Hadoop
WordCount Explained!

IT332 Distributed Systems

Typical problem solved by MapReduce

Read a lot of data

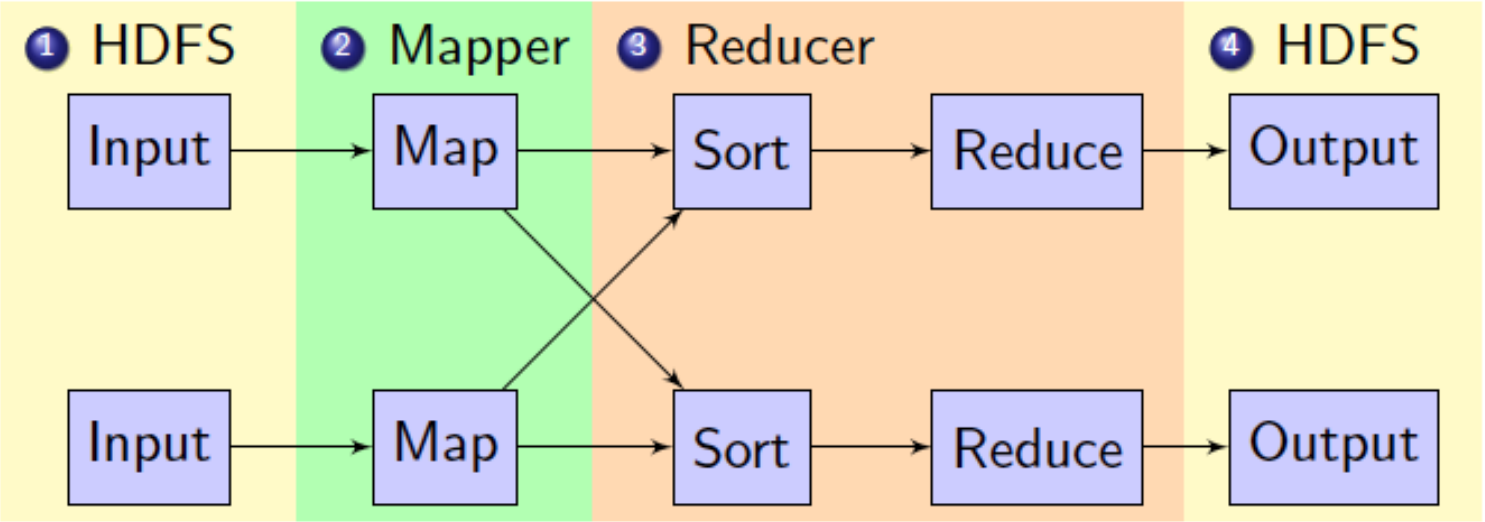
- Map: extract something you care about from each record Shuffle and Sort
- Reduce: aggregate, summarize, filter, or transform Write the results

