UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Apache Hadoop

Core Apache Hadoop

- Core Hadoop is a software **platform** and **framework** for distributed computing of data.
  - It is a **platform** in the sense that it is a long-running system that runs and executes computing tasks.
  - It is a **framework** in the sense that it provides a layer of abstraction to developers of data applications and data analytics, hiding the intricacies of the system.

Major components

- **HDFS** (Hadoop Distributed File System)
  A filesystem that stores data across multiple computers (i.e., in a distributed manner); it is designed to be high throughput, resilient, and scalable

- **YARN** (Yet Another Resource Negotiator)
  A management framework for Hadoop resources; it keeps track of the CPU, RAM, and disk space being used, and tries to make sure processing runs smoothly

- **MapReduce**
  A generalized framework for processing and analyzing data in a distributed fashion.

Scaling Hadoop

- Hadoop **scales out** (it does not **scale up**): which means you can add to your existing system with newer or more powerful pieces.

- For example, scaling up your refrigerator means you buy a larger refrigerator and trash your old one; scaling out means you buy another refrigerator to sit beside your old one.

- If data doubles the double processing power and you'll still process within the same time.

Hadoop cluster

- Running Hadoop means running a set of daemons on the different servers in your network.
  These daemons have specific roles; some exist only on one server, some exist across multiple servers.
  The daemons include:
  - NameNode
  - DataNode
  - Secondary NameNode
  - JobTracker
  - TaskTracker

HDFS

- The Hadoop Distributed File System (HDFS) gives you a way to store a lot of data in a distributed fashion. It works with the other components of Hadoop to serve up data files to systems and frameworks.

- HDFS is implemented as a “master and slave” architecture that is made up of a NameNode (the master) and one or more data nodes (the slaves).

Namenode

- The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. It keeps track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

- The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn’t store any user data or perform any computations for a MapReduce program to lower the workload on the machine. This means that the NameNode server doesn’t double as a DataNode or a TaskTracker.

The NameNode does three major things:

1. it knows where data is,
2. it tells clients where to send data,
3. tells clients where to retrieve data from.

- The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. It keeps track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

- The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn’t store any user data or perform any computations for a MapReduce program to lower the workload on the machine. This means that the NameNode server doesn’t double as a DataNode or a TaskTracker.

Datanode

- Each slave machine in your cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem — reading and writing HDFS blocks to actual files on the local filesystem.

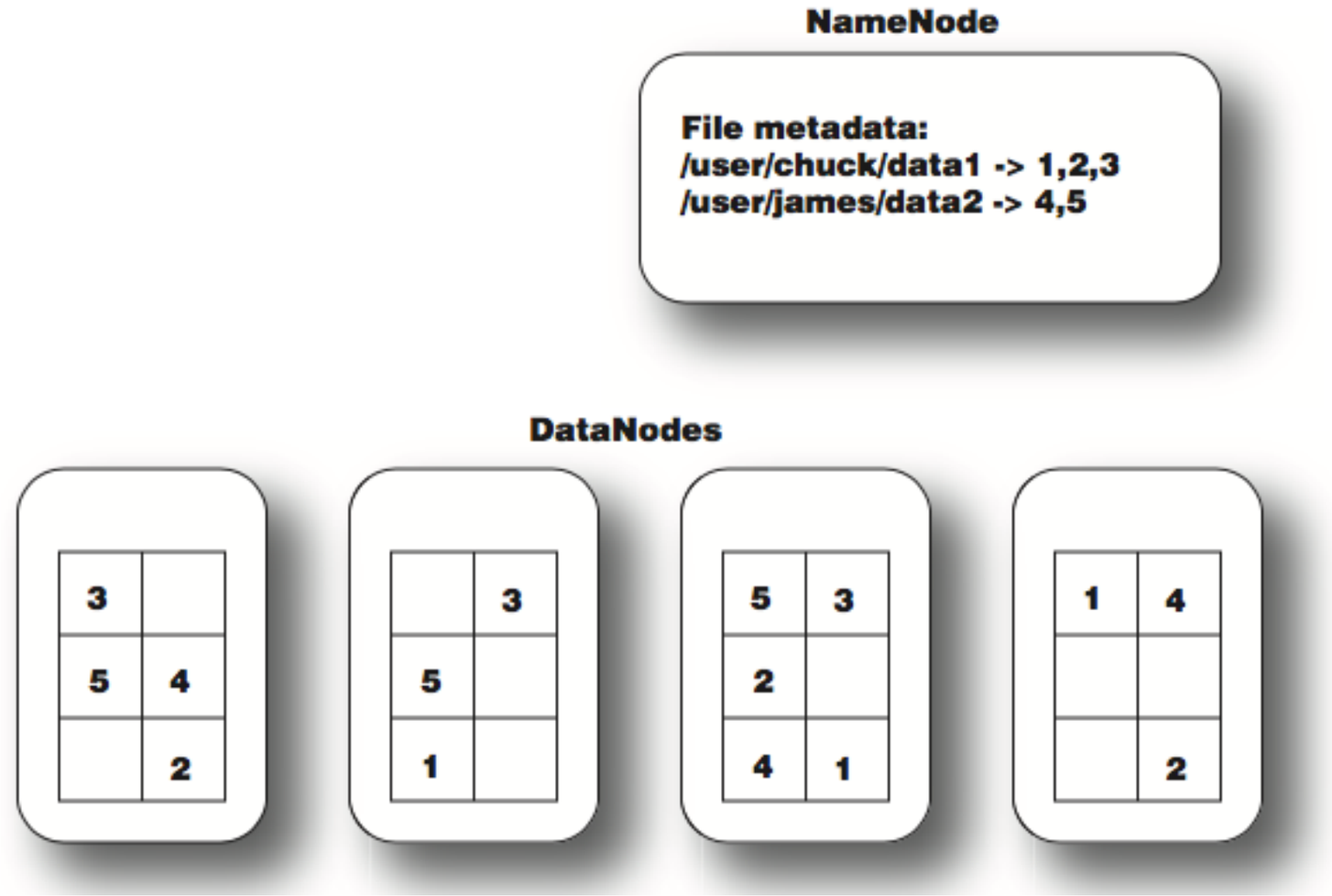
- When you want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell your client which DataNode each block resides in.

- Your client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy.

Namenode + datanode

- NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata — which files are in the system and how each file is broken down into blocks.

- The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.



To/from HDFS

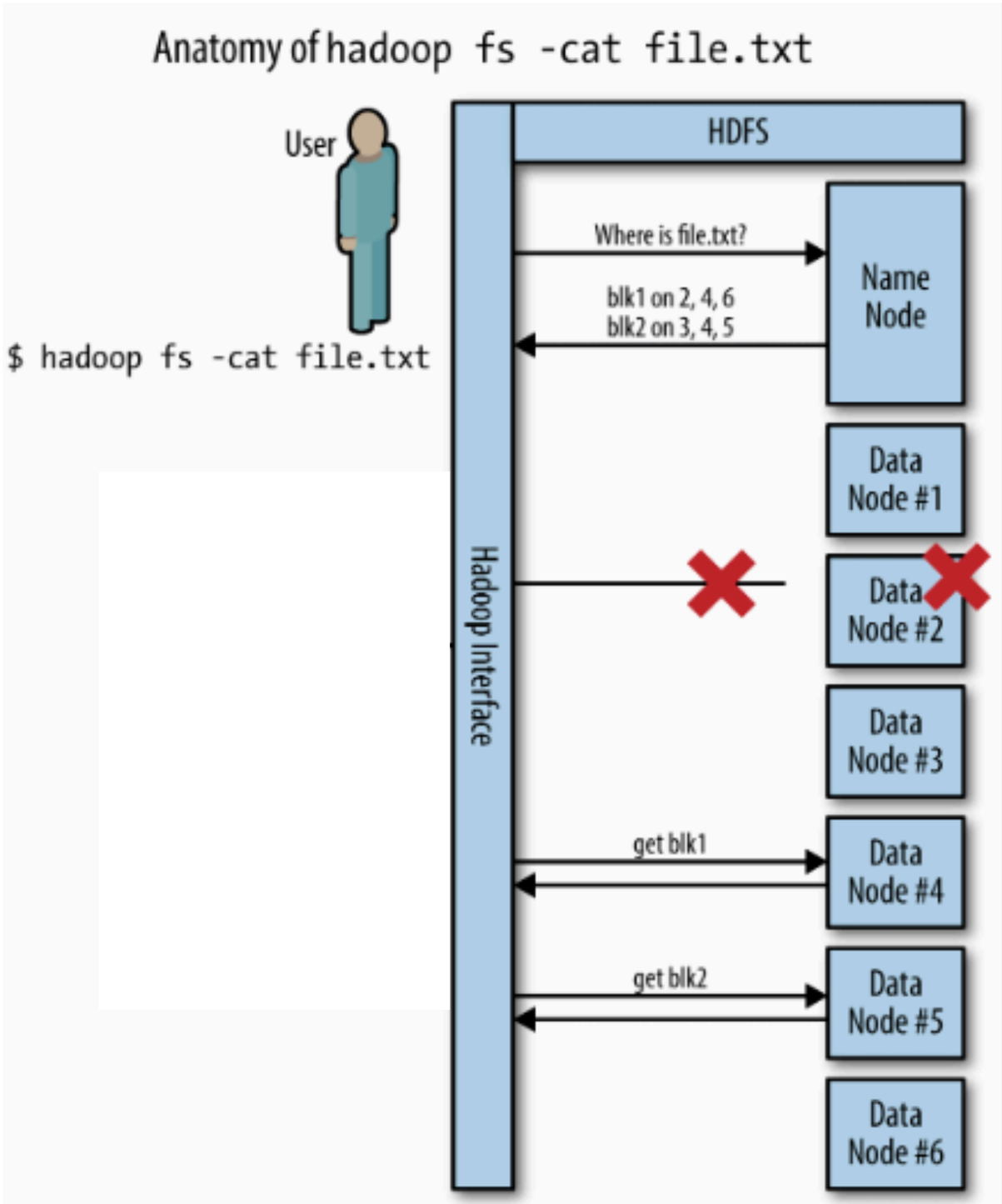
- Transferring data to and from HDFS is a two-step process:

1. The client connects to the NameNode and asks which DataNode it can get the data from or where it should send the data.

