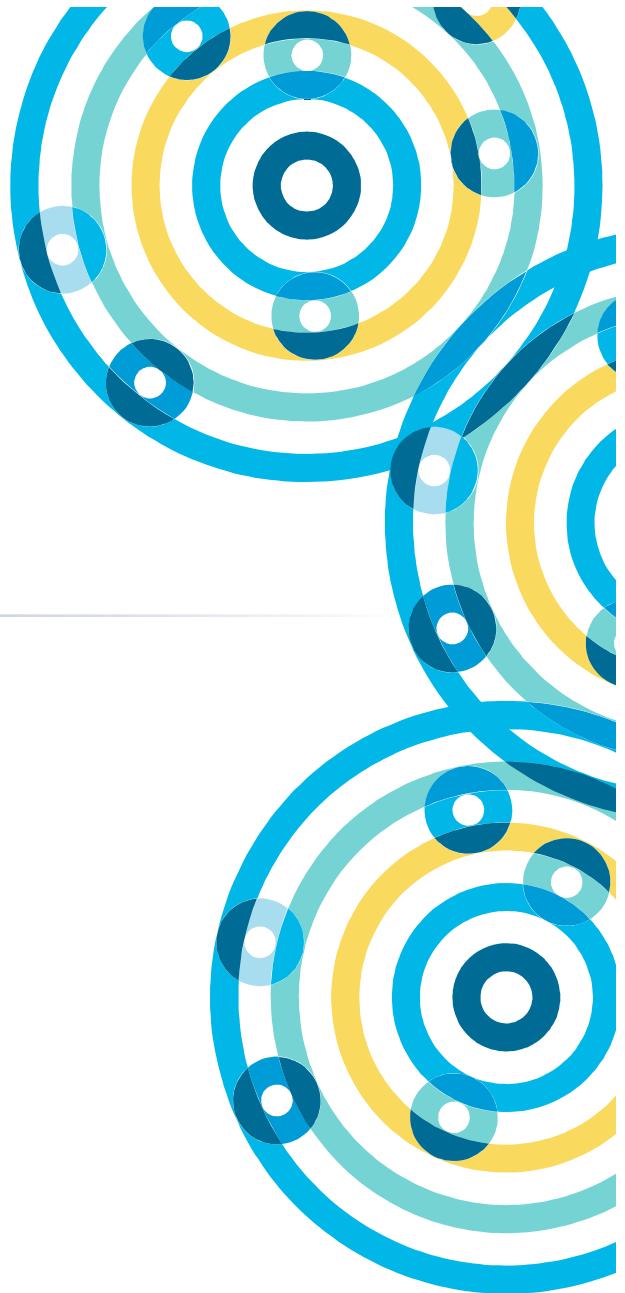


MapReduce



What Is MapReduce?

- **MapReduce is a method for distributing a task across multiple nodes**
- **Each node processes data stored on that node**
 - Where possible
- **Consists of two phases:**
 - Map
 - Reduce

MapReduce: Terminology

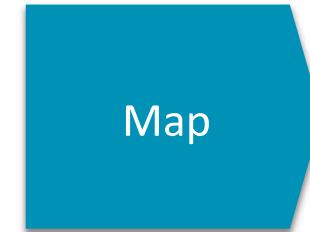
- A **job** is a ‘full program’
 - A complete execution of Mappers and Reducers over a dataset
 - In MapReduce 2, the term *application* is often used in place of ‘job’
- A **task** is the execution of a single Mapper or Reducer over a slice of data
- A **task attempt** is a particular instance of an attempt to execute a task
 - There will be at least as many task attempts as there are tasks
 - If a task attempt fails, another will be started by the JobTracker
 - *Speculative execution* (see later) can also result in more task attempts than completed tasks

A task can be attempted several times because of failure, or speculative execution i.e.

Hadoop Components: MapReduce

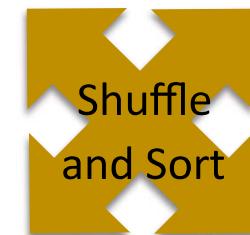
■ The Mapper

- Each Map task (typically) operates on a single HDFS block
- Map tasks (usually) run on the node where the block is stored



■ Shuffle and Sort

- Sorts and consolidates intermediate data from all mappers
- Happens as Map tasks complete and before Reduce tasks start