Outline stays the same,

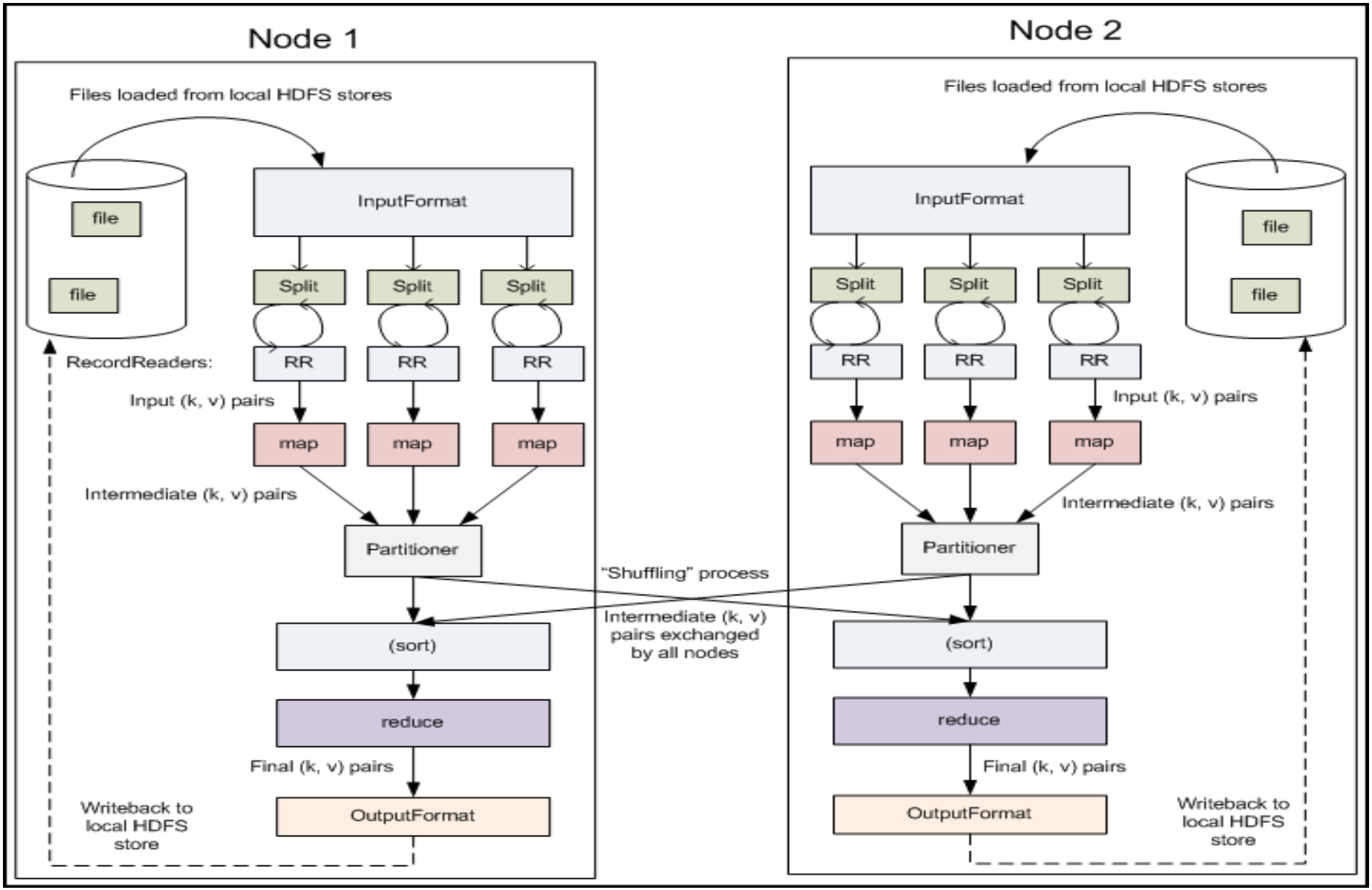
Map and Reduce change to fit the problem

Data Flow

1. Mappers read from HDFS
2. Map output is partitioned by key and sent to Reducers
3. Reducers sort input by key
4. Reduce output is written to HDFS



Detailed Hadoop MapReduce data flow



MapReduce Paradigm

Basic data type: the key-value pair (k,v).

For example, key = URL, value = HTML of the web page.

Programmer specifies two primary methods:

- Map: $(k, v) \mapsto <(k_1,v_1), (k_2,v_2), (k_3,v_3),\ldots,(k_n,v_n)>$
- Reduce: $(k', <v'_1, v'_2,\ldots,v'_{n'}>) \mapsto <(k', v''_1), (k', v''_2),\ldots,(k', v''_{n''})>$

All v' with same k' are reduced together. (Remember the invisible “Shuffle and Sort” step.)

MapReduce Paradigm

- Shuffle

Reducer is input the grouped output of a Mapper. In the phase the framework, for each Reducer, fetches the relevant partition of the output of all the Mappers, via HTTP.

