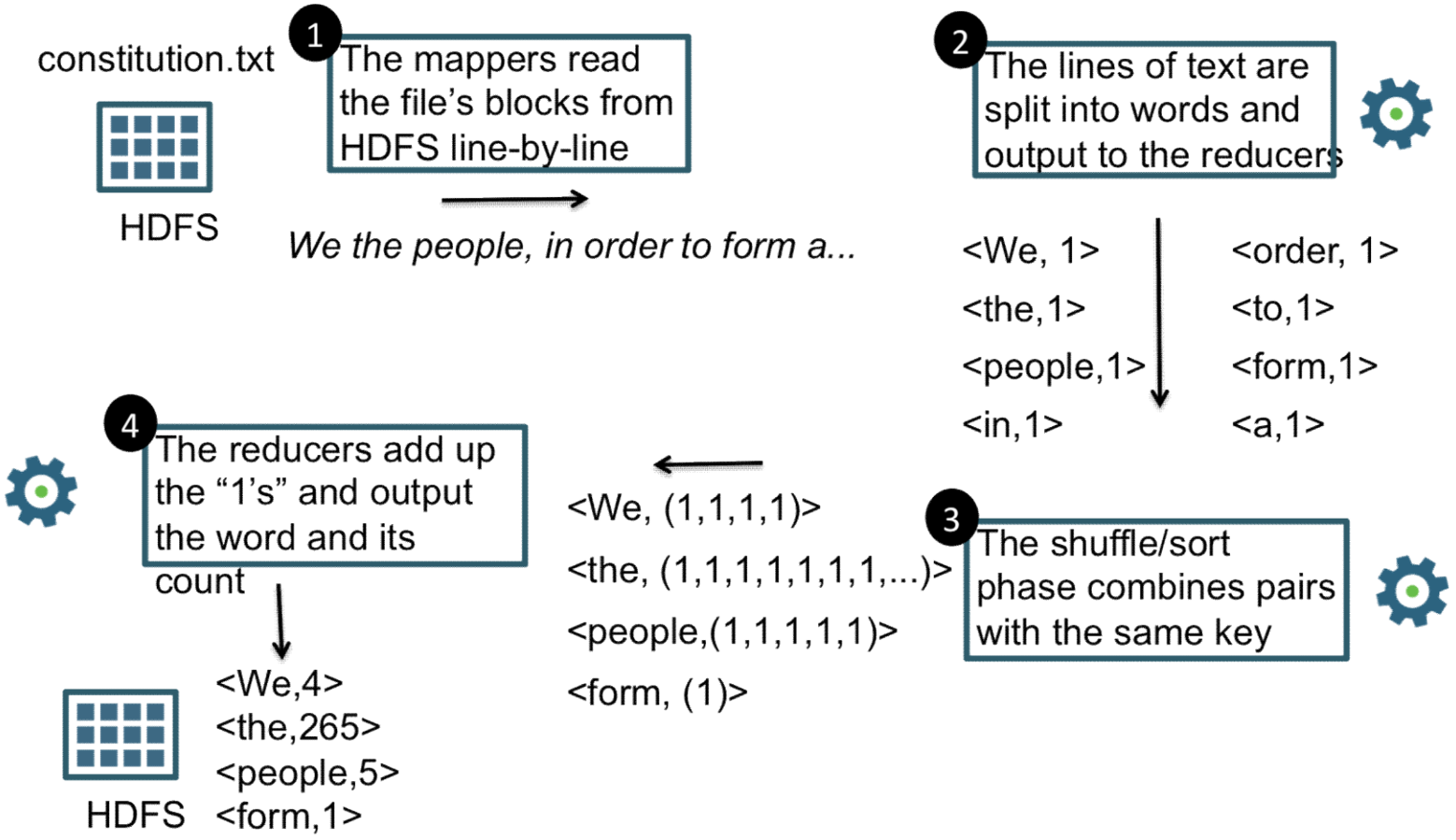
5. Reduce tasks run on a NodeManager as a Java process. Each Reducer processes its input and outputs <key,value> pairs that are typically written to a file in HDFS.



The Key/Value Pairs of MapReduce



WordCount in MapReduce



Demonstration

Understanding MapReduce

WordCountMapper

```
public class WordCountMapper
   extends Mapper<LongWritable, Text, Text, IntWritable> {

        @Override
     protected void map(LongWritable key, Text value,
                                          Context context)
        throws IOException, InterruptedException {
      String currentLine = value.toString();
      String [] words = currentLine.split(" ");
      for(String word : words) {
         Text outputKey = new Text(word);
         context.write(outputKey, new IntWritable(1));
      }
     }
}
```

WordCountReducer

```
public class WordCountReducer
       extends Reducer<Text, IntWritable, Text,
IntWritable> {

    @Override
    protected void reduce(Text key, Iterable<IntWritable>
values,
                        Context context)
           throws IOException, InterruptedException {
        int sum = 0;
        for(IntWritable count : values) {
            sum += count.get();
        }
        IntWritable outputValue = new IntWritable(sum);
        context.write(key, outputValue);
    }
}
```

WordCountJob

```
Job job = Job.getInstance (getConf(), "WordCountJob");
Configuration conf = job.getConfiguration();
job.setJarByClass(getClass());

Path in = new Path(args[0]);
Path out = new Path(args[1]);
FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job,  out);
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
return job.waitForCompletion(true)?0:1;
```