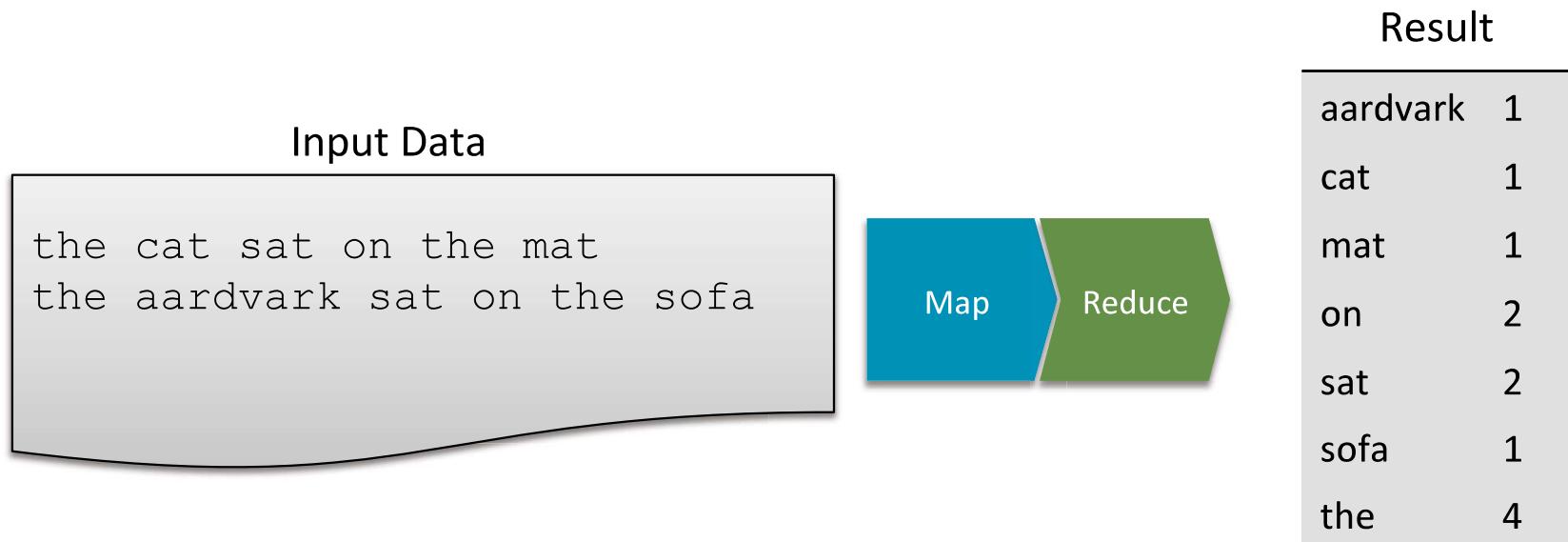


■ The Reducer

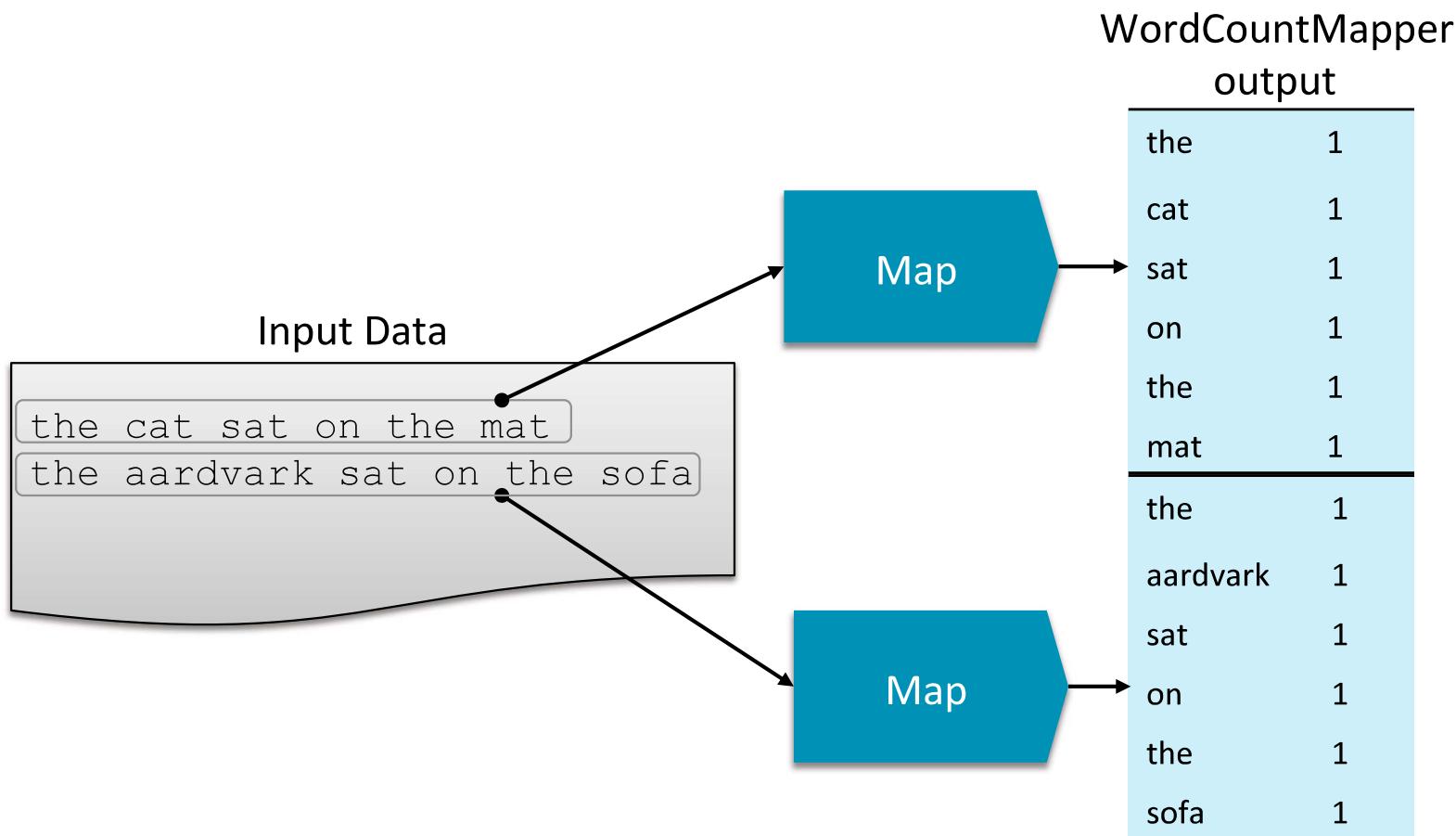
- Operates on shuffled/sorted intermediate data (Map task output)
- Produces final output



Example: Word Count



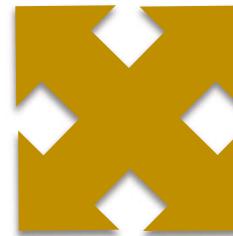
Example: The WordCount Mapper



Example: Shuffle & Sort

Mapper Output

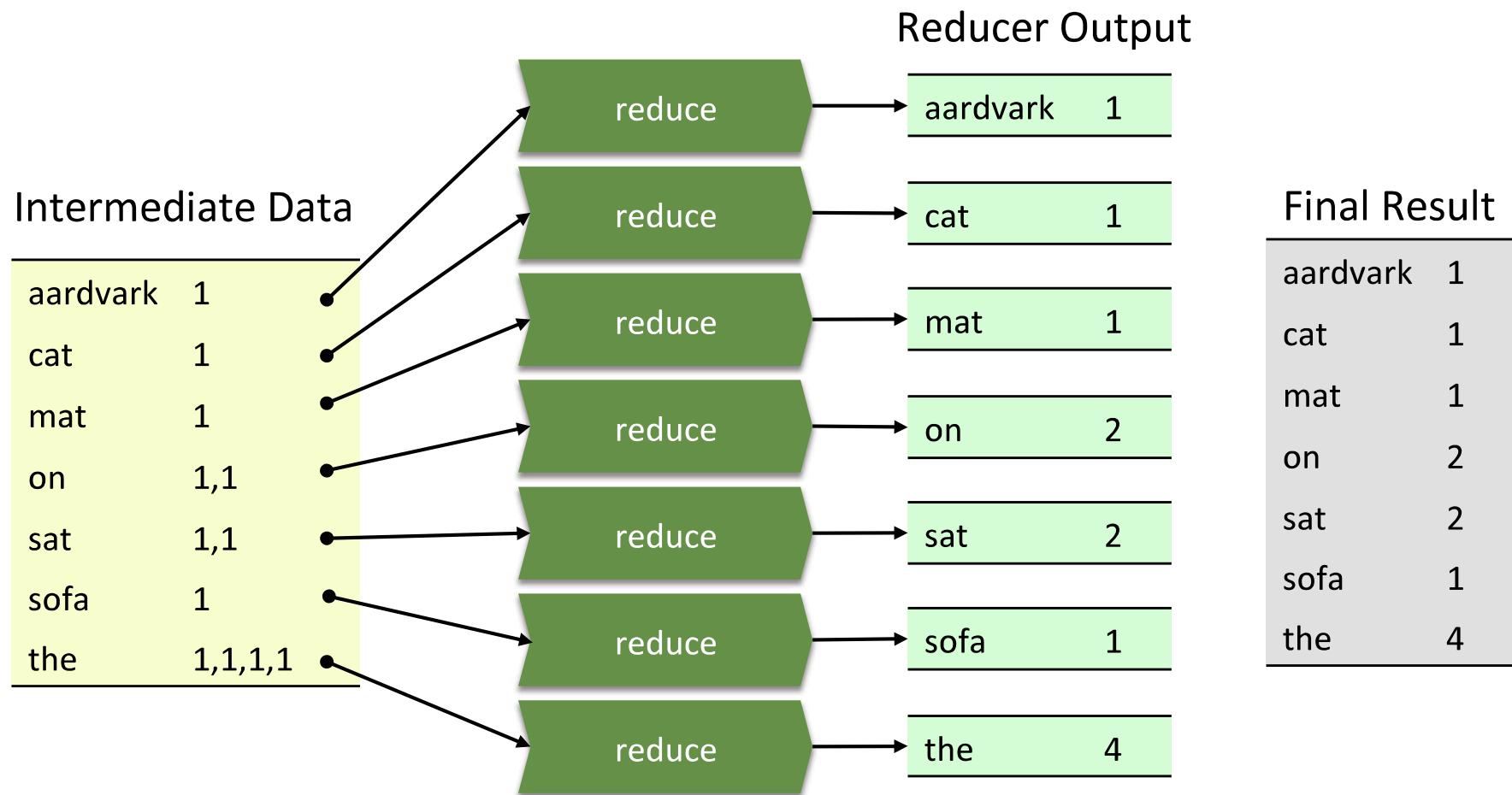
the	1
cat	1
sat	1
on	1
the	1
mat	1
the	1
aardvark	1
sat	1
on	1
the	1
sofa	1