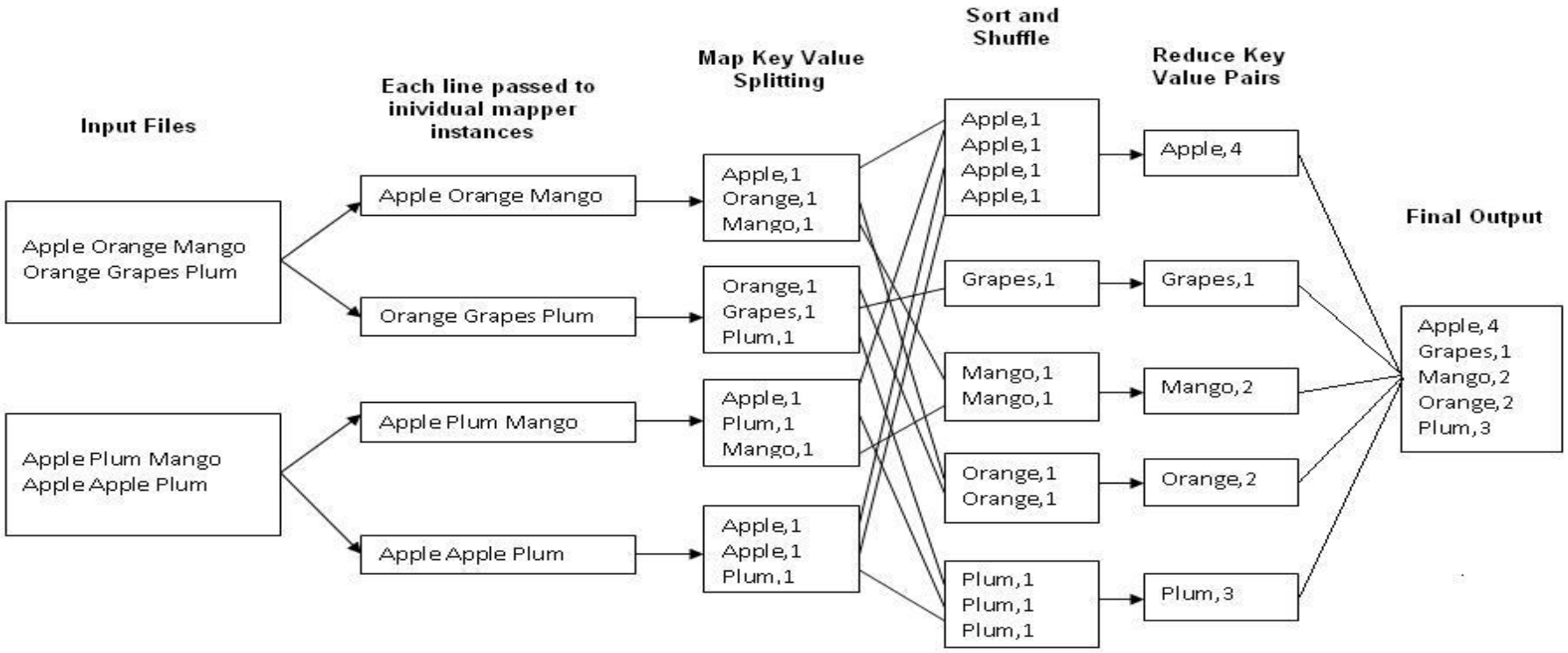
- Sort

The framework groups Reducer inputs by keys (since different Mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously i.e. while outputs are being fetched they are merged

Hadoop framework handles the Shuffle and Sort step ..

MapReduce Paradigm: wordcount flow



Key/value Pairs:

(fileoffset,line) → Map → (word,1) → Reduce → (word,n)

file offset is position within the file.

WordCount in Web Pages

A typical exercise for a new Google engineer in his or her first week

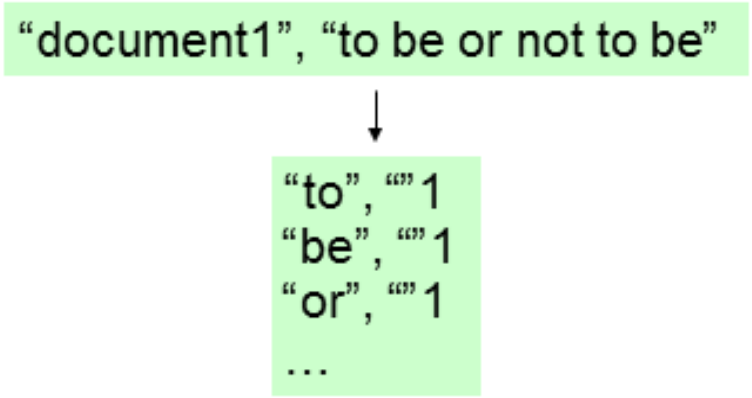
Input: files with one document per record

Specify a *map* function that takes a key/value pair

key = document URL value = document contents

Output of map function is (potentially many) key/value pairs.