Running a MapReduce Job

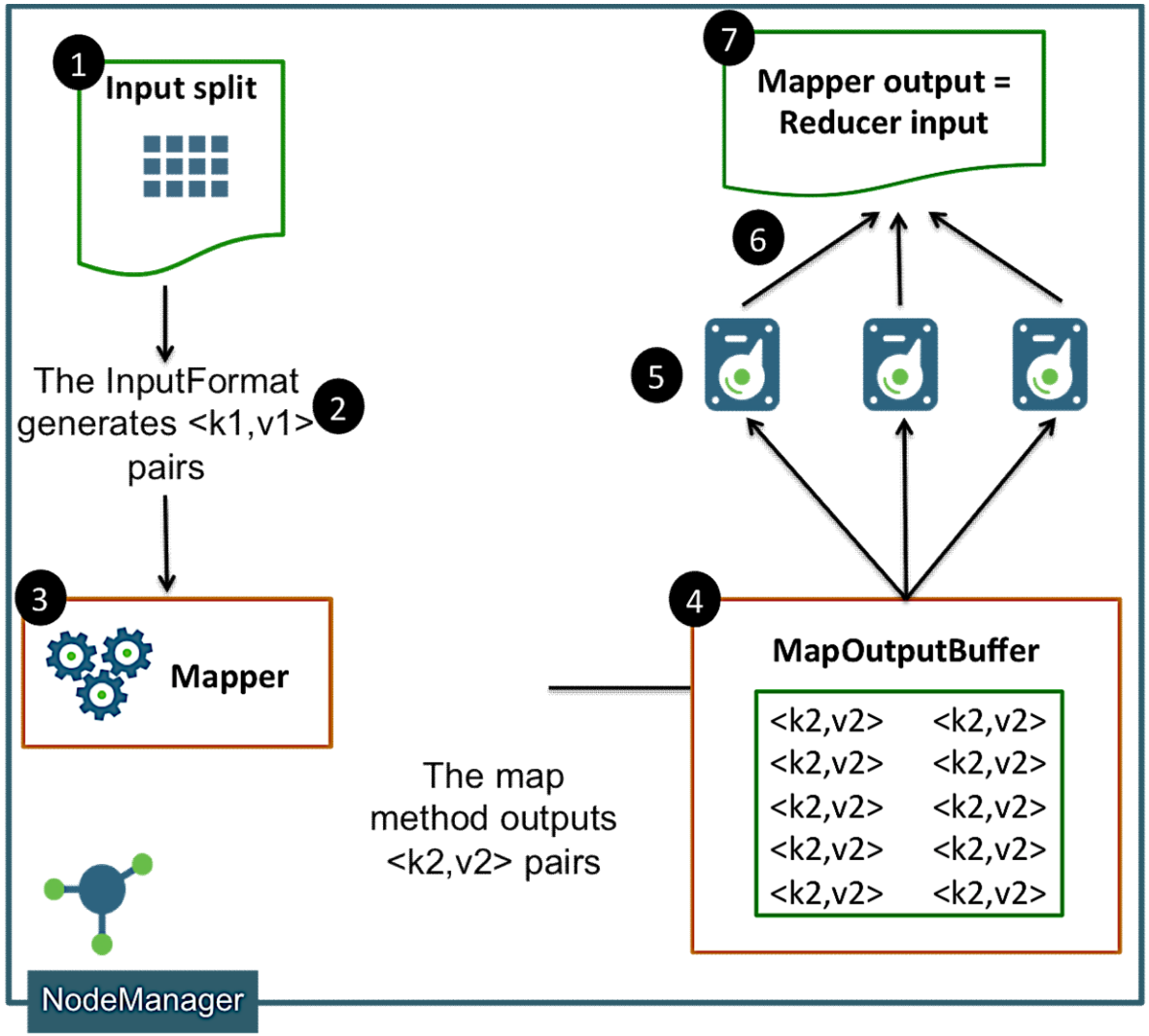
To run a job, perform the following steps:

- Put the input files into HDFS
- If the output directory exists, delete it
- Use **hadoop** to execute the job
- View the output files

```
yarn jar wordcount.jar
my.WordCountJob input/file.txt result
```

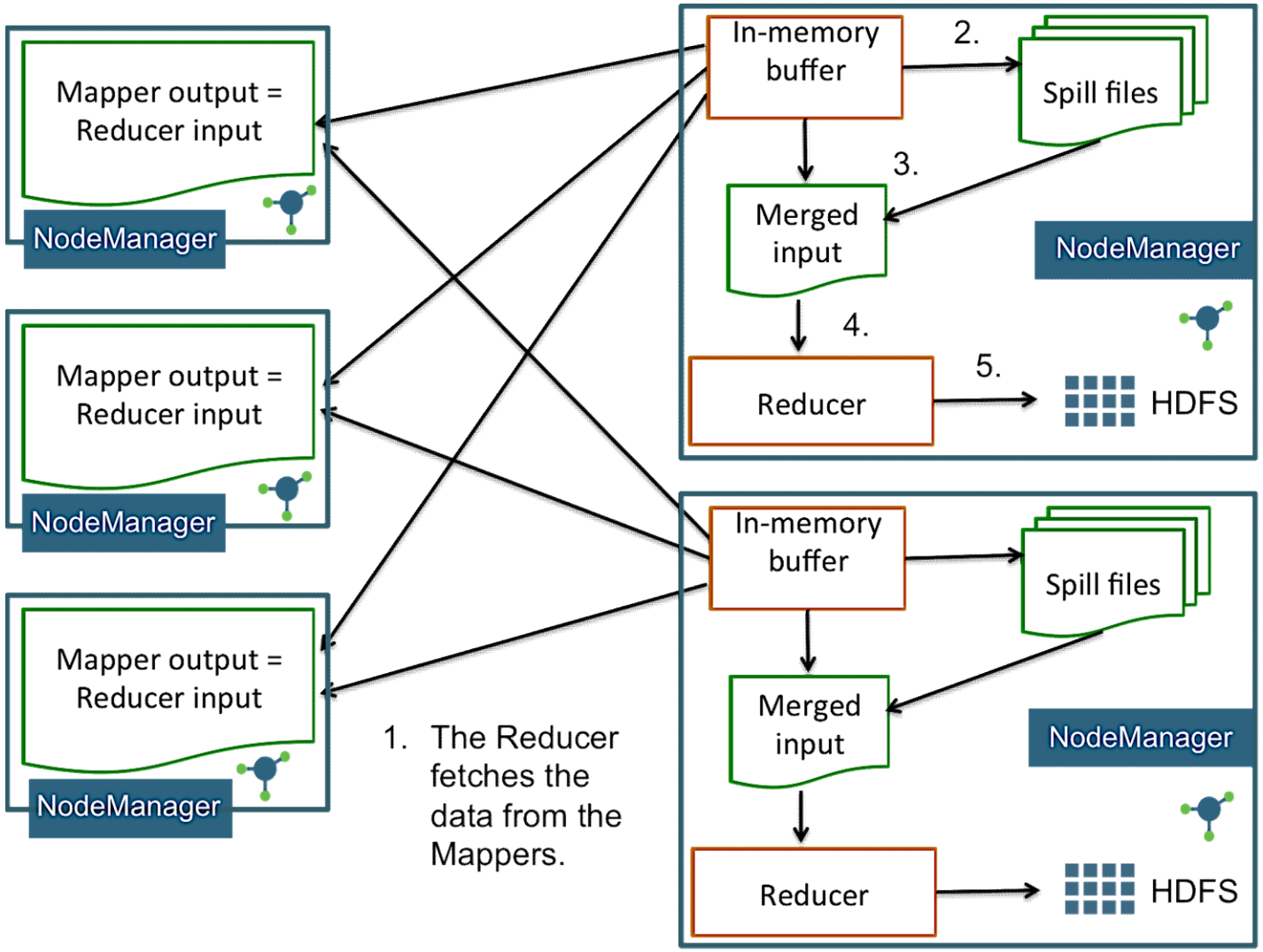
Hands-On

Word Count



Spill files are merged into a single file

Records are sorted and spilled to disk when the buffer reaches a threshold


Mapper output = Reducer input
NodeManager
Mapper output = Reducer input
NodeManager
Mapper output = Reducer input
NodeManager
In-memory buffer
2.
Spill files
3.
Merged input
NodeManager
4.
Reducer
5.
HDFS
In-memory buffer
Spill files
Merged input
NodeManager
Reducer
HDFS
1. The Reducer fetches the data from the Mappers.