Intermediate Data

aardvark	1
cat	1
mat	1
on	1,1
sat	1,1
sofa	1
the	1,1,1,1

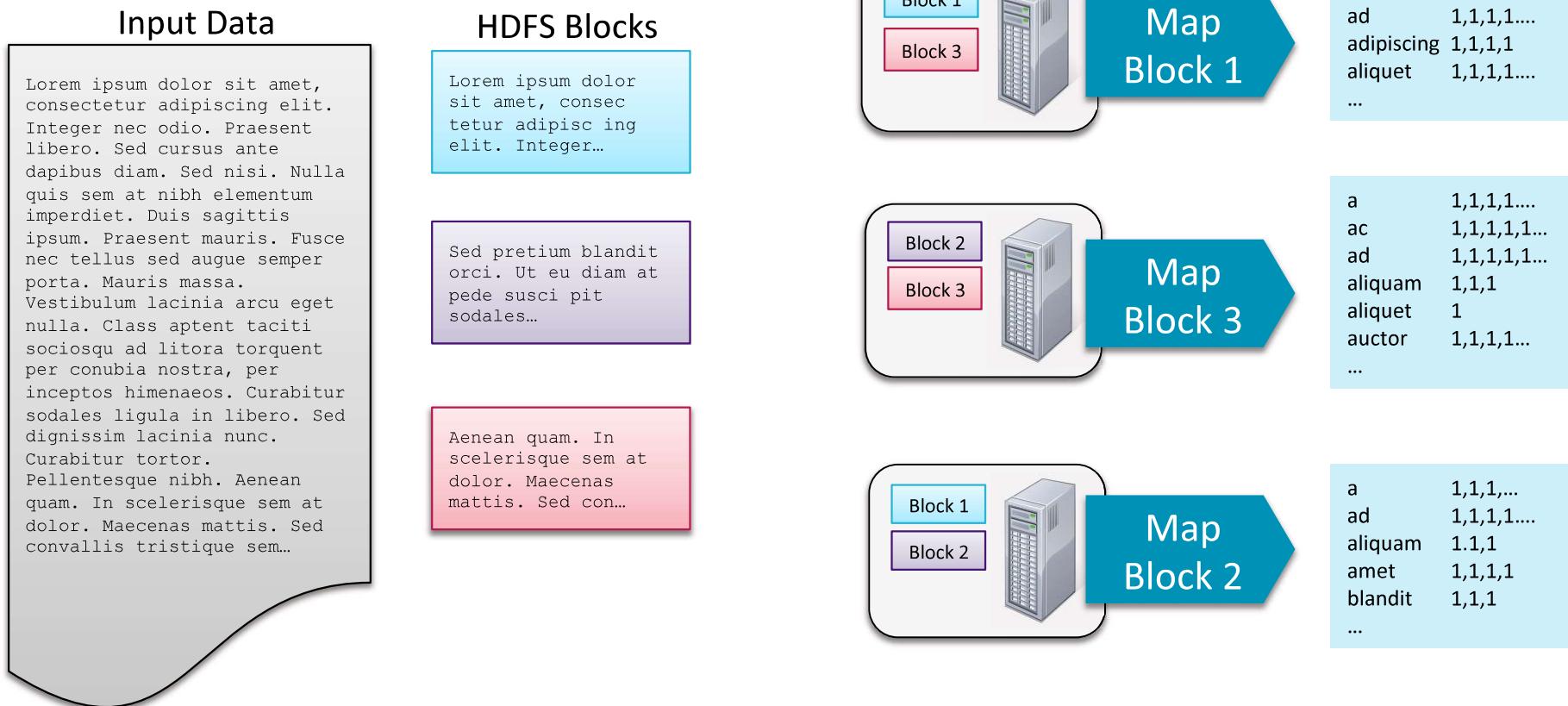
Example: SumReducer



Mappers Run in Parallel

Is shuffle & sort centralized? mappers sorted ou

- Hadoop runs Map tasks on the slave node where the block is stored (when possible)
 - Many Mappers can run in parallel
 - Minimizes network traffic



MapReduce: The Mapper

- Hadoop attempts to ensure that Mappers run on nodes which hold their portion of the data locally, to avoid network traffic
 - Multiple Mappers run in parallel, each processing a portion of the input data
- The Mapper reads data in the form of key/value pairs
 - The Mapper may use or completely ignore the input key
 - For example, a standard pattern is to read one line of a file at a time
 - The key is the byte offset into the file at which the line starts
 - The value is the contents of the line itself
 - Typically the key is considered irrelevant
- If the Mapper writes anything out, the output must be in the form of key/value pairs