2. The client then connects to the DataNode the NameNode indicated and receives or sends the data directly to the DataNode, without the NameNode's involvement.

To/from HDFS

- By default, HDFS stores three copies of your files scattered across the cluster somewhat randomly.
- As a user you'll never notice that there are three copies. Even if some failure occurs and there are temporarily only two copies, you'll never know because it is all handled behind the scenes by the NameNode and the DataNodes



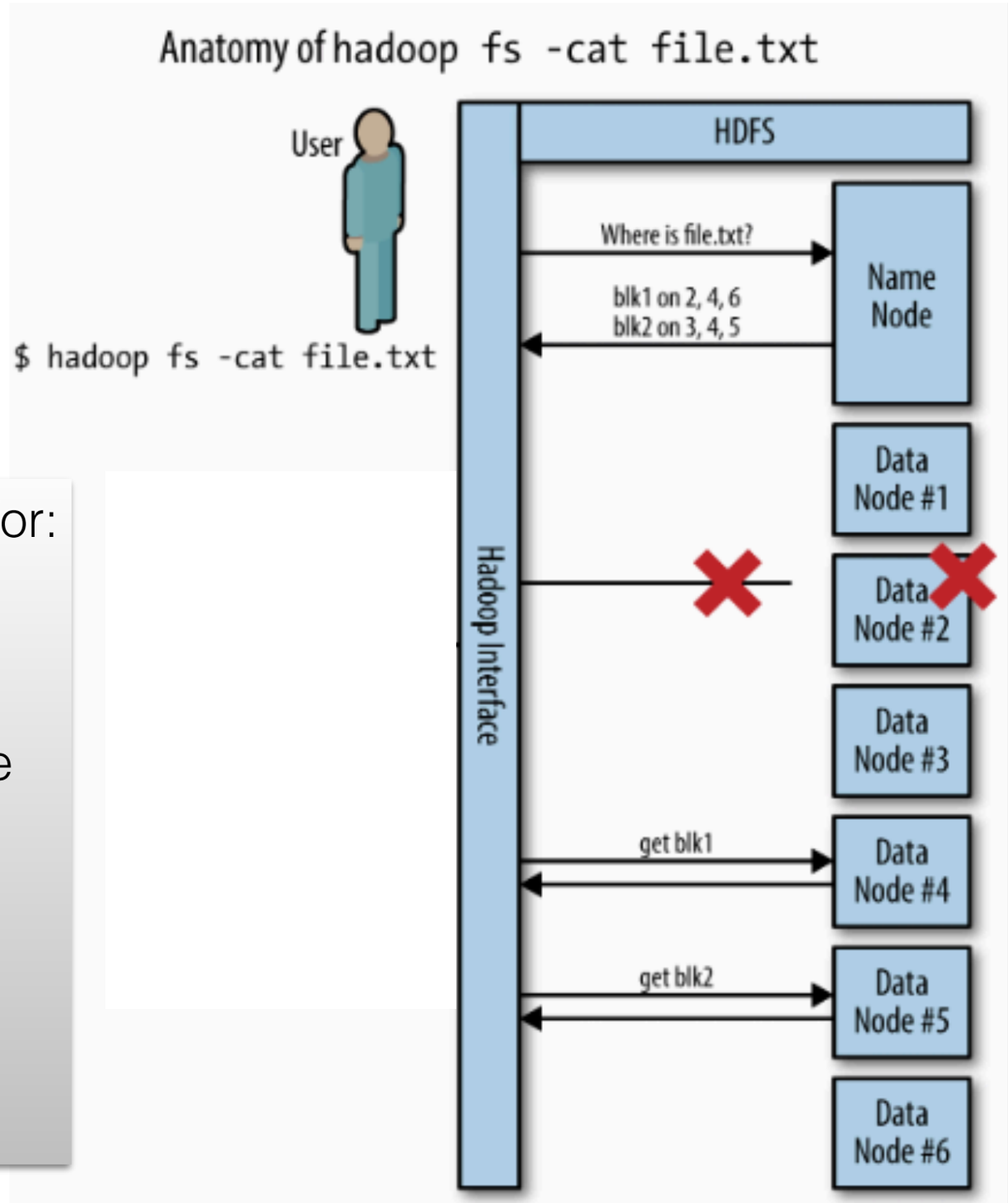
To/from HDFS

- By default, HDFS stores three copies of your files scattered across the cluster somewhat randomly.

Multiple copies of chunks are beneficial for:

1. robustness to node failures (fault tolerance). In case of failure NameNode commands DataNodes to replicate lost copies

2. performance: allows to data locality: perhaps one of the chunks is near the node that needs it



HDFS: non modifiable files

- To allow good throughput HDFS does not offer full functionalities of a traditional file system: files written to HDFS can not be modified

- Either write a file to HDFS, read from it or delete. No edit.

Secondary NameNode

- Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well.
  No other DataNode or TaskTracker daemons run on the same server.

- The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.

- It is used to recover a system in case of name node failure.

YARN

- The role of YARN is to divvy up cluster resources (such as compute and memory) over a pool of computers.

- YARN spreads out tasks and workloads over the cluster, telling each individual computer what it should be running and how many resources should be given to it.

- YARN works in the background and doesn't require much interaction with Hadoop developers, that work with the MapReduce API.

MapReduce

- MapReduce is a generalized framework for analyzing data stored in HDFS over the computers in a cluster. It allows the analytic developer to write code that is completely ignorant of the nature of the massive distributed system underneath it.

- MapReduce itself is a paradigm for distributed computing described in a 2004 paper by engineers at Google. The authors of the paper described a general-purpose method to analyze large amounts of data in a distributed fashion on commodity hardware, in a way that masks a lot of the complexities of distributed systems.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

MapReduce

What is MapReduce

- MapReduce is a broad term. Sometimes it's used to describe the common pattern of breaking an algorithm down into two steps: a **map** over a data structure, followed by a **reduce** operation.

- In many programming languages, **map** is the name of a higher-order function that applies a given function to each element of a list, returning a list of results. It is often called apply-to-all when considered in functional form.

- In functional programming, **reduce** refers to a family of higher-order functions that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.

What is MapReduce

- MapReduce can also be used to mean something more specific — a system that takes an algorithm encoded as a **map** followed by a **reduce** and efficiently distributes it across a cluster of computers.

- The system automatically partitions both the data and its processing between the machines within the cluster, but it also continues to operate if one or more of those machines fails.

- The most popular MapReduce framework is Apache Hadoop.

What is MapReduce

- MapReduce can also be used to mean something more specific — a system that takes an algorithm encoded as a **map** followed by a **reduce** and efficiently distributes it across a cluster of computers.

A **Map** procedure (method) performs filtering and sorting (such as sorting students by first name into queues, one queue for each name)

and its processing between the machines within the cluster, but it also continues to operate if one or more of those machines fails.

- The most popular MapReduce framework is Apache Hadoop.

What is MapReduce

- MapReduce can also be used to mean something more specific — a system that takes an algorithm encoded as a **map** followed by a **reduce** and efficiently distributes it across a cluster of computers.