About YARN

YARN = Yet Another Resource Negotiator

YARN splits up the functionality of the JobTracker in Hadoop 1.x into two separate processes:

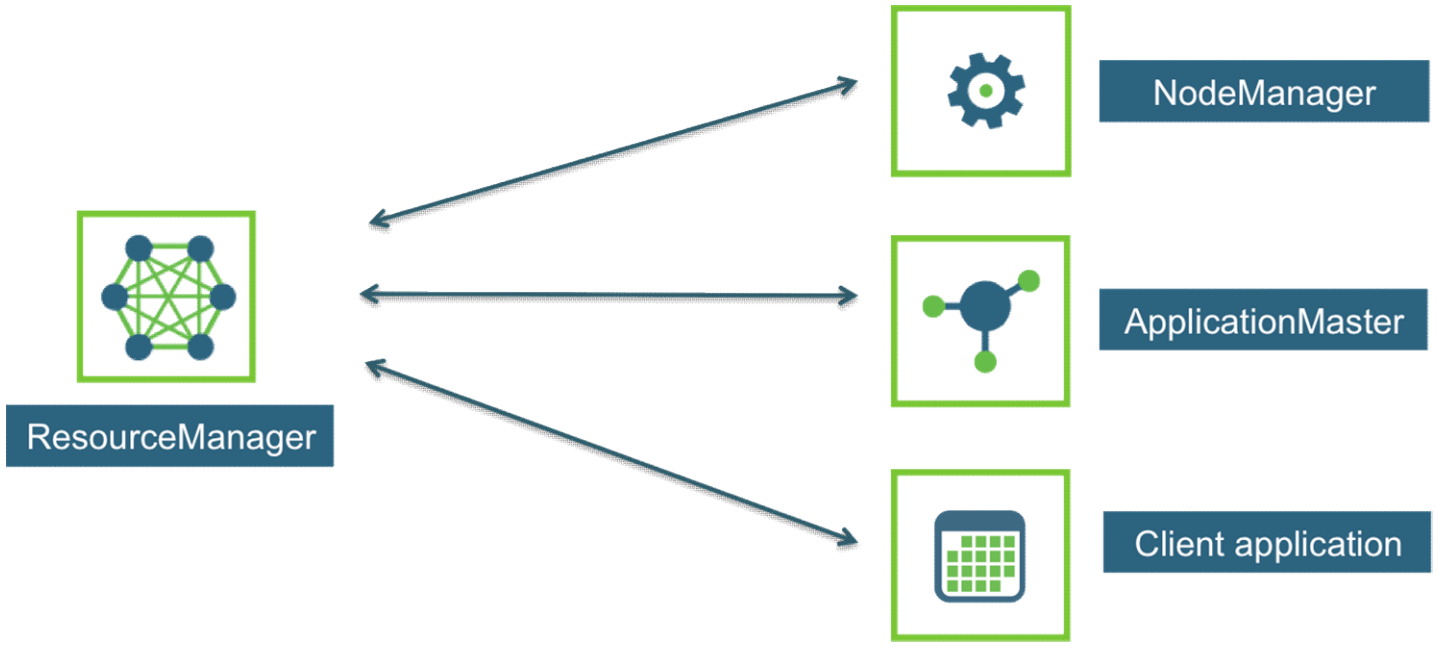
- **ResourceManager**: for allocating resources and scheduling applications
- **ApplicationMaster**: for executing applications and providing failover

Open-Source YARN Use Cases

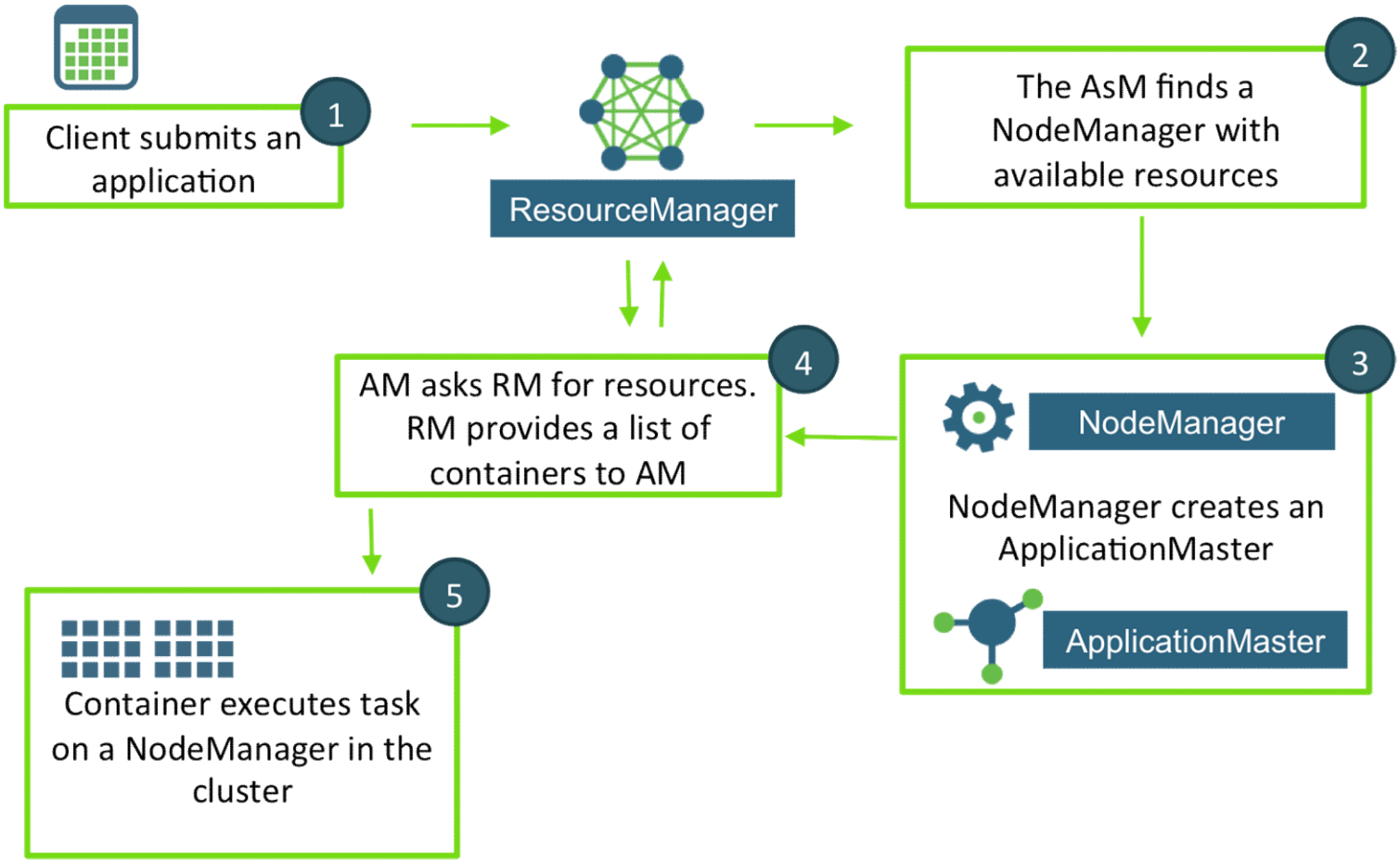
- **Tez:** improves the execution of MapReduce jobs
- **Slider:** for deploying existing distributed applications onto YARN
- **Storm**: for real-time computing
- **Spark**: a MapReduce-like cluster computing framework designed for low-latency iterative jobs and interactive use from an interpreter
- **Open MPI**: a high-performance Message Passing Library that implements MPI-2
- **Apache Giraph**: a graph processing platform