key-value pairs

MapReduce: The Reducer

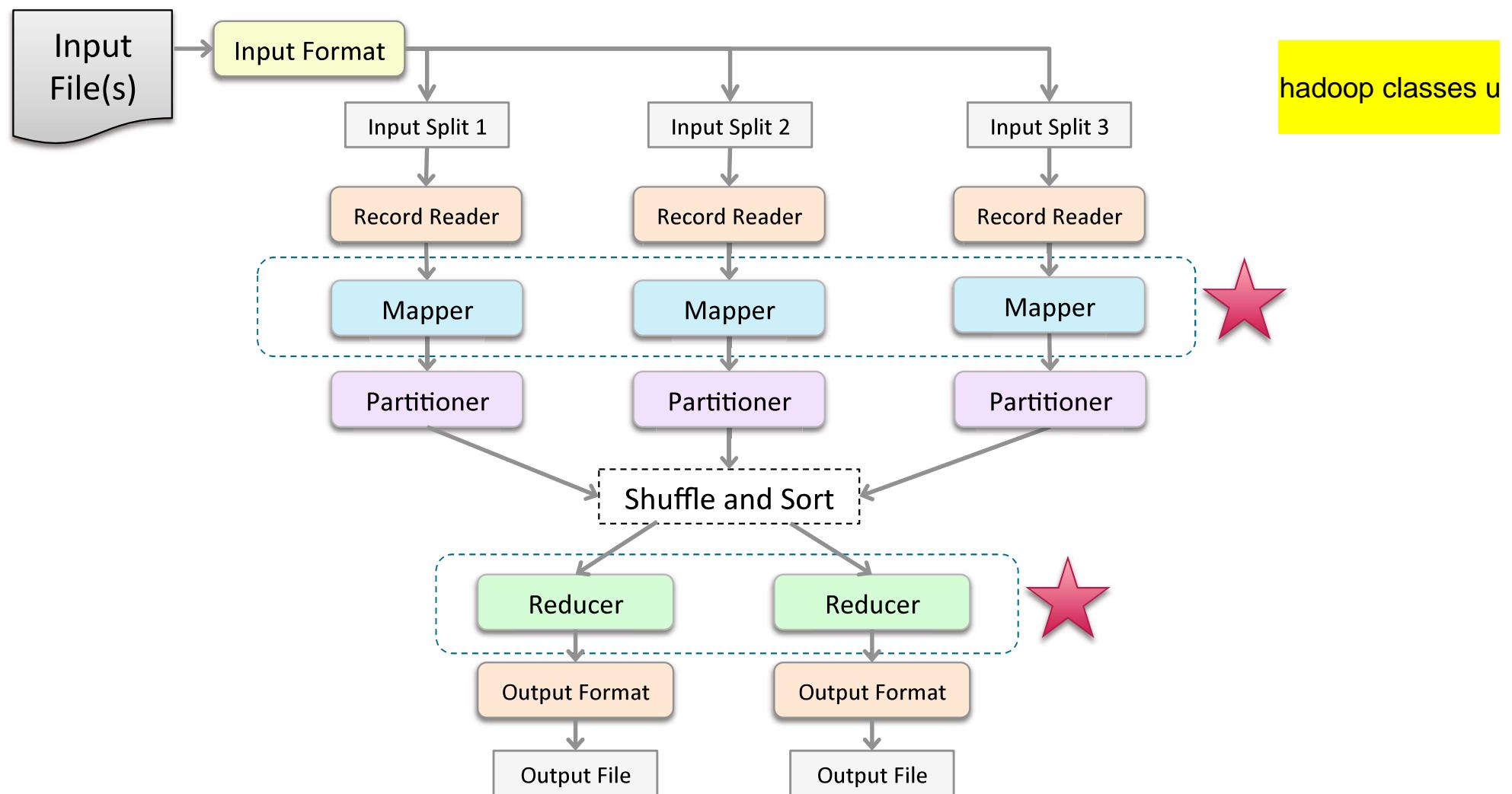
- After the Map phase is over, all intermediate values for a given intermediate key are combined together into a list
- This list is given to a Reducer
 - There may be a single Reducer, or multiple Reducers
 - All values associated with a particular intermediate key are guaranteed to go to the same Reducer
 - The intermediate keys, and their value lists, are passed to the Reducer in sorted key order
 - This step is known as the ‘shuffle and sort’
- The Reducer outputs zero or more final key/value pairs
 - These are written to HDFS
 - In practice, the Reducer usually emits a single key/value pair for each input key

Number of reducers?

Why Do We Care About Counting Words?

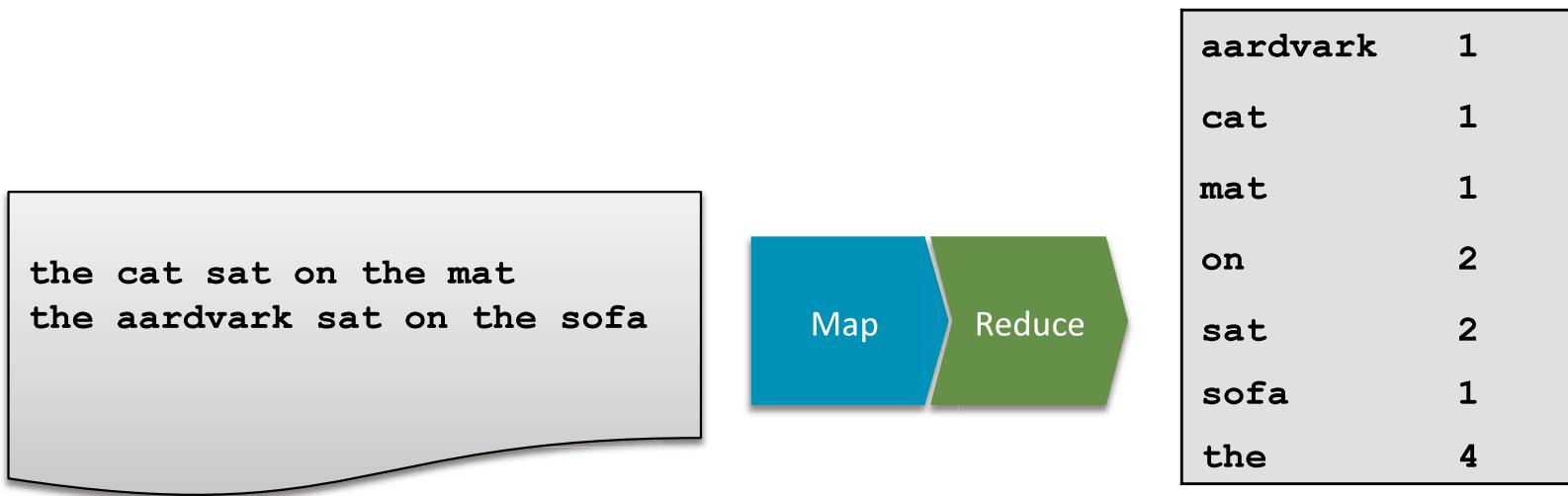
- **Word count is challenging over massive amounts of data**
 - Using a single compute node would be too time-consuming
 - Using distributed nodes requires moving data
 - Number of unique words can easily exceed available memory
 - Would need to store to disk
- **Statistics are simple aggregate functions**
 - Distributive in nature
 - e.g., max, min, sum, count
- **MapReduce breaks complex tasks down into smaller elements which can be executed in parallel**
- **Many common tasks are very similar to word count**
 - e.g., log file analysis

Review: The MapReduce Flow



A Sample MapReduce Program: WordCount

- In an earlier chapter, you ran a sample MapReduce program
 - WordCount, which counted the number of occurrences of each unique word in a set of files
- In this chapter, we will examine the code for WordCount
 - This will demonstrate the Hadoop API



Our MapReduce Program: WordCount

- To investigate the API, we will dissect the WordCount program we covered in the previous chapter
- This consists of three portions
 - The driver code
 - Code that runs on the client to configure and submit the job
 - The Mapper
 - The Reducer
- Before we look at the code, we need to cover some basic Hadoop API concepts