In our case, output (word, "1") once per word in the document

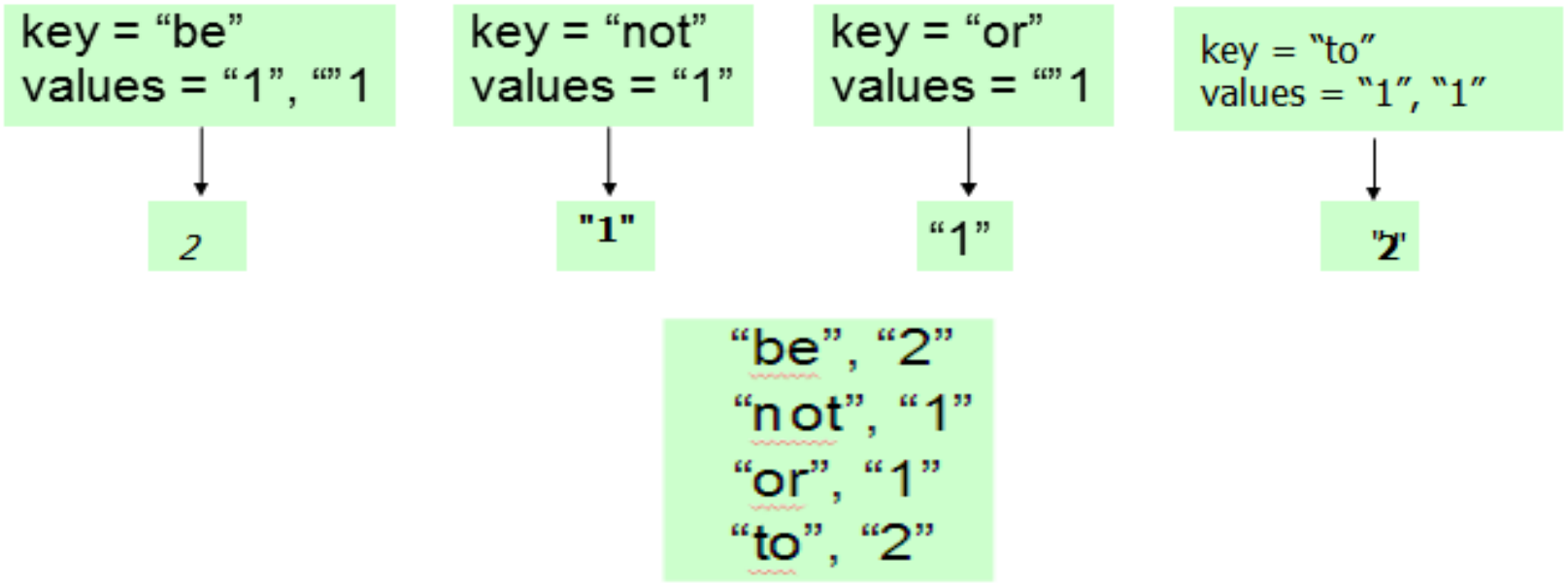


WordCount in Web Pages

MapReduce library gathers together all pairs with the same key (shuffle/sort)

Specify a *reduce* function that combines the values for a key

In our case, compute the sum



Output of reduce (usually 0 or 1 value) paired with key and saved

MapReduce Programe

A MapReduce program consists of the following 3 parts :

- Driver → main (would trigger the map and reduce methods)
- Mapper
- Reducer

it is better to include the map reduce and main methods in 3 different classes

Mapper

```
public static class Map extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {

    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

     while (tokenizer.hasMoreTokens()) {
           word.set(tokenizer.nextToken());
           output.collect(word, one);
    }
  }
 }
```

Mapper

```
//Map class header
public static class Map extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>
```

- `extends MapReduceBase`: Base class for Mapper and Reducer implementations. Provides default no-op implementations for a few methods, most non-trivial applications need to override some of them. Example

- `implements Mapper` : Interface Mapper<K1,V1,K2,V2> has the map method
  - <K1,V1,K2,V2>first pair is the input key/value pair , second is the output key/value pair.
  - <LongWritable, Text, Text, IntWritable> "The data types provided here are **Hadoop specific data types** designed for operational efficiency suited for massive parallel and lightning fast read write operations. All these data types are based out of java data types itself, for example LongWritable is the equivalent for long in java, IntWritable for int and Text for String"

Mapper

```
 //hadoop supported data types for the key/value pairs
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
//Map method header
public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
```

- `LongWritable key, Text value` : Data type of the input Key and Value to the mapper.

- `OutputCollector<Text, IntWritable> output` : output collector **<K,V>**: "collect data output by either the Mapper or the Reducer i.e. intermediate outputs or the output of the job"
  ****k** and **v** are the **output** Key and Value from the mapper. EX: <"the",1>

- `Reporter reporter`: the reporter is used to report the task status internally in Hadoop environment to avoid time outs.