The Components of YARN

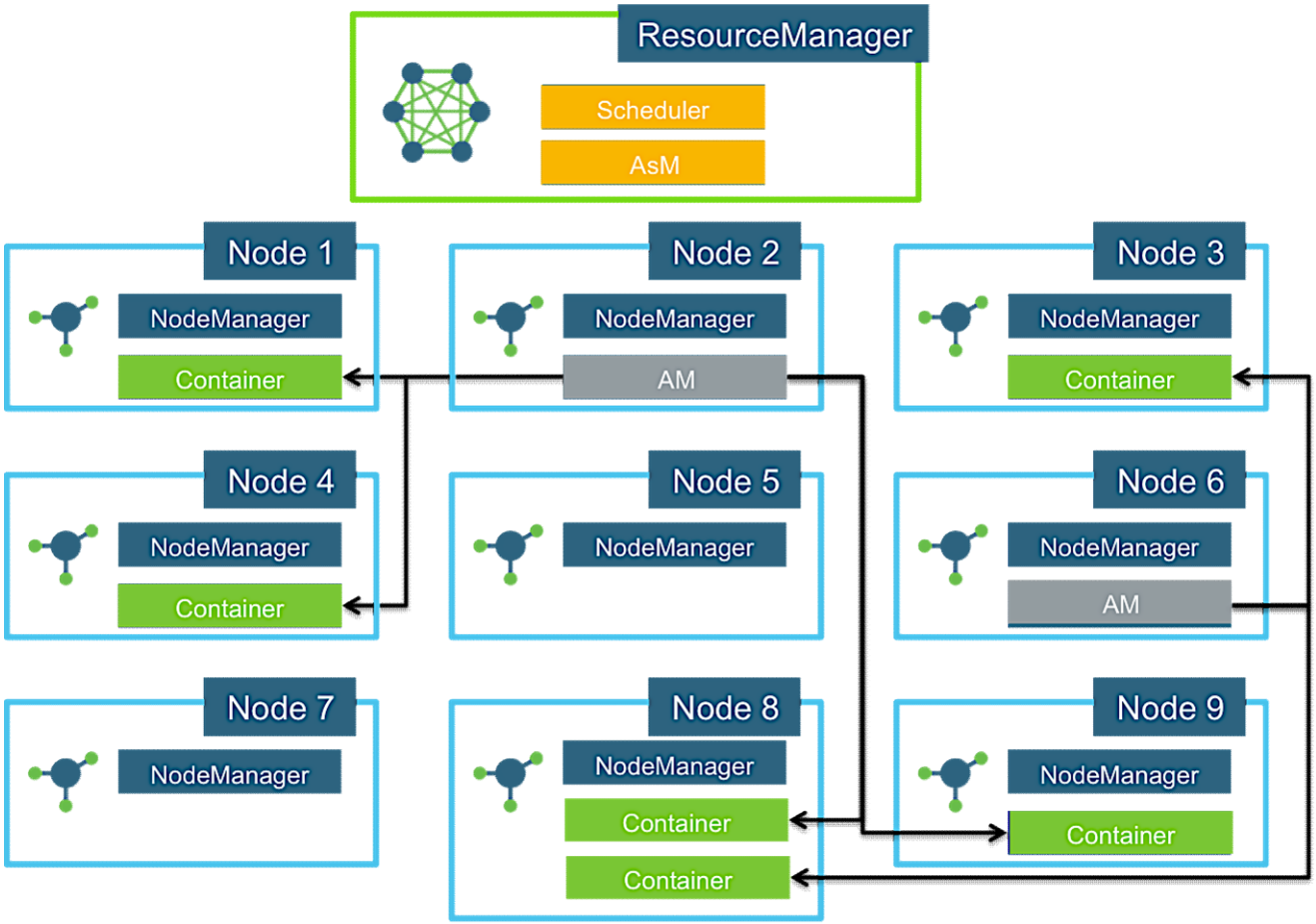
The **ResourceManager** communicates with the **NodeManagers**, **ApplicationMasters**, and **Client** applications.

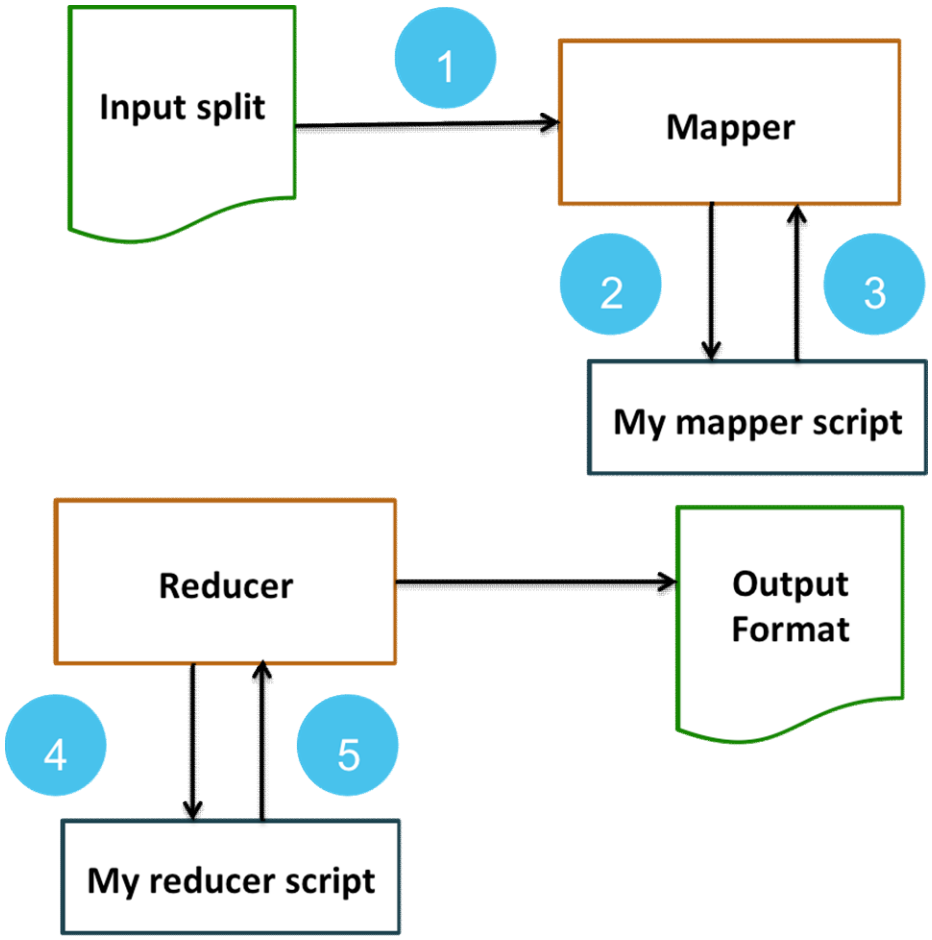


Lifecycle of a YARN Application



A Cluster View Example





1. The InputFormat generates <k1,v1> pairs
2. Converts the <k1,v1> pairs to lines and sends them to the stdin of the process
3. The stdout of the process is converted into <k2,v2> pairs
4. Converts <k2,(v2,v2,...)> pairs to lines and send them to the stdin of the process
5. The stdout of the process is converted into <k3, v3> pairs

Running a Hadoop Streaming Job

A Streaming job is a MapReduce job defined in the hadoop-streaming.jar file:

```
hadoop  jar $HADOOP_HOME/lib/hadoop-streaming.jar
    -input input_directories
    -output output_directories
    -mapper mapper_script
    -reducer reducer_script
```

Review

1. What are the three main phases of a MapReduce job?

2. Suppose the Mappers of a MapReduce job output <key,value> pairs that are of type <integer,string>. What will the pairs that are processed by the corresponding Reducers look like?