A **Map** procedure (method) performs filtering and sorting (such as sorting students by first name into queues, one queue for each name)

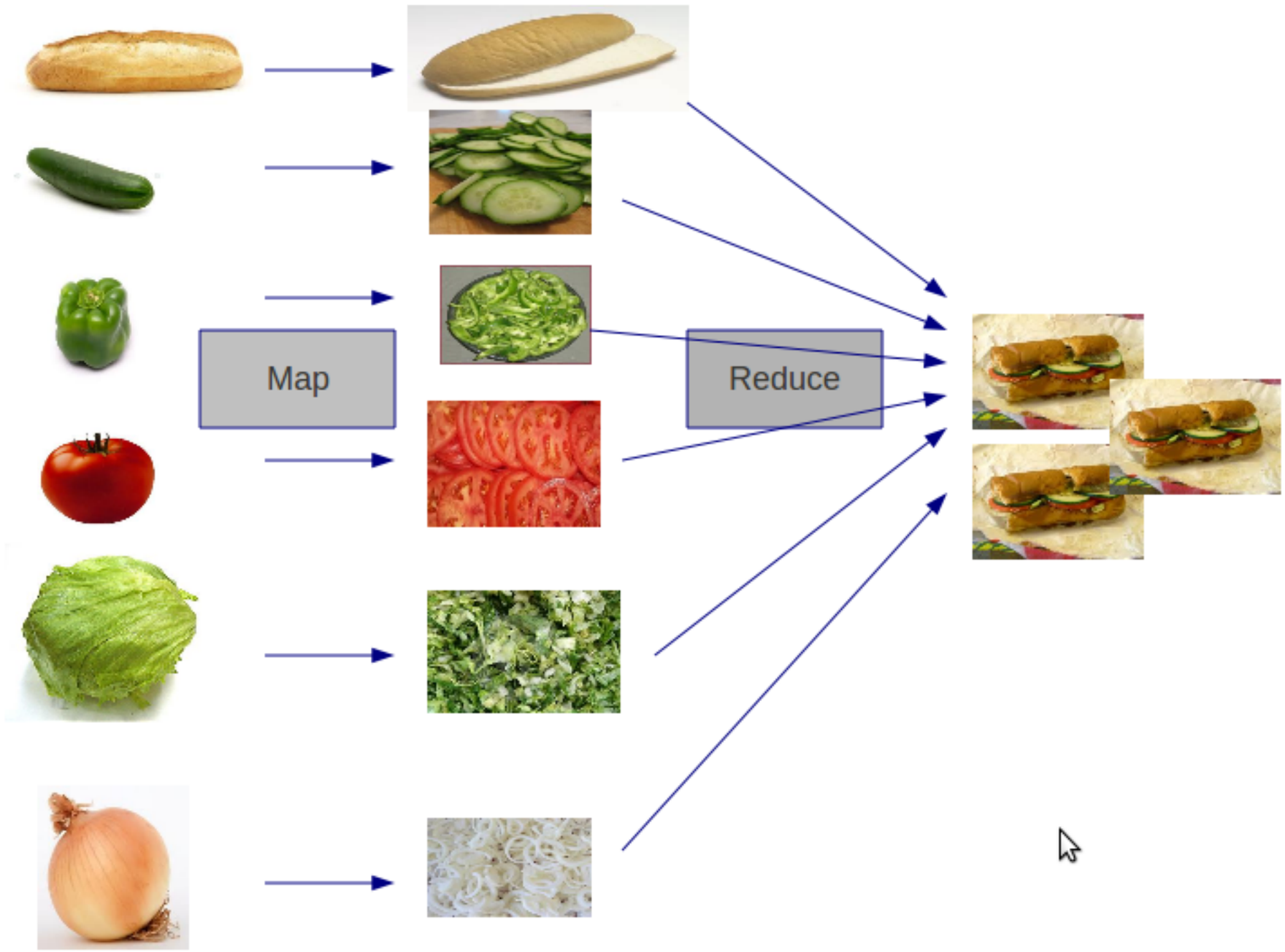
and its processing between the machines within the

A **Reduce** method performs a summary operation (such as counting the number of students in each queue, yielding name frequencies)

more of those machines fails.

- The most popular MapReduce framework is Apache Hadoop.

What is MapReduce



Hadoop basics

- Hadoop is all about processing large amounts of data.
  Unless your data is measured in gigabytes or more, it's unlikely to be the right tool for the job.
  Its power comes from the fact that that it splits data into sections, each of which is then processed independently by separate machines.

- A MapReduce task is constructed from two primary types of components, **mappers** and **reducers**.

  - **Mappers** take some input format (by default, lines of plain text) and map it to a number of key/value pairs.

  - **Reducers** then convert these key/value pairs to the ultimate output format (normally also a set of key/value pairs).

  - Mappers and reducers are distributed across many different physical machines. there's no requirement for there to be the same number of mappers as reducers.

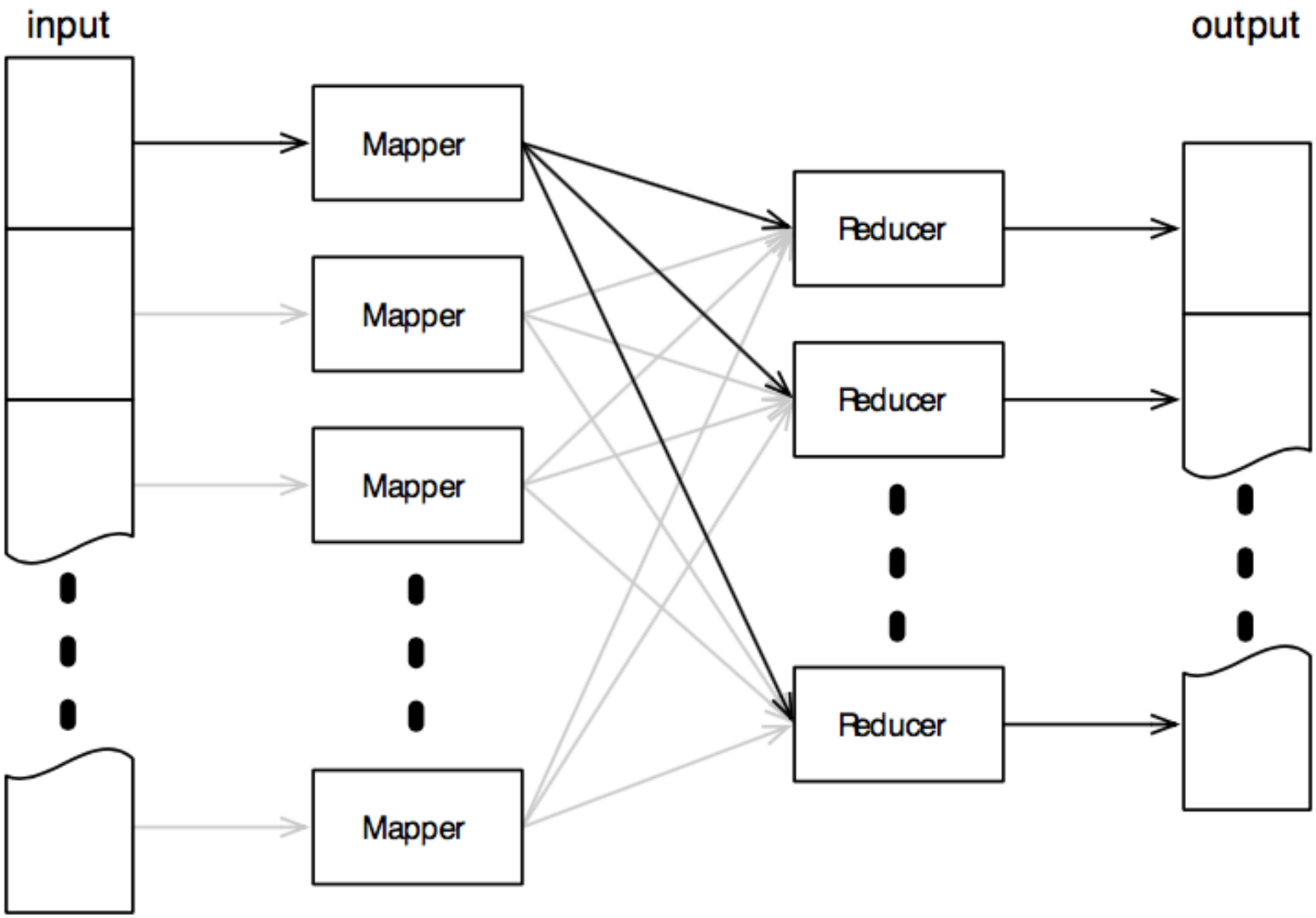
Hadoop basics

- The input typically comprises one or more large text files. Hadoop splits these files (typical split is 64MB) and sends each split to a single mapper. The mapper outputs a number of key/value pairs, which Hadoop then sends to the reducers.