Mapper

```
public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {

      //Convert the input line in Text type to a String
      String line = value.toString();
     //Use a tokenizer to split the line into words
      StringTokenizer tokenizer = new StringTokenizer(line);
   //Iterate through each word and a form key value pairs
   while (tokenizer.hasMoreTokens()) {
    //Assign each work from the tokenizer(of String type)
   //to a Text 'word'
              word.set(tokenizer.nextToken());
   //Form key value pairs for each word as <word,one> and push
   //it to the output collector
              output.collect(word, one);
      }
    }
```

Reducer

```
 //Class Header similar to the one in map
public static class Reduce extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {
//Reduce Header similar to the one in map with different
//key/value data type
//data from map will be <"word",{1,1,..}> so we get it with an
//Iterator so we can go through the sets of values
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
   //Initaize a variable 'sum' as 0
   int sum = 0;
    //Iterate through all the values with respect to a key and
   //sum up all of them
      while (values.hasNext()) {
        sum += values.next().get();
      }
    //Push to the output collector the Key and the obtained
   sum as value
      output.collect(key, new IntWritable(sum));
    }
```

Main(The Driver)

- Given the Mapper and Reducer code, the short main() starts the Map-Reduction running.
- The Hadoop system picks up a bunch of values from the command line on its own.
- Then the main() also specifies a few key parameters of the problem in the JobConf object.
- JobConf is the primary interface for a user to describe a map-reduce job to the Hadoop framework for execution (such as what Map and Reduce classes to use and the format of the input and output files).
- Other parameters, i.e. the number of machines to use, are optional and the system will determine good values for them if not specified.
- Then the framework tries to faithfully execute the job as-is described by JobConf.

Main(The Driver)

Assume we did the word count on book how many of (“the”,1) have as out put then share with other machines ?

Sloution :

we can perform a "local reduce" on the outputs produced by a single Mapper, before the intermediate values are shuffled (expensive I/O) to the Reducers.

Combiner class "mini-reduce"

use when the operation is commutative and associative. Ex:Median is an example that doesn't work !

(SUM is commutative and associative)

machine A emits <the, 1>, <the, 1>

machine B emits <the, 1>.

a Combiner on machine A emits <the, 2>. This value, along with the <the, 1> from machine B will both go to the Reducer node → we have now saved bandwidth but preserved the computation.

(This is why our reducer actually reads the value out of its input, instead of simply assuming the value is 1.)

Main

```
public static void main(String[] args) throws Exception {

//creating a JobConf object and assigning a job name for identification
purposes
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
   //Setting configuration object with the Data Type of output Key and Value
for   //map and reduce if you have diffrent type of outputs there is other set
method //for them
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
  //Providing the mapper and reducer class names
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);  //set theCombiner class
    conf.setReducerClass(Reduce.class);
// The default input format, "TextInputFormat," will load data in as //
(LongWritable, Text) pairs. The long value is the byte offset of the line in
//the file.
    conf.setInputFormat(TextInputFormat.class);
// The basic (default) instance is TextOutputFormat, which writes (key, value)
//pairs on individual lines of a text file.
    conf.setOutputFormat(TextOutputFormat.class);
//the hdfs input and output directory to be fetched from the command line
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
//submits the job to MapReduce. and returns only after the job has completed
    JobClient.runJob(conf);
  }
```

Chaining Jobs

- Not every problem can be solved with a MapReduce program, but fewer still are those which can be solved with a single MapReduce job.
- Many problems can be solved with MapReduce, by writing several MapReduce steps which run in series to accomplish a goal:

  Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3…

- You can easily chain jobs together in this fashion by writing multiple driver methods, one for each job.
- Call the first driver method, which uses JobClient.runJob() to run the job and wait for it to complete. When that job has completed, then call the next driver method, which creates a new JobConf object referring to different instances of Mapper and Reducer, etc.

How it Works?

- Purpose
  - Simple serialization for keys, values, and other data
- Interface Writable
  - Read and write binary format
  - Convert to String for text formats

Interface

- Text
  - Stores text using standard UTF8 encoding. It provides methods to serialize, deserialize, and compare texts at byte level.
  - Methods:
    - **set(byte[] utf8)**

Classes

- OutputCollector
- Collects data output by either the Mapper or the Reducer i.e. intermediate outputs or the output of the job.
- Methods:
  - **collect(K key, V value)**
- We have linked class and Interfaces in previous slides.

Exercises

- Get WordCount running!
- Extend WordCount to count Bigrams:
  - Bigrams are simply sequences of two consecutive words.
    - For example, the previous sentence contains the following bigrams: "*Bigrams are*", "*are simply*", "*simply sequences*", "*sequence of*", etc.