3. What happens if all the <key,value> pairs output by a Mapper do not fit into the memory of the Mapper?

4. What determines the number of Mappers of a MapReduce job?

5. What determines the number of Reducers of a MapReduce job?

6. **True or False**: The shuffle/sort phase sorts the keys and values as they are passed to the Reducer.

7. What are the four main components of YARN?

8. What happens if a container fails to complete its task in a YARN application?

Hands-On

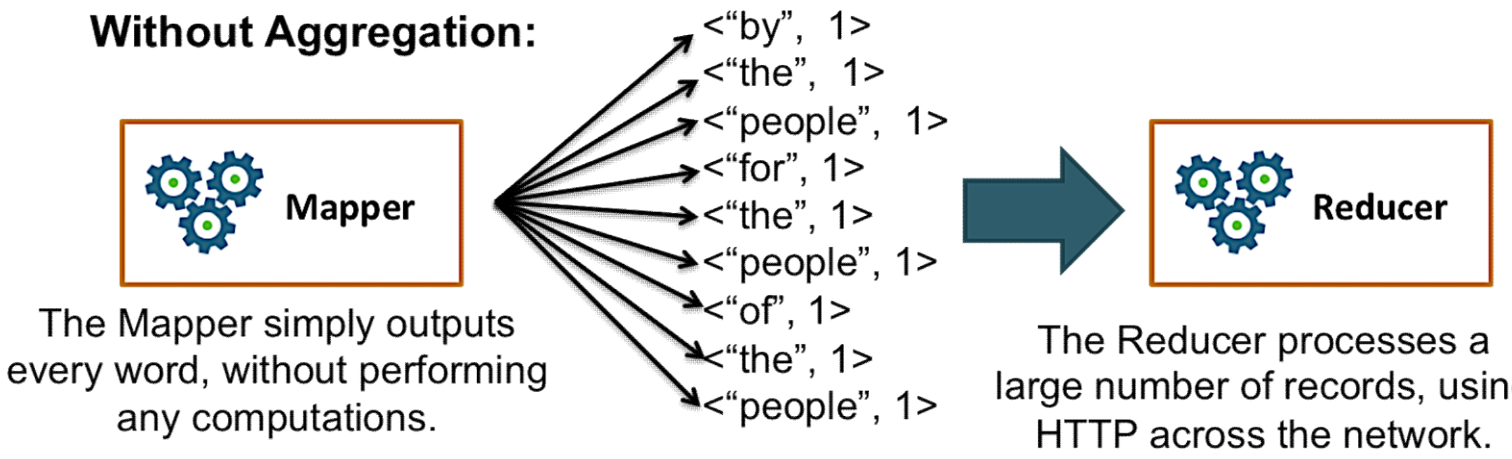
Distributed Grep

Hands-On

Inverted Index

Map Aggregation

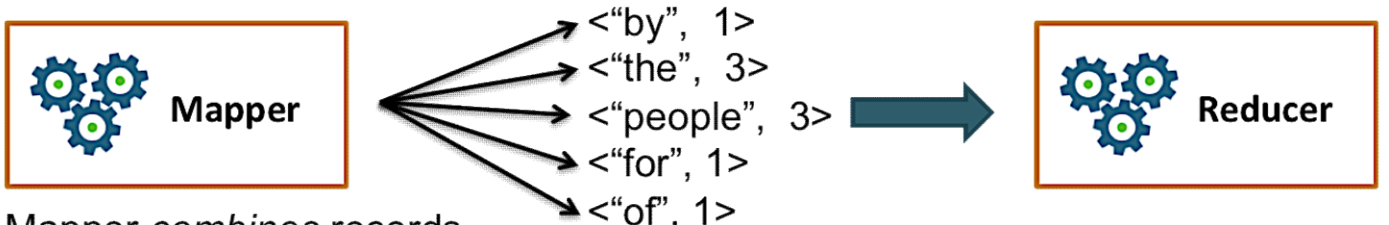
Without Aggregation:



The Mapper simply outputs every word, without performing any computations.

The Reducer processes a large number of records, using HTTP across the network.

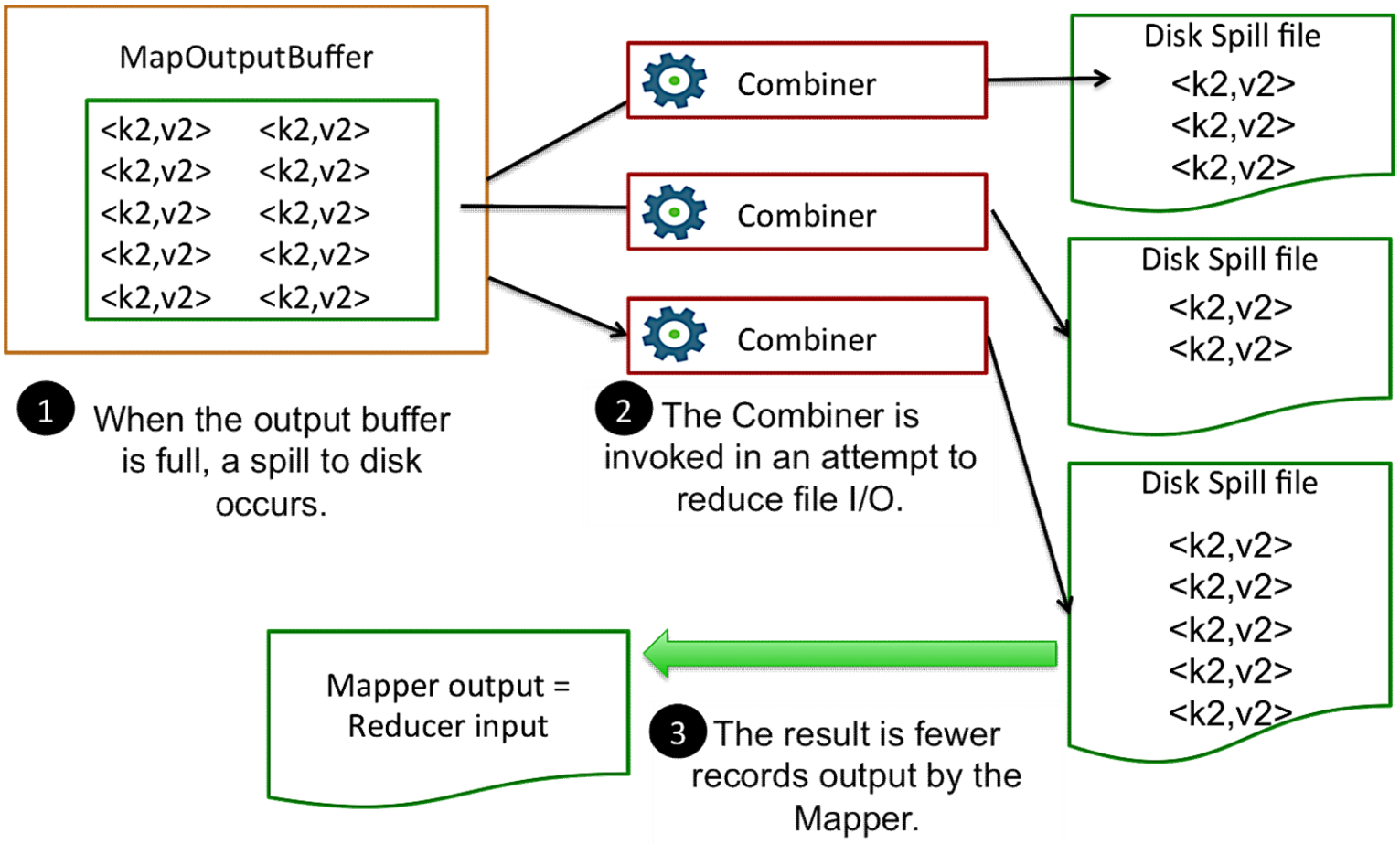
With Aggregation:



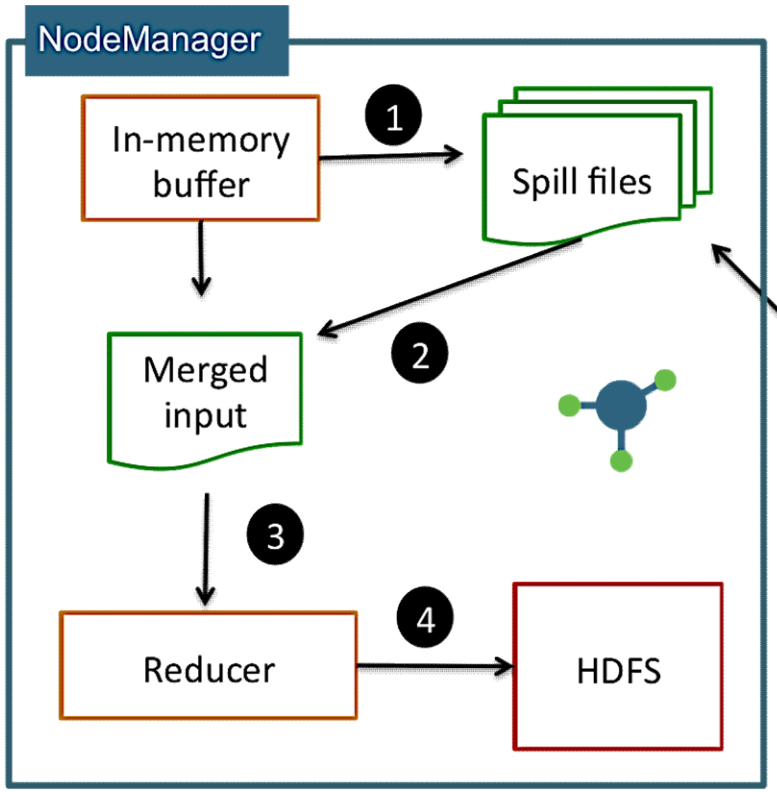
The Mapper *combines* records in a manner that does not affect the algorithm.

Expensive network traffic is decreased.

Overview of Combiners



Reduce-side Combining



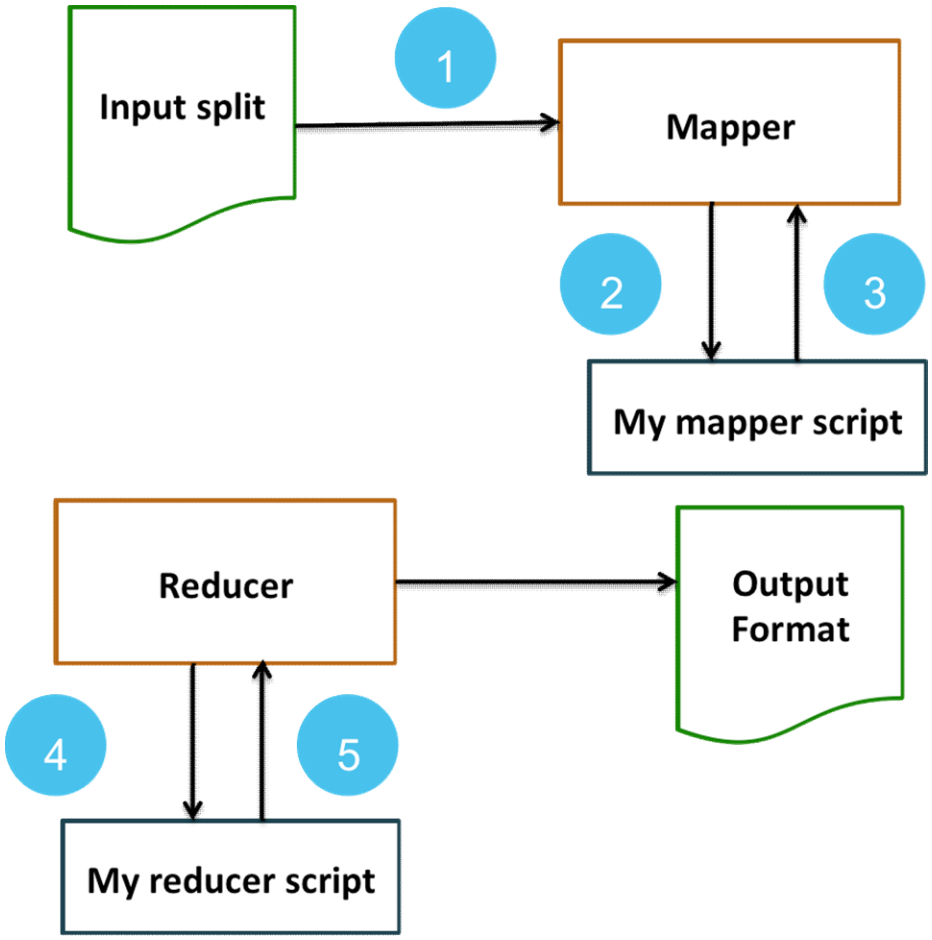
The combiner is also used in the reduce phase if the intermediate <key,value> pairs from Mappers are spilled to disk.