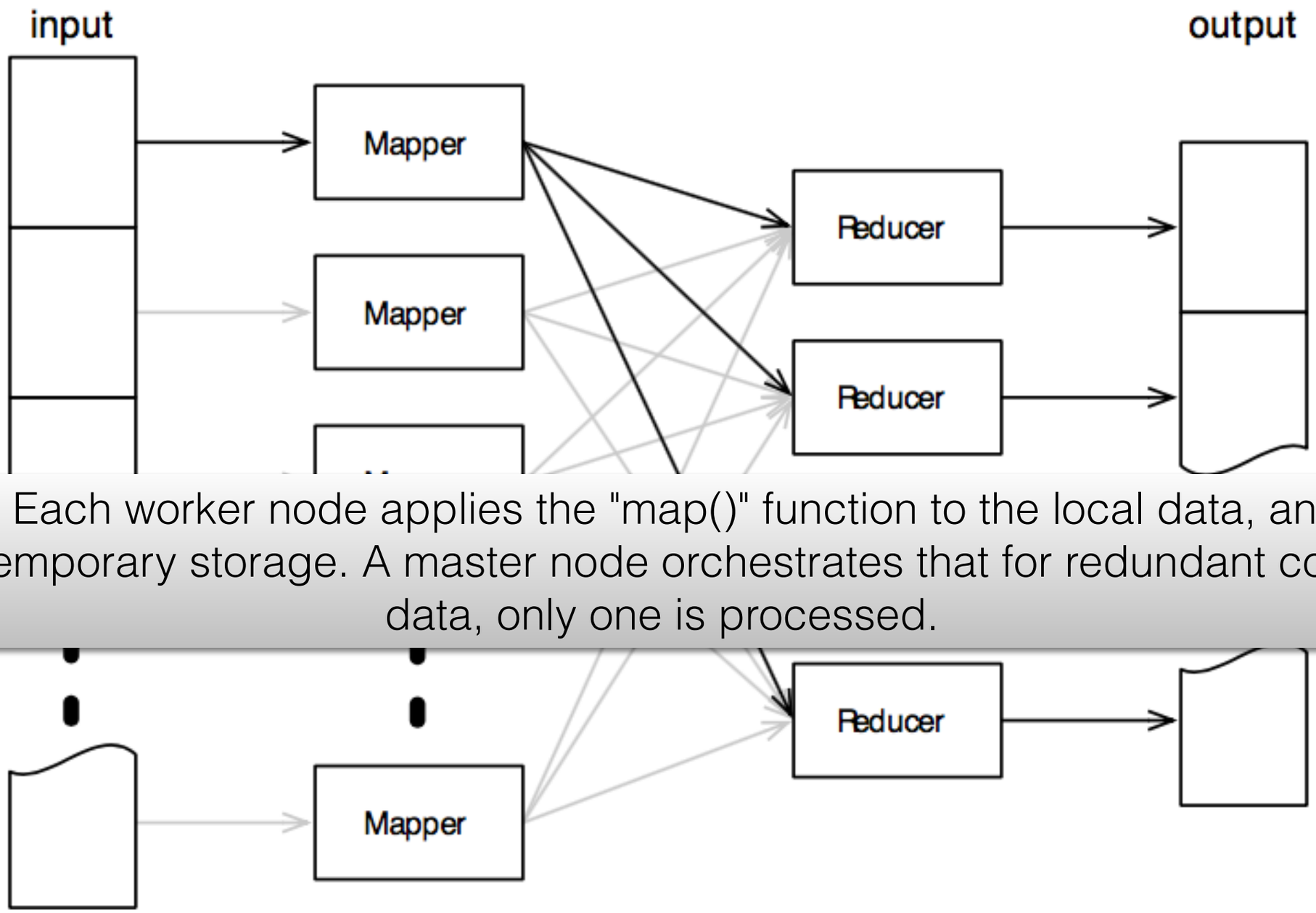
- The key/value pairs from a single mapper are sent to multiple reducers. Which reducer receives a particular key/value pair is determined by the key — Hadoop guarantees that all pairs with the same key will be processed by the same reducer, no matter which mapper generated them. This is commonly called the **shuffle** phase.

- Hadoop calls the reducer once for each key, with a list of all the values associated with it. The reducer combines these values and generates the final output (which is typically, but not necessarily, also key/value pairs).

Hadoop high-level data flow

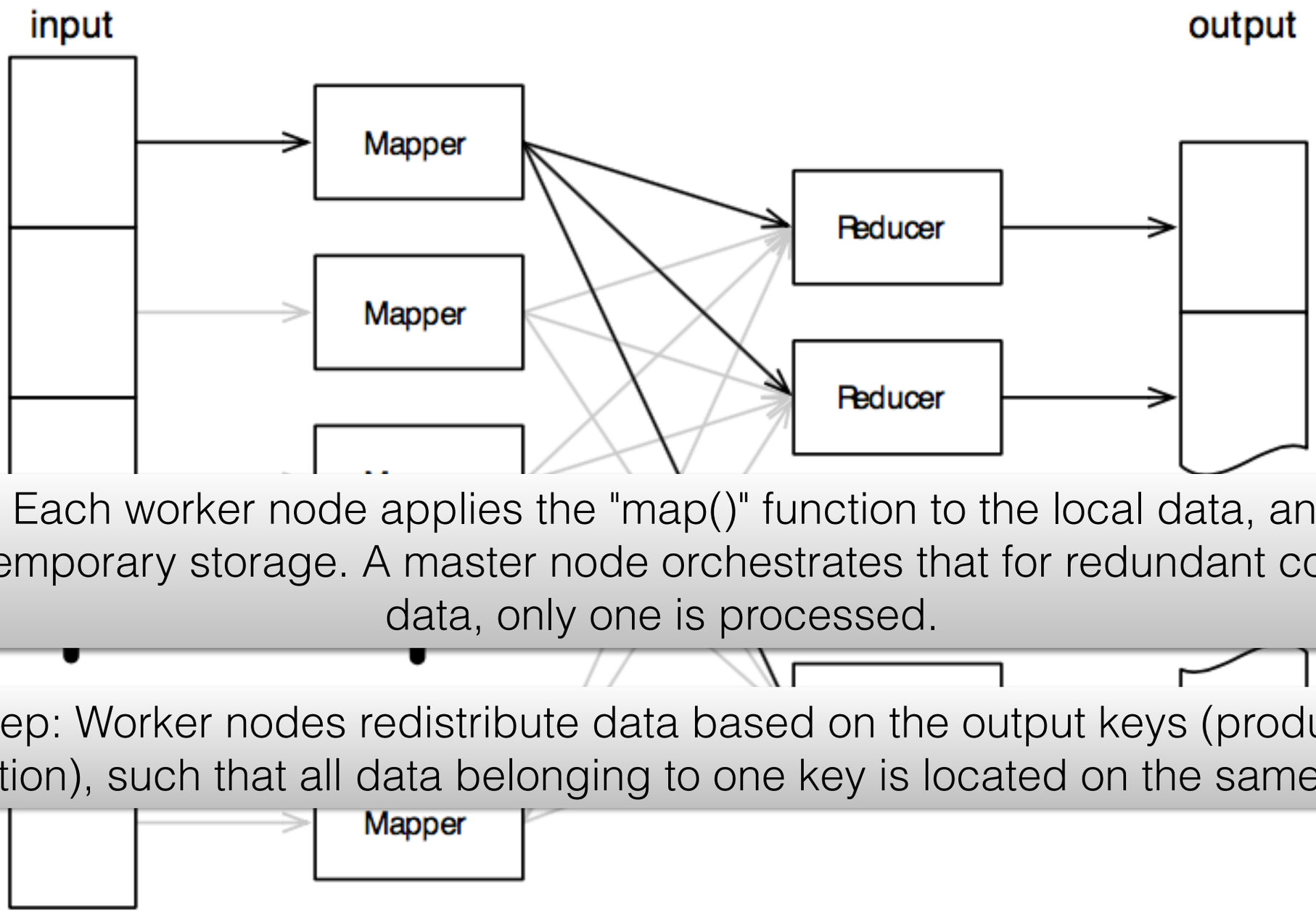


Hadoop high-level data flow



"Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

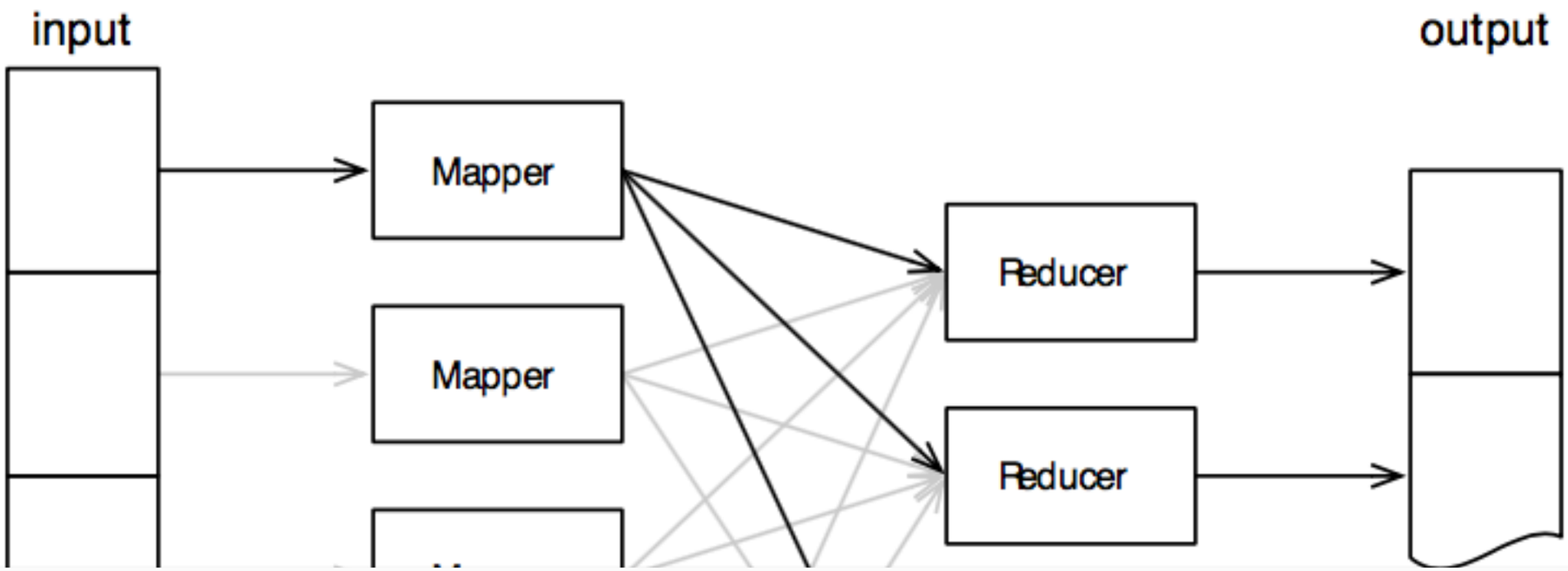
Hadoop high-level data flow



"Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

Shuffle" step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

Hadoop high-level data flow



"Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

Shuffle" step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

"Reduce" step: Worker nodes now process each group of output data, per key, in parallel.