3 Parts

Getting Data to the Mapper

- **The data passed to the Mapper is specified by an *InputFormat***

- Specified in the driver code
 - Defines the location of the input data
 - Typically a file or directory
 - Determines how to split the input data into *input splits*
 - Each Mapper deals with a single input split
 - Creates a RecordReader object
 - RecordReader parses the input data into key/value pairs to pass to the Mapper

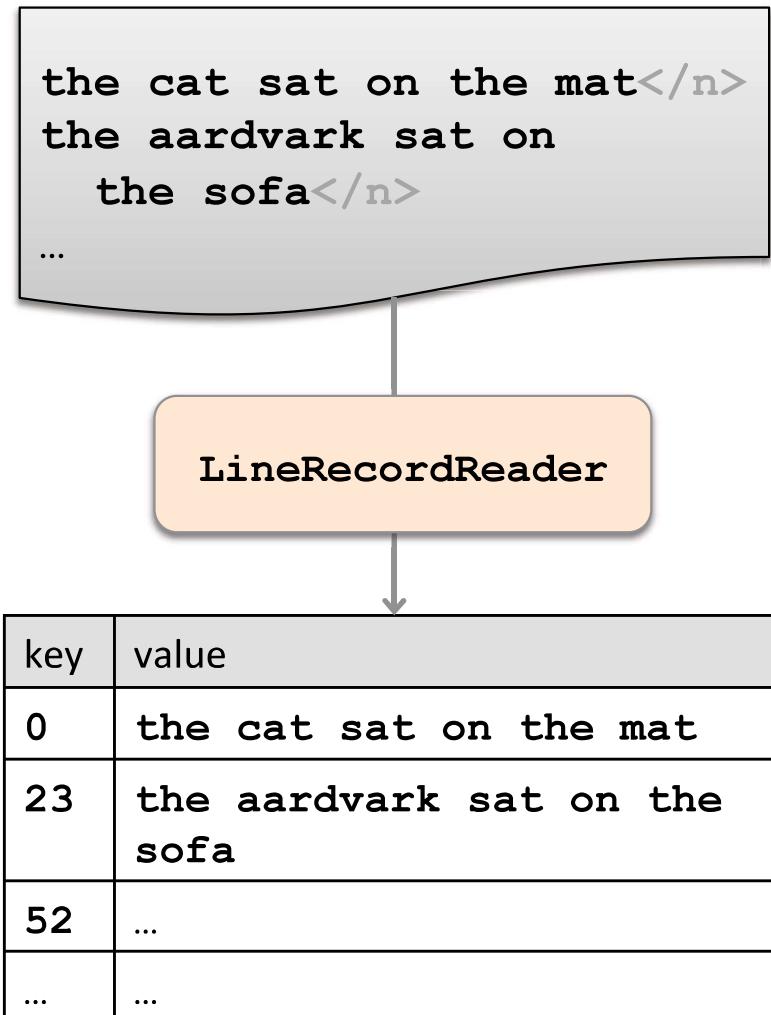
InputFormat

RecordReader

Example: TextInputFormat

- **TextInputFormat**

- The default
- Creates LineRecordReader objects
- Treats each \n-terminated line of a file as a value
- Key is the byte offset of that line within the file



Other Standard InputFormats

- **FileInputFormat**

- Abstract base class used for all file-based InputFormats

- **KeyValueTextInputFormat**

- Maps \n-terminated lines as ‘key [separator] value’
 - By default, [separator] is a tab

- **SequenceFileInputFormat**

- Binary file of (key, value) pairs with some additional metadata

- **SequenceFileAsTextInputFormat**

- Similar, but maps (key.toString(), value.toString())

Keys and Values are Objects

- **Keys and values in Hadoop are Java Objects**
 - Not primitives
- **Values are objects which implement Writable**
- **Keys are objects which implement WritableComparable**

What is `Writable`?

- **The `Writable` interface makes serialization quick and easy for Hadoop**
- **Any value's type must implement the `Writable` interface**
- **Hadoop defines its own ‘box classes’ for strings, integers, and so on**
 - `IntWritable` for ints
 - `LongWritable` for longs
 - `FloatWritable` for floats
 - `DoubleWritable` for doubles
 - `Text` for strings
 - Etc.

What is WritableComparable?

- A **WritableComparable** is a **Writable** which is also **Comparable**
 - Two WritableComparables can be compared against each other to determine their ‘order’
 - Keys must be WritableComparables because they are passed to the Reducer in sorted order
 - We will talk more about WritableComparables later
- Note that despite their names, all Hadoop box classes implement both **Writable** and **WritableComparable**
 - For example, IntWritable is actually a WritableComparable

The Driver: Complete Code

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {

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

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.printf(
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());
            return -1;
        }
        Job job = new Job(getConf());
        job.setJarByClass(WordCount.class);
        job.setJobName("Word Count");

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

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

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

Driver Class Definition

```
public class WordCount extends Configured implements Tool {  
  
    public static void main(String[] args) {  
        int exitCode = ToolRunner.run(getConf(), WordCount.class, args);  
        System.exit(exitCode);  
    }  
  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            return -1;  
        }  
        Job job = new Job(getConf());  
        job.setJarByClass(WordCount.class); job.setJobName("Word Count");  
  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.setMapperClass(WordMapper.class);  
        job.setReducerClass(SumReducer.class);  
        job.setMapOutputKeyClass(Text.class);  
        job.setMapOutputValueClass(IntWritable.class);  
  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        boolean success = job.waitForCompletion(true);  
        return success ? 0 : 1;  
    }  
}
```

The driver class implements the Tool interface and extends the Configured class.

Main Method

```
public class WordCount extends Configured implements Tool {  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new Configuration(),  
            new WordCount(), args);  
        System.exit(exitCode);  
    }  
  
    public int run(String[] args) throws Exception {  
        if (args.length != 2)  
            System.out.printf("Usage: %s [g  
        return -1;  
    }  
    Job job = new Job(getConf());  
    job.setJarByClass(WordCount.class);  
    job.setJobName("Word Count");  
  
    FileInputFormat.setInputPaths(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
    job.setMapperClass(WordMapper.class);  
    job.setReducerClass(SumReducer.class);  
    job.setMapOutputKeyClass(Text.class);  
    job.setMapOutputValueClass(IntWritable.class);  
  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
  
    ...  
}
```

The driver main method calls ToolRunner.run.

Run Method

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

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.printf(
                "Usage: %s [generic options] <input dir> <output dir>\n",
                getClass().getSimpleName());
            return -1;
        }
        Job job = new Job(getConf());
        job.setJarByClass(WordCount.class);
        job.setJobName("Word Count");

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        ...
    }
}
```

The driver `run` method creates, configures, and submits the job.

Creating a New Job Object

- **The Job class allows you to set configuration options for your MapReduce job**
 - The classes to be used for your Mapper and Reducer
 - The input and output directories
 - Many other options
- **Any options not explicitly set in your driver code will be read from your Hadoop configuration files**
 - Usually located in /etc/hadoop/conf
- **Any options not specified in your configuration files will use Hadoop's default values**
- **You can also use the Job object to submit the job, control its execution, and query its state**

Configuring the Job: Specifying the InputFormat

- The default InputFormat (`TextInputFormat`) will be used unless you specify otherwise
- To use an InputFormat other than the default, use e.g.