Example of a Combiner

```
public class WordCountCombiner
    extends Reducer<Text, IntWritable, Text, IntWritable> {

          private IntWritable outputValue = new IntWritable();

    @Override
    protected void reduce(Text key,
            Iterable<IntWritable> values, Context context)
           throws IOException, InterruptedException {
        int sum = 0;
        for(IntWritable count : values) {
           sum += count.get();
        }
        outputValue.set(sum);
        context.write(key, outputValue);
    }
}
```



1. The InputFormat generates <k1,v1> pairs
2. Converts the <k1,v1> pairs to lines and sends them to the stdin of the process
3. The stdout of the process is converted into <k2,v2> pairs
4. Converts <k2,(v2,v2,...)> pairs to lines and send them to the stdin of the process
5. The stdout of the process is converted into <k3, v3> pairs

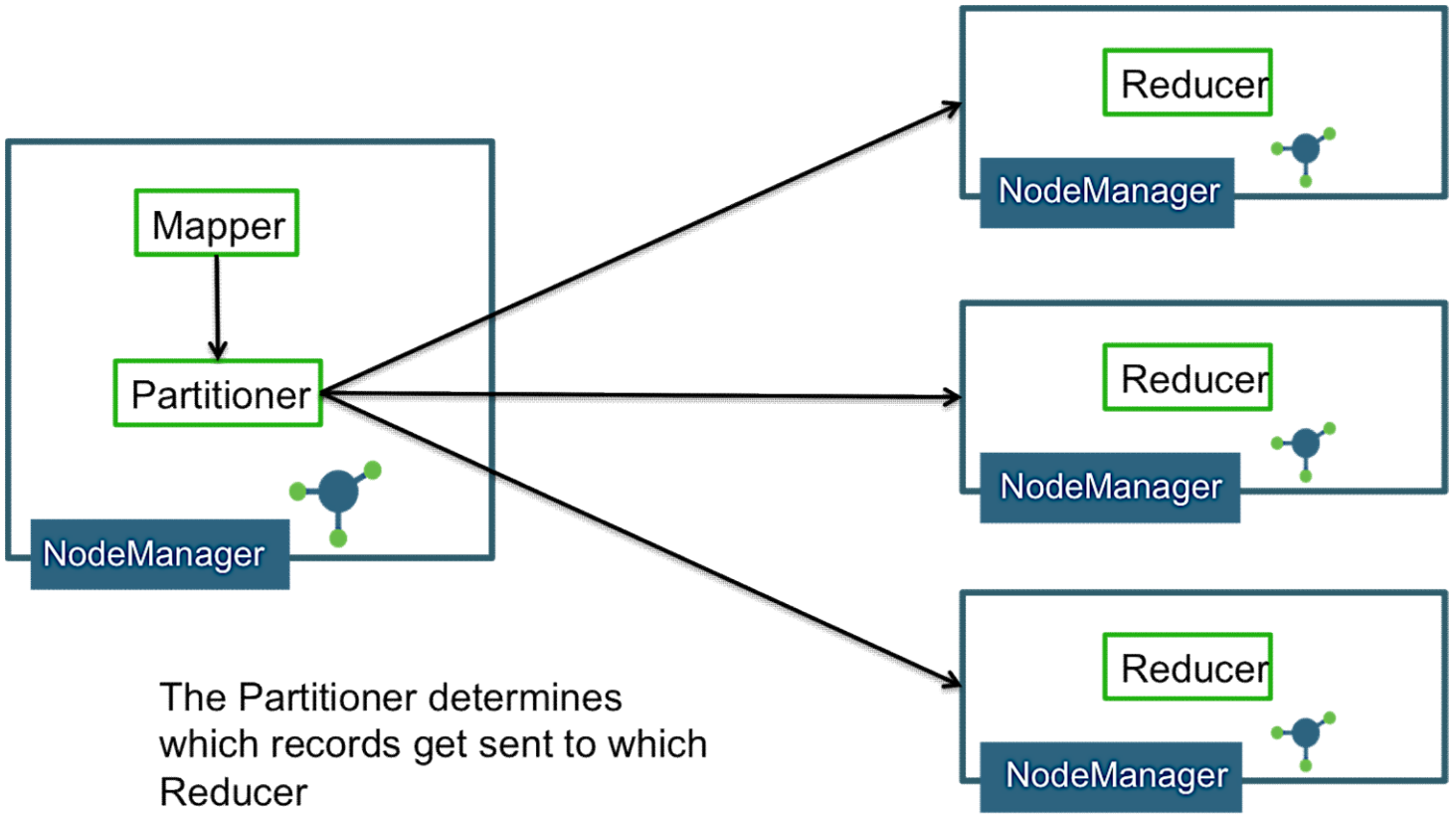
Running a Hadoop Streaming Job

A Streaming job is a MapReduce job defined in the hadoop-streaming.jar file:

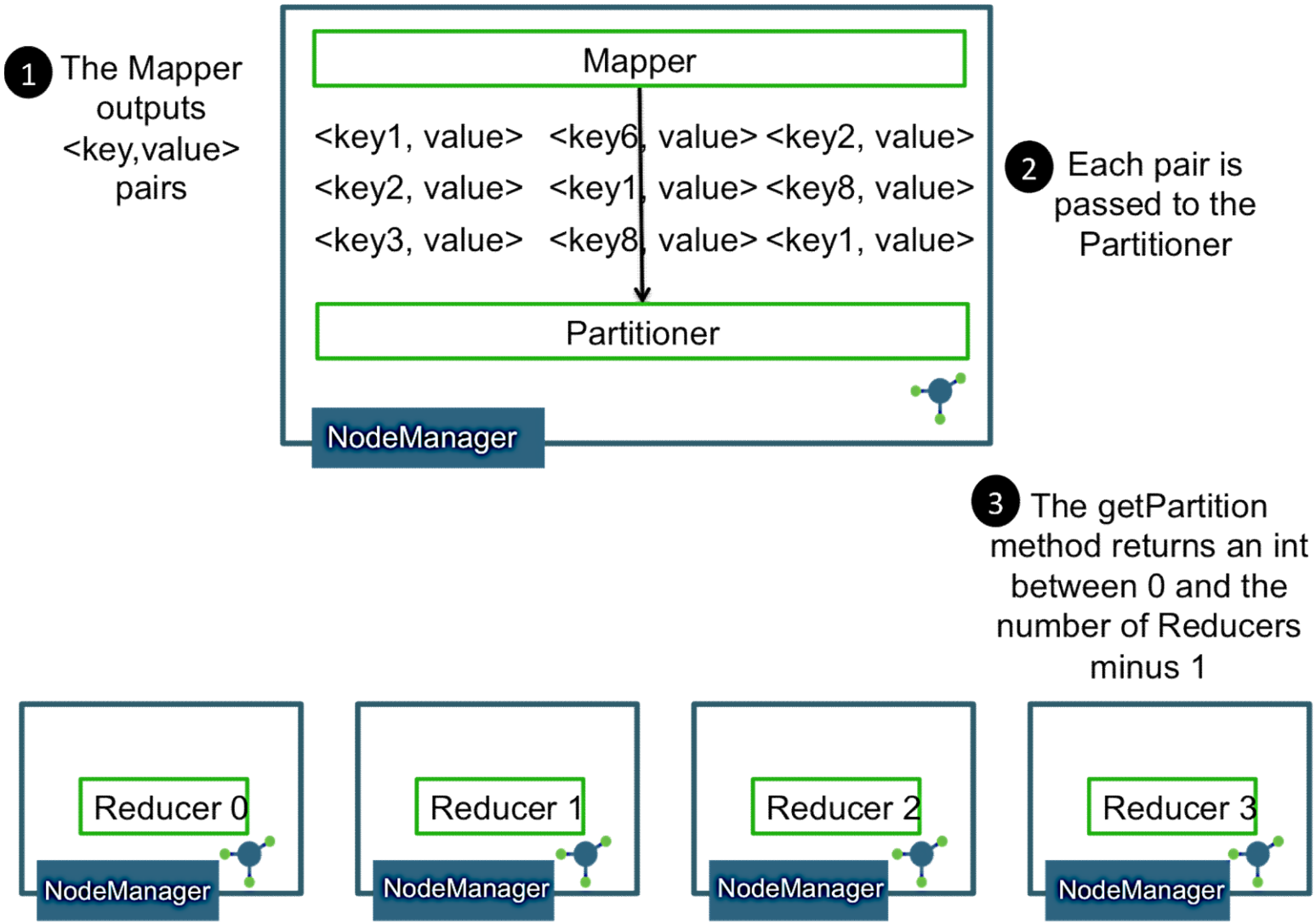
```
hadoop  jar $HADOOP_HOME/lib/hadoop-streaming.jar
    -input input_directories
    -output output_directories
    -mapper mapper_script
    -reducer reducer_script
```