Data flow

	Input	Output
Map	<k1, v1>	list(<k2, v2>)
Reduce	<k2, list(v2)>	list(<k3, v3>)

- MapReduce uses lists and (key/value) pairs as its main data primitives. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types.

- The map and reduce functions must some constraints on the types of keys and values.

Data flow: map

- The input to the application must be structured as a list of (key/value) pairs, list(<k1, v1>).
  - E.g. the input format for processing multiple files is usually `list(<String filename, String file_content>)`.
    The input format for processing one large file, such as a log file, is `list(<Integer line_number, String log_event>)`.
- The list of (key/value) pairs is broken up and each individual (key/value) pair, <k1, v1>, is processed by calling the **map** function of the mapper.
  In practice, the key k1 is often ignored by the mapper.
  The mapper transforms each <k1, v1> pair into a list of <k2, v2> pairs.
  The details of this transformation largely determine what the MapReduce program does.

  Note that the (key/value) pairs are processed in arbitrary order. The transformation must be self-contained in that its output is dependent only on one single (key/value) pair.

Data flow: reduce

- The output of all the mappers are (conceptually) aggregated into one giant list of <k2, v2> pairs.

- All pairs sharing the same k2 are grouped together into a new (key/value) pair, <k2, list(v2)>.
  The framework asks the reducer to process each one of these aggregated (key/value) pairs individually.

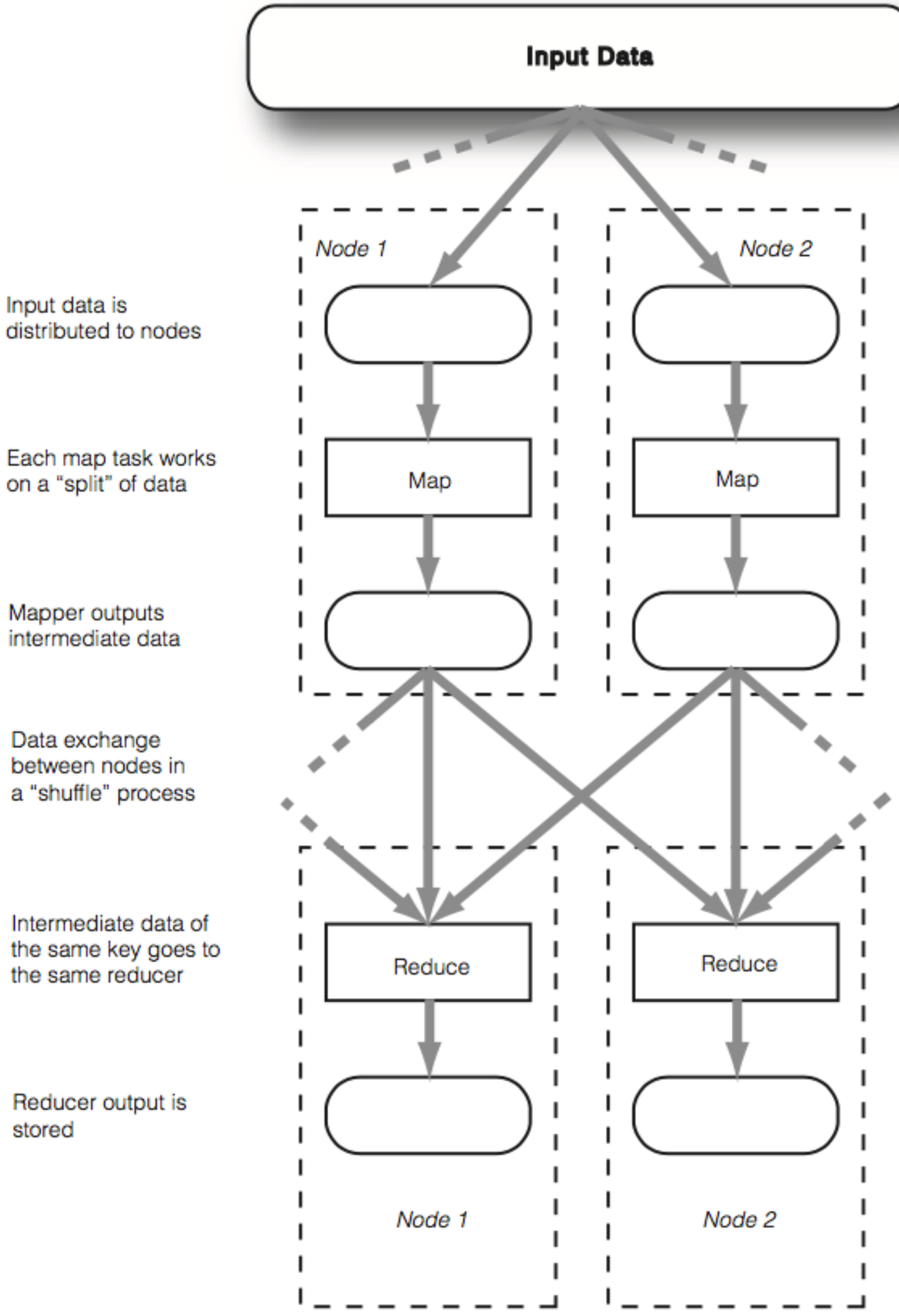
- The MapReduce framework automatically collects all the <k3, v3> pairs and writes them to file(s).

  - The data types k2 and k3, v2 and v3, may or may not be the same.

UNIVERSITÀ DEGLI STUDI FIRENZE

Data flow

- The general MapReduce data flow.
- Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the “shuffle” step. This restriction on communication greatly helps scalability.



Testing and Scaling

- Decomposing a data processing application into mappers and reducers is sometimes nontrivial.

- But, once you write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change.

- Test MapReduce applications on a single machine, then deploy on an Hadoop cluster/ grid…

JobTracker

- The JobTracker daemon is the liaison between your application and Hadoop.
  Once you submit your code to your cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running.
  Should a task fail, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries.

- There is only one JobTracker daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster.

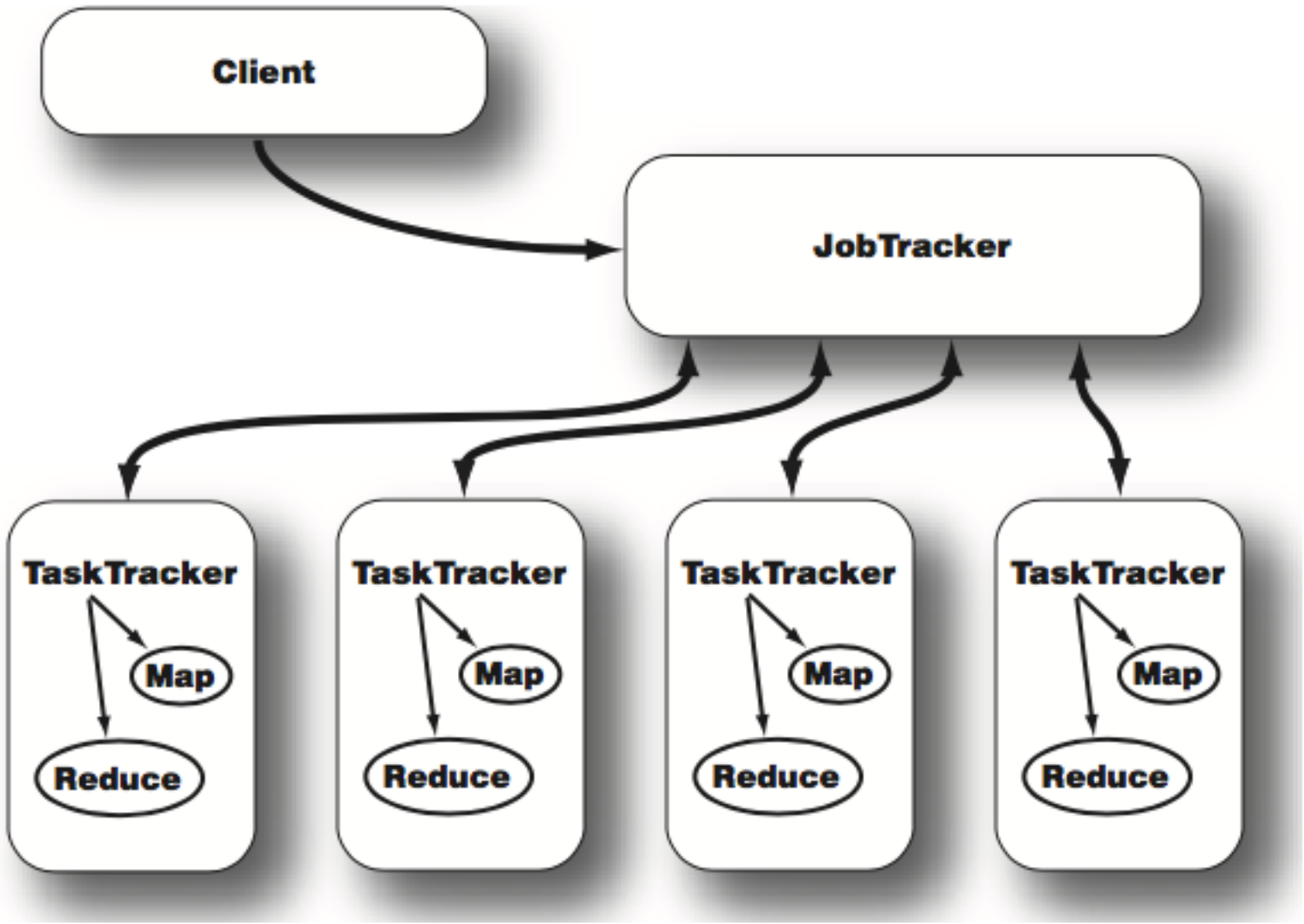
TaskTracker

- As with the storage daemons, the computing daemons also follow a master/slave architecture: the JobTracker is the master overseeing the overall execution of a MapReduce job and the TaskTrackers manage the execution of individual tasks on each slave node.

- Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple JVMs to handle many map or reduce tasks in parallel.

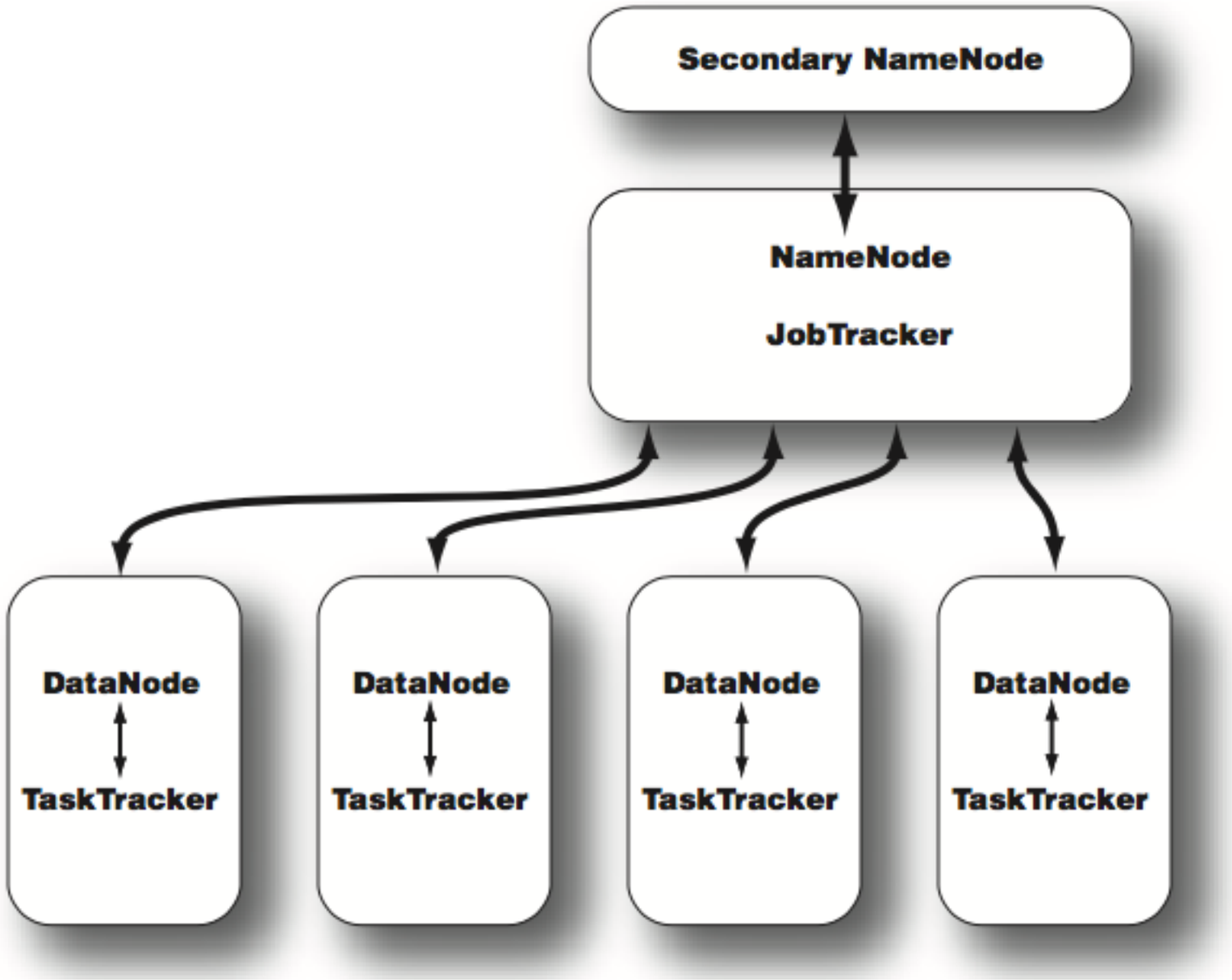
- One responsibility of the TaskTracker is to constantly communicate with the JobTracker. If the JobTracker fails to receive a heartbeat from a TaskTracker within a specified amount of time, it will assume the TaskTracker has crashed and will resubmit the corresponding tasks to other nodes in the cluster.

JobTracker + TaskTracker



- JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster.

Overall architecture



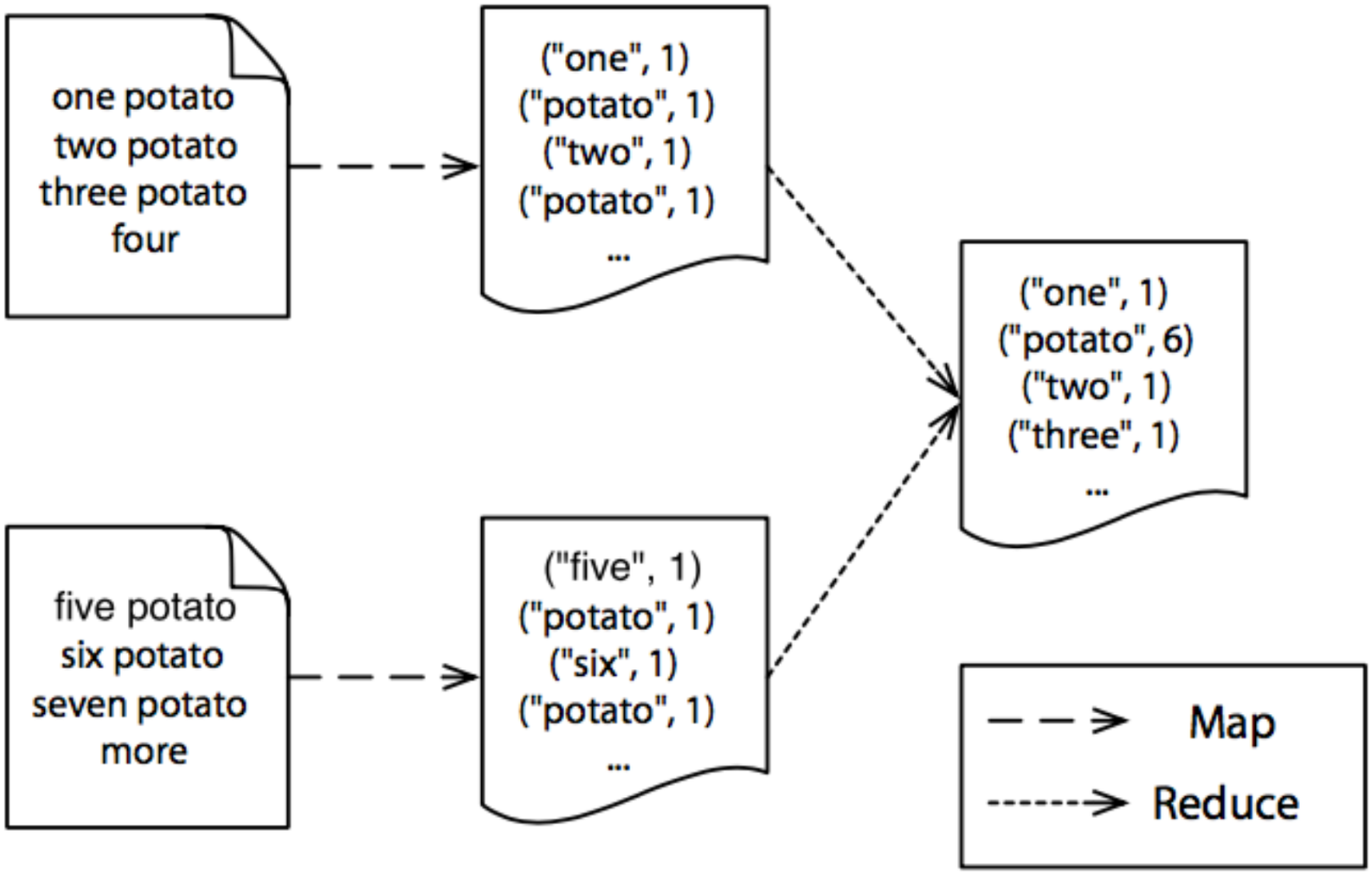
- Topology of a typical Hadoop cluster. It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

A simple Hadoop program

Counting Words with Hadoop

- Task: counting the number of words in a collection of plain-text files, e.g. tags of images in a social network, or words from Wikipedia.