```
job.setInputFormatClass(KeyValueTextInputFormat.class)
```

Configuring the Job: Determining Which Files To Read

- By default, `FileInputFormat.setInputPaths()` will read all files from a specified directory and send them to Mappers
 - Exceptions: items whose names begin with a period (.) or underscore (_)
 - Globs can be specified to restrict input
 - For example, `/2010/*/01/*`
- Alternatively, `FileInputFormat.addInputPath()` can be called multiple times, specifying a single file or directory each time
- More advanced filtering can be performed by implementing a **PathFilter**
 - Interface with a method named `accept`
 - Takes a path to a file, returns `true` or `false` depending on whether or not the file should be processed

Configuring the Job: Specifying Final Output With OutputFormat

- **FileOutputFormat.setOutputPath() specifies the directory to which the Reducers will write their final output**
- **The driver can also specify the format of the output data**
 - Default is a plain text file
 - Could be explicitly written as
`job.setOutputFormatClass(TextOutputFormat.class)`
- **We will discuss OutputFormats in more depth in a later chapter**

Configuring the Job: Specifying the Mapper and Reducer Classes

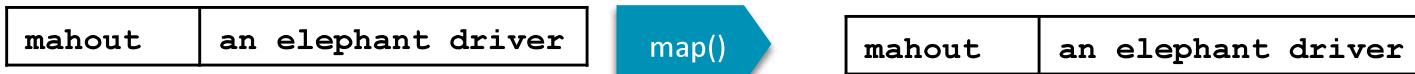
```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.out.printf("Usage: WordCount <input dir> <output dir>\n");  
            System.exit(-1);  
        }  
        Job job = new Job();  
        job.setJarByClass(WordCount.class);  
        job.setJobName("Word Count");  
  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.setMapperClass(WordMapper.class);  
        job.setReducerClass(SumReducer.class);  
  
        job  
        job  
        job  
        job  
        boolean success = job.waitForCompletion(true);  
        System.exit(success ? 0 : 1);  
    }  
}
```

Give the `Job` object information about which classes are to be instantiated as the Mapper and Reducer.

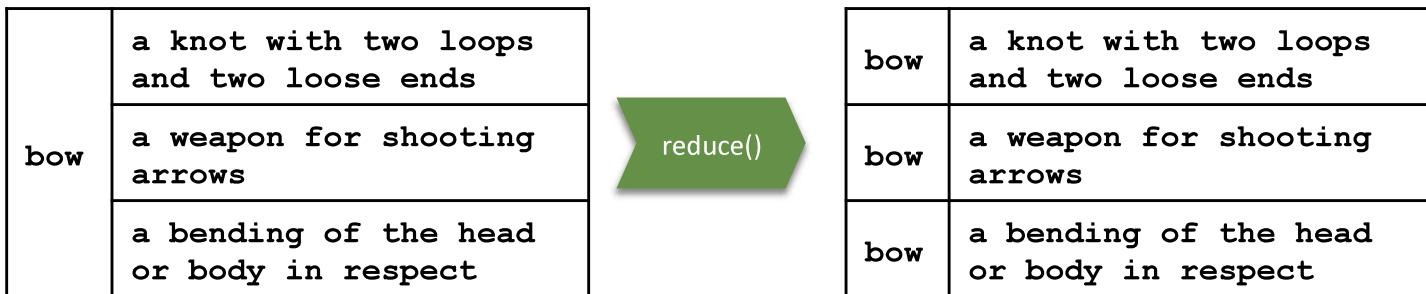
Default Mapper and Reducer Classes

- Setting the Mapper and Reducer classes is optional
- If not set in your driver code, Hadoop uses its defaults

- IdentityMapper



- IdentityReducer



Configuring the Job: Specifying the Intermediate Data Types

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.out.printf("Usage: WordCount <input dir> <output dir>\n");  
            System.exit(-1);  
        }  
        Job job = new Job();  
        job.setJarByClass(WordCount.class);  
        job.setJobName("Word Count");  
  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.setMapperClass(WordMapper.class);  
        job.setReducerClass(SumReducer.class);  
  
        job.setMapOutputKeyClass(Text.class);  
        job.setMapOutputValueClass(IntWritable.class);  
  
        job  
        job  
        job  
        job  
        job  
    }  
}
```

Specify the types for the intermediate output keys and values produced by the Mapper.

Configuring the Job: Specifying the Final Output Data Types

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.out.printf("Usage: WordCount <input dir> <output dir>\n");  
            System.exit(-1);  
        }  
        Job job = new Job();  
        job.setJarByClass(WordCount.class);  
        job.setJobName("Word Count");  
  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.setMapperClass(WordMapper.class);  
        job.setReducerClass(WordReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        boolean success = job.waitForCompletion(true);  
        System.exit(success ? 0 : 1);  
    }  
}
```

Specify the types for the Reducer's output keys and values.

Running The Job (1)

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.out.printf("Usage: WordCount <input dir> <output dir>\n");  
            System.exit(-1);  
        }  
        Job job = new Job();  
        job.setJarByClass(WordCount.class);  
        job.setJobName("Word Count");  
  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    }  
}
```

Start the job and wait for it to complete. Parameter is a Boolean, specifying verbosity: if true, display progress to the user. Finally, exit with a return code.

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

Running The Job

- **There are two ways to run your MapReduce job:**
 - `job.waitForCompletion()`
 - Blocks (waits for the job to complete before continuing)
 - `job.submit()`
 - Does not block (driver code continues as the job is running)
- **The client determines the proper division of input data into InputSplits, and then sends the job information to the JobTracker daemon on the cluster**

ToolRunner Command Line Options

- ToolRunner allows the user to specify configuration options on the command line
- Commonly used to specify Hadoop properties using the **-D** flag
 - Will override any default or site properties in the configuration
 - But will *not* override those set in the driver code

```
$ hadoop jar myjar.jar MyDriver \
-D mapred.reduce.tasks=10 myinputdir myoutputdir
```

- Note that **-D** options must appear before any additional program arguments
- Can specify an XML configuration file with **-conf**
- Can specify the default filesystem with **-fs uri**
 - Shortcut for **-D fs.default.name=uri**

-D options must appear before any additional arguments

fs.default.name has been deprecated and replaced by fs.default

Make sure you pass conf object initialized by ToolRunner:Conf

The Mapper: Complete Code

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();

        for (String word : line.split("\\W+")) {
            if (word.length() > 0) {

                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

The Mapper: Class Declaration (1)

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();

        for (String word : line.split("\\w+")) {
            if (word.length() > 0) {

                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

Mapper classes extend the Mapper base class.

The Mapper: Class Declaration (2)

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, IntWritable
        co Input key and
        value types
        nco Intermediate key
        and value types
        String intermediateKey) throws IOException, InterruptedException {
        for (String word : value.toString().split(" ")) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

Specify generic types that declare four type parameters: the input key and value types, and the output (intermediate) key and value types. Keys must be WritableComparable, and values must be Writable.