Partitioning and Sorting

What is a Partitioner?



The Partitioner determines which records get sent to which Reducer


1 The Mapper outputs <key,value> pairs
Mapper
<key1, value> <key6, value> <key2, value>
<key2, value> <key1, value> <key8, value>
<key3, value> <key8, value> <key1, value>
Partitioner
NodeManager
2 Each pair is passed to the Partitioner
3 The getPartition method returns an int between 0 and the number of Reducers minus 1
Reducer 0
NodeManager
Reducer 1
NodeManager
Reducer 2
NodeManager
Reducer 3
NodeManager

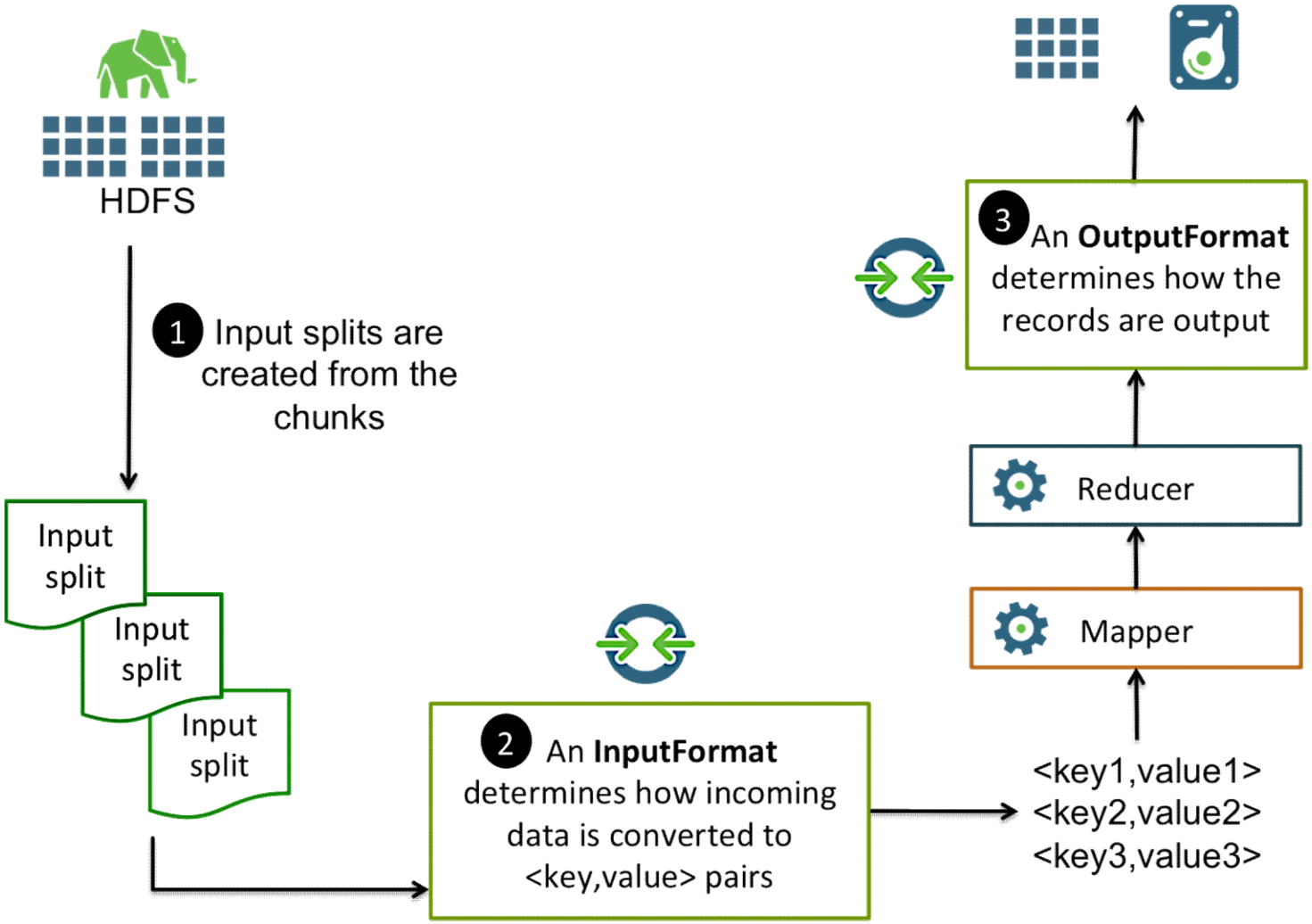
The Default Partitioner

```
public class HashPartitioner<K, V>
                extends Partitioner<K, V>
{

public int getPartition(K key, V value,
                        int numReduceTasks) {
          return (key.hashCode()
                  & Integer.MAX_VALUE)
                  % numReduceTasks;
   }
}
```

Writing a Custom Partitioner

```
public class WordCountPartitioner
    extends Partitioner<Text, IntWritable> {
   public int getPartition(Text key,
             IntWritable value,
             int numReduceTasks) {
      if(numReduceTasks == 1) {
         return 0;
      }
      return (key.toString().length()
          * value.get()) % numReduceTasks;
   }
}
```


HDFS
1 Input splits are created from the chunks
Input split
Input split
Input split
2 An InputFormat determines how incoming data is converted to <key,value> pairs
<key1,value1>
<key2,value2>
<key3,value3>
Mapper
Reducer
3 An OutputFormat determines how the records are output

The Built-in Input Formats

`FileInputFormat<K,V>`

- useful parent class

`TextInputFormat<LongWritable, Text>`

- default Input Format

`SequenceFileInputFormat<K,V>`

`KeyValueTextInputFormat<Text,Text>`

- key is the first token of a line of text

`CombineFileInputFormat<K,V>`

`MultipleInputs`

The Built-in Output Formats

`FileOutputFormat<K,V>`

- useful parent class

`TextOutputFormat<K,V>`

- the default

`SequenceFileOutputFormat<K,V>`
`MultipleOutputs<K,V>`

- for sending output to multiple destinations

`NullOutputFormat<K,V>`

- no output is generated

`LazyOutputFormat<K,V>`