Counting Words with Hadoop

- Our **mapper** will process text a line at a time, break each line into words and output a single key/value pair for each word.
  The key will be the word itself, and the value will be the constant integer 1.

- Our **reducer** will take all the key/value pairs for a given word and sum the values, generating a single key/value pair for each word, where the value is a count of the number of times that word occurred in the input.

Hadoop data types

- The MapReduce framework won't allow keys and values to be any arbitrary class. I.e. can not use standard Java classes, such as Integer, String, and so forth.
  This is because the MapReduce framework has a certain defined way of serializing the key/value pairs to move them across the cluster's network, and only classes that support this kind of serialization can function as keys or values in the framework.

- Classes that implement the `Writable` interface can be values, and classes that implement the `WritableComparable<T>` interface can be either keys or values.
  Note that the `WritableComparable<T>` interface is a combination of the `Writable` and `java.lang.Comparable<T>` interfaces.
  We need the comparability requirement for keys because they will be sorted at the reduce stage, whereas values are simply passed through.

Example of WritableComparable class

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class Edge implements WritableComparable<Edge> {
    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode() { return departureNode;}

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
     return (departureNode.compareTo(o.departureNode) != 0)
         ? departureNode.compareTo(o.departureNode)
         : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

Example of WritableComparable class

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class Edge implements Writable
    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode()

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
     return (departureNode.compareTo(o.departureNode) != 0)
        ? departureNode.compareTo(o.departureNode)
         : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

Methods of Writable interface. They work with the Java DataInput and DataOutput classes to serialize the class contents.

Example of WritableComparable class

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class Edge implements Writable
    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode()

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOExc
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
     return (departureNode.compareTo(o.departureNode) != 0)
        ? departureNode.compareTo(o.departureNode)
         : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

Methods of Writable interface. They work with the Java DataInput and DataOutput classes to serialize the class contents.

Method of the Comparable interface. It returns -1, 0, or 1 if the called Edge is less than, equal to, or greater than the given Edge.

The Mapper

```
public static class Map extends
                    Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);

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

        String line = value.toString();
        Iterable<String> words = new Words(line);
        for (String word: words)
            context.write(new Text(word), one);
    }

}
```

The mapper handles plain text data, not key/value pairs, so the input key type is unused (we pass Object) and the input value type is Text.
The output key type is also Text, with a value type of IntWritable.
Text is a Hadoop wrapper to store text using the UTF8 format

```
public static class Map extends
                    Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);

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

        String line = value.toString();
        Iterable<String> words = new Words(line);
        for (String word: words)
            context.write(new Text(word), one);
    }

}
```

The Reducer

```
public static class Reduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key,
                     Iterable<IntWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val: values)
      sum += val.get();
    context.write(key, new IntWritable(sum));
  }

}
```

It also takes type parameters indicating the input and output key and value types.
In our case, Text for both key types and IntWritable for both value types.

```
public static class Reduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key,
                     Iterable<IntWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val: values)
      sum += val.get();
    context.write(key, new IntWritable(sum));
  }

}
```

It also takes type parameters indicating the input and output key and value types. In our case, Text for both key types and IntWritable for both value types.

```
public static class Reduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key,
                     Iterable<IntWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val: values)
      sum += val.get();
    context.write(key, new IntWritable(sum));
  }
```

The reduce() method will be called once for each key, with values containing a collection of all the values associated with that key.

API

```
public static class MapClass extends Mapper<K1, V1, K2, V2> {

    public void map(K1 key, V1 value, Context context)
                    throws IOException, InterruptedException {
    }
}


public static class Reduce extends Reducer<K2, V2, K3, V3> {

    public void reduce(K2 key, Iterable<V2> values, Context
context)
                    throws IOException, InterruptedException { }

}
```

The Driver

- We need a driver, that tells Hadoop how to run mapper and reducer.
- We are going to set the names of mapper and reducer classes, set the input and value types, tell Hadoop where to find input data and where to write output data.
- Then we'll start the job and wait for its completion.

Job and Configuration

- The Configuration class configures a job
- The Job class defines and controls the execution of a job.
  - A job's construction and submission for execution are under Job.

The Driver

```
public class WordCount extends Configured implements Tool {

  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = Job.getInstance(conf, "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(), args);
    System.exit(res);
  }
}
```

The Driver

```
public class WordCount extends Configured implements Tool {

  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = Job.getInstance(conf, "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
  }
```

- In this simple case there's no need to set the input key and value type, because Hadoop assumes by default that we're processing text files.
- We don't need to independently set the mapper output or reducer input key/value types, because Hadoop assumes by default that they're the same as the output key/value types.
- We set the output key and value types

Processing XML

- Hadoop's default splitter divides files at line boundaries, meaning that it's likely to split files in the middle of XML tags.
  - There are other default splitters to handle compressed files, K/V pairs, specified n lines splitters…
- There's need to use a splitter that is aware of the structure of the data being processed, like a Mahout XML splitter
  - Implement interface `InputFormat<K, V>` to create other managers of input format

Processing XML

```
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  conf.set("xmlinput.start", "<text");
  conf.set("xmlinput.end", "</text>");

  Job job = Job.getInstance(conf, "wordcount");
  job.setJarByClass(WordCount.class);
  job.setInputFormatClass(XmlInputFormat.class);
  job.setMapperClass(Map.class);
  job.setCombinerClass(Reduce.class);
  job.setReducerClass(Reduce.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0])); 15
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  boolean success = job.waitForCompletion(true);
  return success ? 0 : 1;
}
```

Processing XML

```
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  conf.set("xmlinput.start", "<text");
  conf.set("xmlinput.end", "</text>");

  Job job = Job.getInstance(conf, "wordcount");
  job.setJarByClass(WordCount.class);
  job.setInputFormatClass(XmlInputFormat.class);
  job.setMapperClass(Map.class);
  job.setCombinerClass(Reduce.class);
  job.setReducerClass(Reduce.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0])); 15
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  boolean success = job.waitForCompletion(true);
  return success ? 0 : 1;
}
```

tell Hadoop to use XmlInputFormat instead of the default splitter

Processing XML

```
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  conf.set("xmlinput.start", "<text");
  conf.set("xmlinput.end", "</text>");

  Job job = Job.getInstance(conf, "wordcount");
  job.setJarByClass(WordCount.class);
  job.setInputFormatClass(XmlInputFormat.class);
  job.setMapperClass(Map.class);
  job.setCombinerClass(Reduce.class);
  job.setReducerClass(Reduce.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0])); 15
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  boolean success = job.waitForCompletion(true);
  return success ? 0 : 1;
}
```

let the splitter know which tags we're interested in

tell Hadoop to use XmlInputFormat instead of the default splitter

`XmlInputFormat` doesn't perform a full XML parse; instead it simply looks for start and end patterns. If the `<text>` tag takes attributes, we just search for `<text`

```
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  conf.set("xmlinput.start", "<text");
  conf.set("xmlinput.end", "</text>");

  Job job = Job.getInstance(conf, "wordcount");
  job.setJarByClass(WordCount.class);
  job.setInputFormatClass(XmlInputFormat.class);
  job.setMapperClass(Map.class);
  job.setCombinerClass(Reduce.class);
  job.setReducerClass(Reduce.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0])); 15
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  boolean success = job.waitForCompletion(true);
  return success ? 0 : 1;
}
```

let the splitter know which tags we're interested in

tell Hadoop to use XmlInputFormat instead of the default splitter

Processing XML

```
private final static Pattern textPattern =
Pattern.compile("^<text.*>(.*)</text>$", Pattern.DOTALL);


public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
  String text = value.toString();
  Matcher matcher = textPattern.matcher(text);
  if (matcher.find()) {
    Iterable<String> words = new Words(matcher.group(1));
    for (String word: words)
      context.write(new Text(word), one);
  }
}
```

Each split will consist of text between the `xmlinput.start` and `xmlinput.end` patterns, including the matching patterns. So we use a something like a regular-expression to strip the `<text></text>` tags before counting words (to avoid overcounting the word `text`).