The map Method

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String word = value.toString();
        for (String w : words) {
            if (w.equals(word)) {
                context.write(new Text(w), new IntWritable(1));
            }
        }
    }
}
```

The map method is passed a key, a value, and a Context object. The Context is used to write the intermediate data. It also contains information about the job's configuration.

The map Method: Processing The Line (1)

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();

        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (Character.isLetter(c)) {
                word = Character.toLowerCase(c);
                for (int j = i + 1; j < line.length(); j++) {
                    char nextChar = line.charAt(j);
                    if (Character.isLetter(nextChar)) {
                        word += nextChar;
                    } else {
                        break;
                    }
                }
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

value is a Text object, so retrieve the string it contains.

The map Method: Processing The Line (2)

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();

        for (String word : line.split("\\w+")) {
            if (word.length() > 0) {

                cont
            }
        }
    }
}
```

Split the string up into words using a regular expression with non-alphanumeric characters as the delimiter, and then loop through the words.

The map Method: Outputting Intermediate Data

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

```
@O
pu
```

To emit a (key, value) pair, call the `write` method of the Context object. The key will be the word itself, the value will be the number 1. Recall that the output key must be a `WritableComparable`, and the value must be a `Writable`.

```
    context.write(new Text(word), new IntWritable(1));
}
}
}
```

The Reducer: Complete Code

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int wordCount = 0;

        for (IntWritable value : values) {
            wordCount += value.get();
        }

        context.write(key, new IntWritable(wordCount));
    }
}
```

The Reducer: Class Declaration

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    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));
    }
}
```

Reducer classes extend the `Reducer` base class. The four generic type parameters are: input (intermediate) key and value types, and final output key and value types.

Intermediate key and value types

Output key and value types

The reduce Method

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
        context.write(key, new IntWritable(wordCount));
    }
}
```

The reduce method receives a key and an Iterable collection of objects (which are the values emitted from the Mappers for that key); it also receives a Context object.

The reduce Method: Processing The Values

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
    }
}
```

We use the Java for-each syntax to step through all the elements in the collection. In our example, we are merely adding all the values together. We use `value.get()` to retrieve the actual numeric value each time.

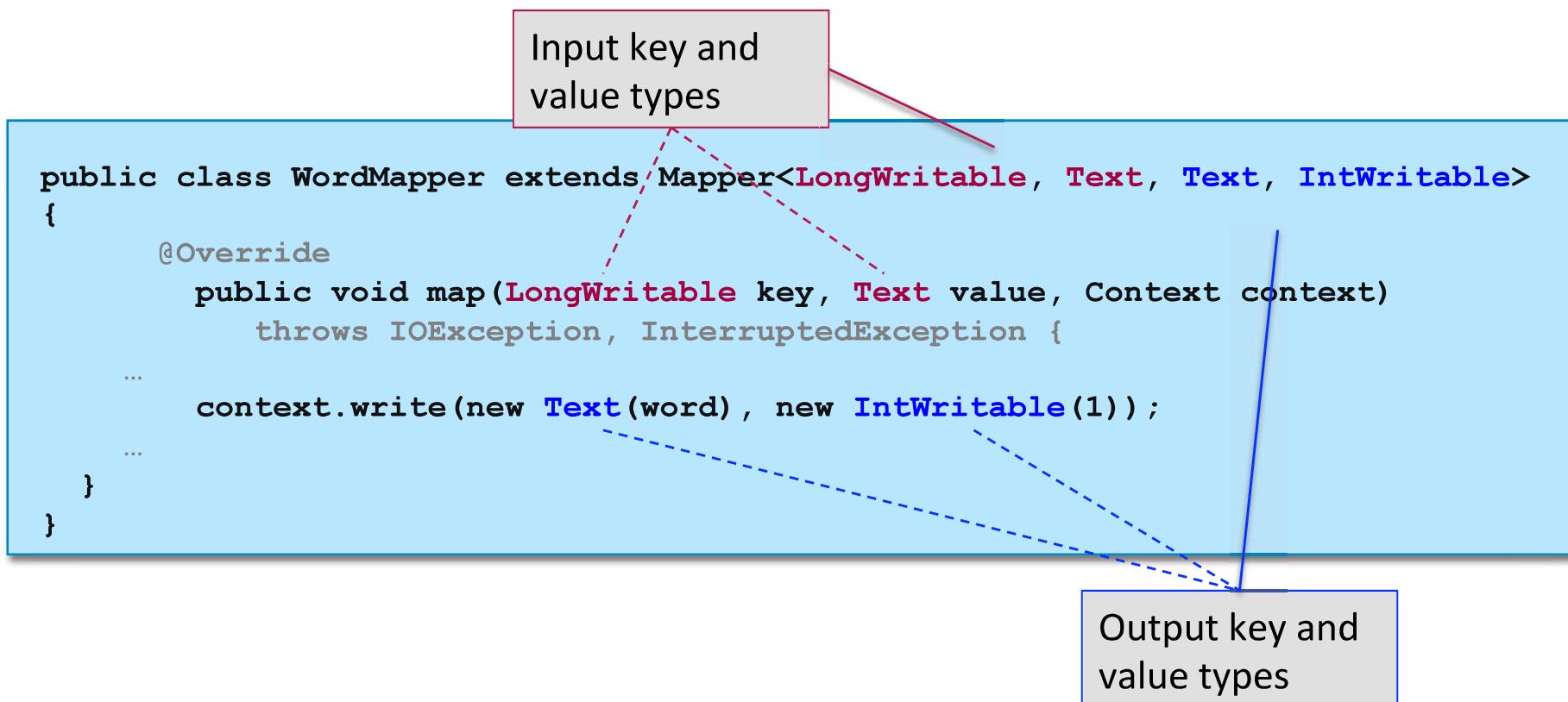
The reduce Method: Writing The Final Output

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
        context.write(key, new IntWritable(wordCount));
    }
}
```

Finally, we write the output key-value pair to HDFS using the `write` method of our `Context` object.

Ensure Types Match (1)

- Mappers and Reducers declare input and output type parameters
- These must match the types used in the class



Ensure Types Match (2)

- Output types must also match those set in the driver

```
public class WordMapper extends Mapper<LongWritable,  
Text, Text, IntWritable> {  
    ...  
}
```

Mapper

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        ...  
        job.setMapOutputKeyClass(Text.class);  
        job.setMapOutputValueClass(IntWritable.class);  
  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        ...  
    }  
}
```

driver code

```
public class SumReducer extends Reducer<Text,  
IntWritable, Text, IntWritable> {  
    ...  
}
```

Reducer

What Is The Old API?