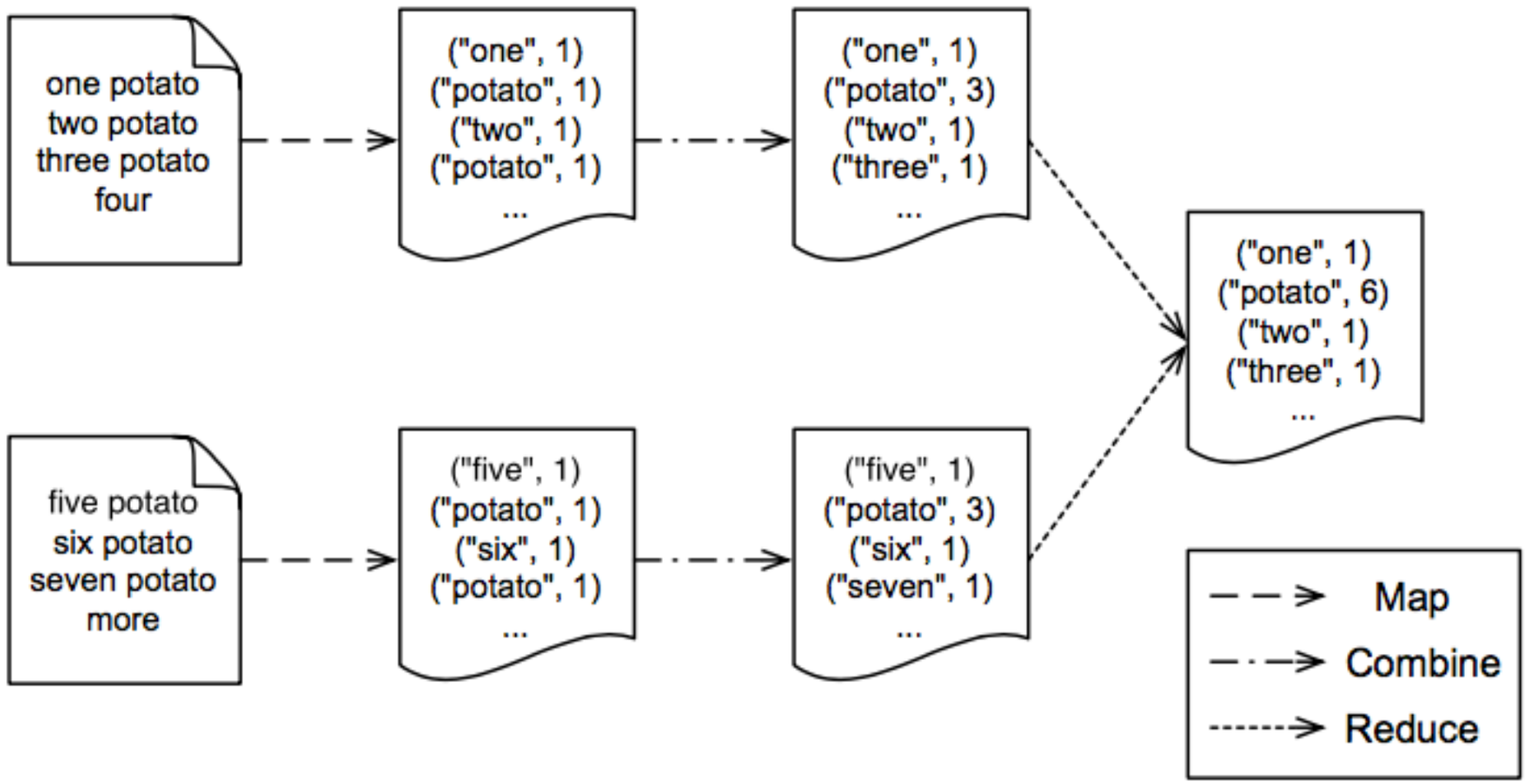
```
private final static Pattern textPattern =
Pattern.compile("^<text.*>(.*)</text>$", Pattern.DOTALL);


public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
  String text = value.toString();
  Matcher matcher = textPattern.matcher(text);
  if (matcher.find()) {
    Iterable<String> words = new Words(matcher.group(1));
    for (String word: words)
      context.write(new Text(word), one);
  }
}
```

The Combiner

- A combiner is an optimization that allows key/value pairs to be combined before they're sent to a reducer.
  - It's a "local reduce" before we distribute the mapper results.
- A reducer, like in this case, may be used also as a combiner.
- Hadoop does not guarantee use of a combiner if one is provided, so we need to make sure that our algorithm doesn't depend on whether, or how often, it is used.

The Combiner



The Partitioner

- With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper.
- The default behavior is to hash the key to determine the reducer. Hadoop enforces this strategy by use of the HashPartitioner class.
  - Sometimes we may need to write our own partitioner

The Partitioner

Consider the Edge for which we wrote the `WritableComparable<T>` interface. Two edges with the same start (e.g. a departure of a travel) and different end (arrival of a travel) would be treated differently. If we wanted to assign them to the same reducer we need to implement a `Partitioner<T, Writable>` interface.

```
public class EdgePartitioner implements
                                Partitioner<Edge, Writable>  {
    @Override
    public int getPartition(Edge key, Writable value,
                            int numPartitions)  {
      return key.getDepartureNode().hashCode() % numPartitions;
    }

    @Override
    public void configure(JobConf conf) { }
}
```

The Partitioner

Consider the Edge for which we wrote the `WritableComparable<T>` interface. Two edges with the same start (e.g. a departure of a travel) and different end (arrival of a travel) would be treated differently. If we wanted to assign them to the same reducer we need to implement a `Parti`

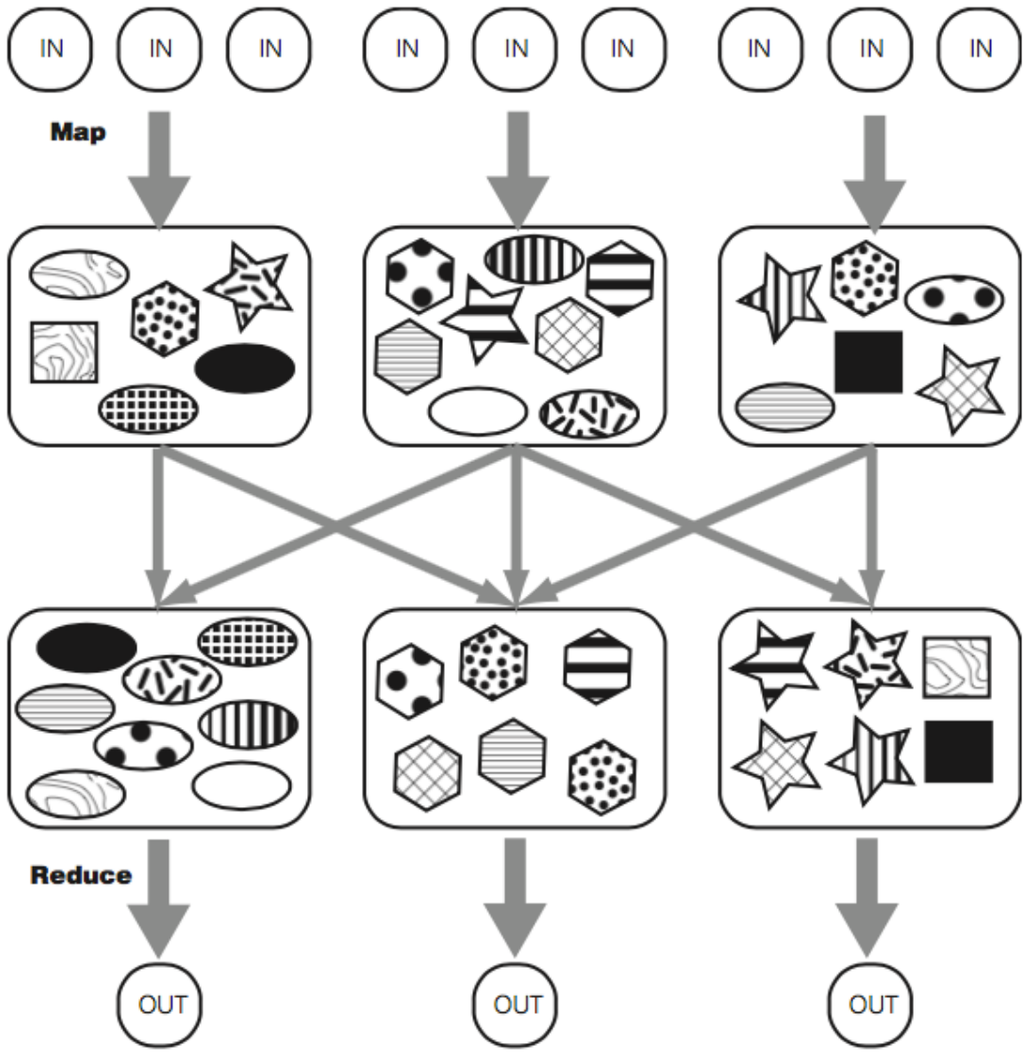
Returns an integer between 0 and the number of reduce tasks indexing to which reducer the (key/value) pair will be sent.

```
public                                    Partitioner<Edge, Writable>  {

    @Override
    public int getPartition(Edge key, Writable value,
                            int numPartitions)  {
      return key.getDepartureNode().hashCode() % numPartitions;
    }

    @Override
    public void configure(JobConf conf) { }
}
```

Partitioning and shuffling



- The MapReduce data flow - partitioning and shuffling. Each icon is a key/value pair. The shapes represents keys, whereas the inner patterns represent values. After shuffling, all icons of the same shape (key) are in the same reducer. Different keys can go to the same reducer, as seen in the rightmost reducer. The partitioner decides which key goes where. Note that the leftmost reducer has more load due to more data under the “ellipse” key.

Output

- The output has no splits, as each reducer writes its output only to its own file. The output files reside in a common directory and are typically named `part-nnnnn`, where nnnnn is the partition ID of the reducer.

- Again: create your own class or use one of the default. A reducer may use `NullOutputFormat<K,V>` to output nothing from Hadoop and instead create its own output

Books

- The Art of Concurrency, Paul Butcher, Pragmatic Bookshelf - Chapt. 8

- Hadoop in Action, Chuck Lam, Manning - Chapt. 1-4