- **When Hadoop 0.20 was released, a ‘New API’ was introduced**
 - Designed to make the API easier to evolve in the future
 - Favors abstract classes over interfaces
- **Some developers still use the Old API**
 - Until CDH4, the New API was not absolutely feature-complete
- **All the code examples in this course use the New API**

New API vs. Old API: Some Key Differences (1)

New API	Old API
<pre>import org.apache.hadoop.mapreduce.*</pre> Driver code: <pre>Configuration conf = new Configuration(); Job job = new Job(conf); job.setJarByClass(Driver.class); job.setSomeProperty(...); ... job.waitForCompletion(true);</pre>	<pre>import org.apache.hadoop.mapred.*</pre> Driver code: <pre>JobConf conf = new JobConf(Driver.class); conf.setSomeProperty(...); ... JobClient.runJob(conf);</pre>
Mapper: <pre>public class MyMapper extends Mapper { public void map(Keytype k, Valuetype v, Context c) { ... c.write(key, val); } }</pre>	Mapper: <pre>public class MyMapper extends MapReduceBase implements Mapper { public void map(Keytype k, Valuetype v, OutputCollector o, Reporter r) { ... o.collect(key, val); } }</pre>

New API vs. Old API: Some Key Differences (2)

New API	Old API
Reducer: <pre>public class MyReducer extends Reducer { public void reduce(Keytype k, Iterable<Valuetype> v, Context c) { for(Valuetype eachval : v) { // process eachval c.write(key, val); } } }</pre>	Reducer: <pre>public class MyReducer extends MapReduceBase implements Reducer { public void reduce(Keytype k, Iterator<Valuetype> v, OutputCollector o, Reporter r) { while(v.hasNext()) { // process v.next() o.collect(key, val); } } }</pre>
setup(Context c) (See later)	configure(JobConf job)
cleanup(Context c) (See later)	close()

MRv1 vs MRv2, Old API vs New API

- There is a lot of confusion about the New and Old APIs, and MapReduce version 1 and MapReduce version 2
- The chart below should clarify what is available with each version of MapReduce

	Old API	New API
MapReduce v1	✓	✓
MapReduce v2	✓	✓

- Summary: Code using either the Old API or the New API will run under MRv1 and MRv2

The setup Method

- It is common to want your Mapper or Reducer to execute some code before the `map` or `reduce` method is called for the first time
 - Initialize data structures
 - Read data from an external file
 - Set parameters
- The `setup` method is run before the `map` or `reduce` method is called for the first time

```
public void setup(Context context)
```

The `cleanup` Method

- Similarly, you may wish to perform some action(s) after all the records have been processed by your Mapper or Reducer
- The `cleanup` method is called before the Mapper or Reducer terminates

```
public void cleanup(Context context) throws  
    IOException, InterruptedException
```

Passing Parameters

```
public class MyDriverClass {  
    public int main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
        conf.setInt ("paramname", value);  
        Job job = new Job(conf);  
        ...  
        boolean success = job.waitForCompletion(true);  
        return success ? 0 : 1;  
    }  
}
```

```
public class MyMapper extends Mapper {  
  
    public void setup(Context context) {  
        Configuration conf = context.getConfiguration();  
        int myParam = conf.getInt("paramname", 0);  
        ...  
    }  
    public void map...  
}
```

Reuse of Objects is Good Practice (1)

- It is generally good practice to reuse objects
 - Instead of creating many new objects
- Example: Our original WordCount Mapper code

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String word = value.toString();
        for (String line : lines) {
            if (line.startsWith(word)) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

Each time the `map()` method is called, we create a new `Text` object and a new `IntWritable` object.

Reuse of Objects is Good Practice (2)

- Instead, this is better practice:

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text wordObject = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();

        for (String word : line.split("\\w+")) {
            if (word.length() > 0) {
                wordObject.set(word);
                context.write(wordObject, one);
            }
        }
    }
}
```

Create objects for the key and value outside of your map () method

Reuse of Objects is Good Practice (3)

- Instead, this is better practice:

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text wordObject = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String word = value.toString();
        if (word.length() > 0) {
            wordObject.set(word);
            context.write(wordObject, one);
        }
    }
}
```

Within the `map()` method, populate the objects and write them out. Hadoop will take care of serializing the data so it is perfectly safe to re-use the objects.

Object Reuse: Caution!

- **Hadoop re-uses objects all the time**
- **For example, each time the Reducer is passed a new value, the same object is reused**
- **This can cause subtle bugs in your code**
 - For example, if you build a list of value objects in the Reducer, each element of the list will point to the same underlying object
 - Unless you do a deep copy

Map-Only MapReduce Jobs

- **There are many types of job where only a Mapper is needed**

- **Examples:**

- Image processing
 - File format conversion
 - Input data sampling
 - ETL

Why mappers only? Shu

Creating Map-Only Jobs

- To create a Map-only job, set the number of Reducers to 0 in your Driver code

```
job.setNumReduceTasks(0);
```

- Call the Job.setOutputKeyClass and Job.setOutputValueClass methods to specify the output types
 - Not the Job.setMapOutputKeyClass and Job.setMapOutputValueClass methods
- Anything written using the Context.write method in the Mapper will be written to HDFS
 - Rather than written as intermediate data
 - One file per Mapper will be written

How Many Reducers Do You Need?

- An important consideration when creating your job is to determine the number of Reducers specified
- Default is a single Reducer
- With a single Reducer, one task receives *all* keys in sorted order
 - This is sometimes advantageous if the output must be in completely sorted order
 - Can cause significant problems if there is a large amount of intermediate data
 - Node on which the Reducer is running may not have enough disk space to hold all intermediate data
 - The Reducer will take a long time to run

Jobs Which Require a Single Reducer

- If a job needs to output a file where all keys are listed in sorted order, a single Reducer must be used
- Alternatively, the `TotalOrderPartitioner` can be used
 - Uses an externally generated file which contains information about intermediate key distribution
 - Partitions data such that all keys which go to the first Reducer are smaller than any which go to the second, etc
 - In this way, multiple Reducers can be used
 - Concatenating the Reducers' output files results in a totally ordered list

Jobs Which Require a Fixed Number of Reducers

- Some jobs will require a specific number of Reducers
- Example: a job must output one file per day of the week
 - Key will be the weekday
 - Seven Reducers will be specified
 - A Partitioner will be written which sends one key to each Reducer

Jobs With a Variable Number of Reducers (1)

- **Many jobs can be run with a variable number of Reducers**
- **Developer must decide how many to specify**
 - Each Reducer should get a reasonable amount of intermediate data, but not too much
 - Chicken-and-egg problem
- **Typical way to determine how many Reducers to specify:**
 - Test the job with a relatively small test data set
 - Extrapolate to calculate the amount of intermediate data expected from the ‘real’ input data
 - Use that to calculate the number of Reducers which should be specified

Jobs With a Variable Number of Reducers (2)

MRv1: Reducers Slots

- **Note: you should take into account the number of Reduce slots likely to be available on the cluster**
 - If your job requires one more Reduce slot than there are available, a second ‘wave’ of Reducers will run
 - Consisting just of that single Reducer
 - Potentially doubling the amount of time spent on the Reduce phase
 - In this case, increasing the number of Reducers further may cut down the time spent in the Reduce phase
 - Two or more waves will run, but the Reducers in each wave will have to process less data