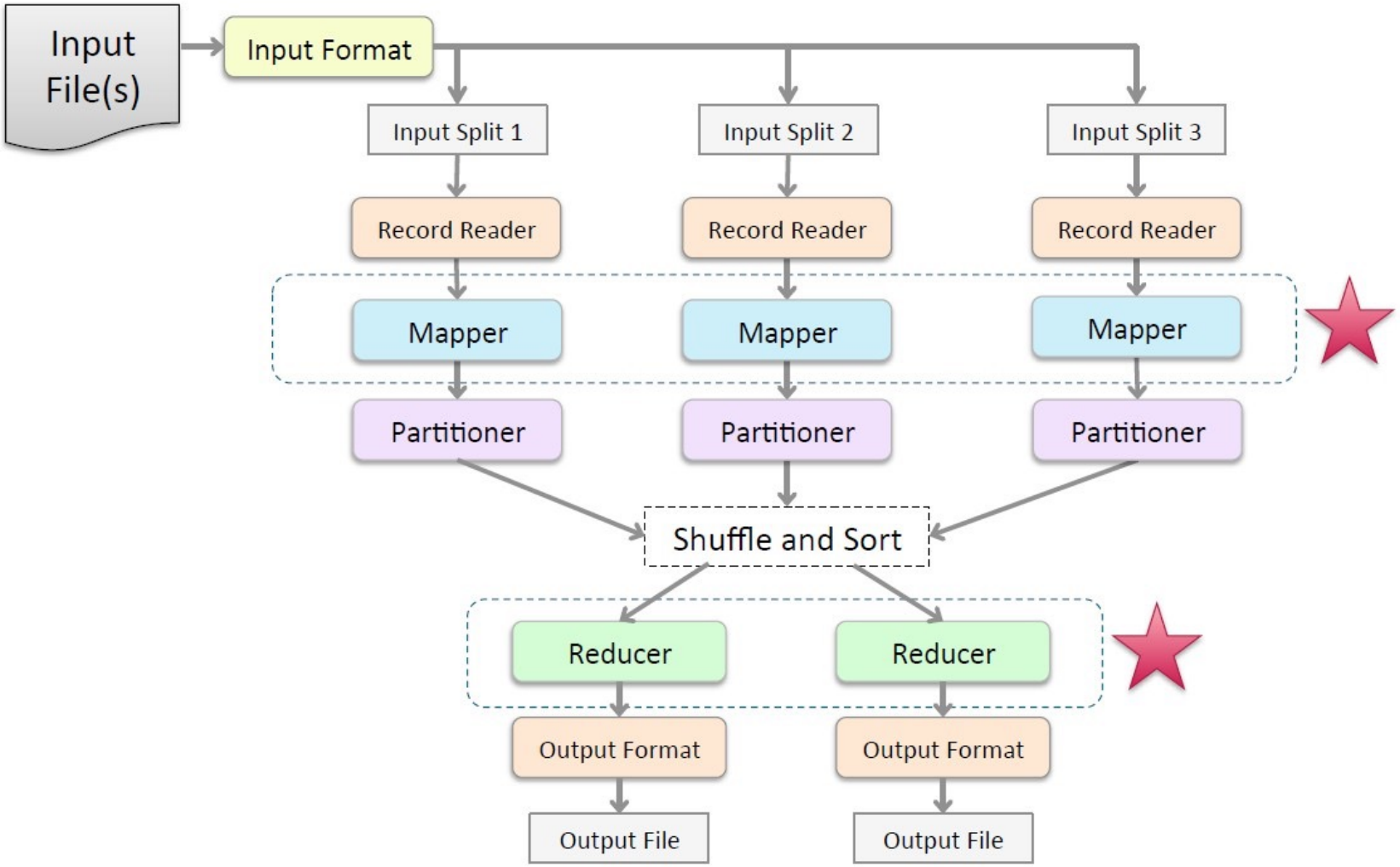
2. Apache Hadoop Core (part 2 – Map Reduce)

MapReduce: Implementación

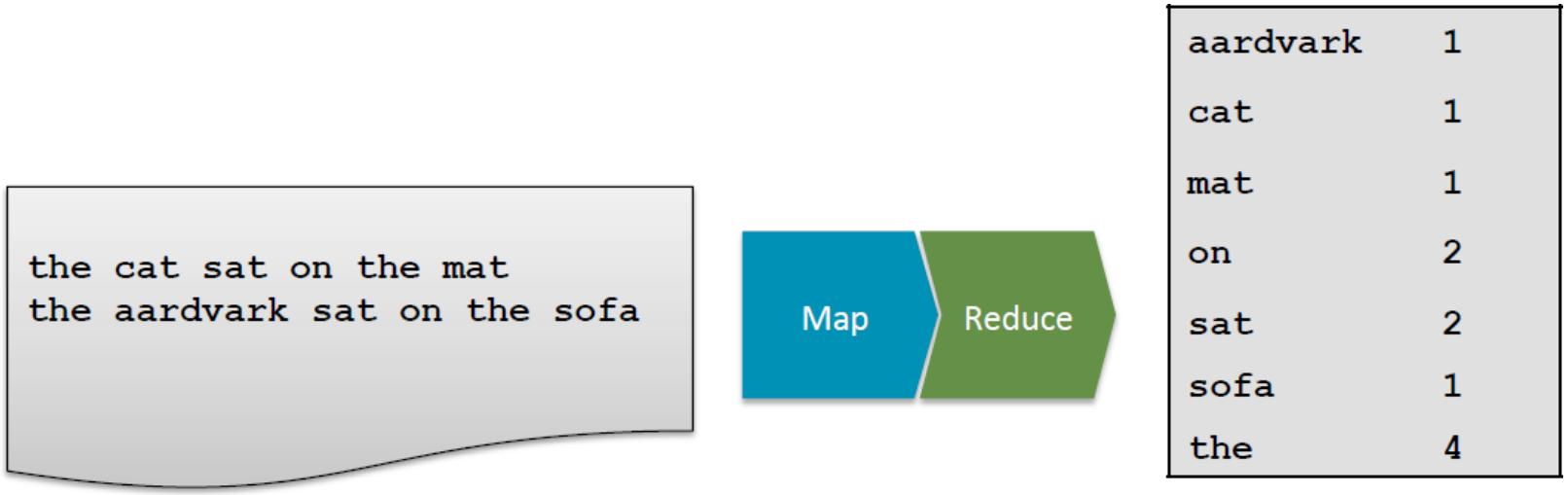
Recordatorio



MapReduce: Implementación

Ejemplo: WordCount

Vamos a repasar la API de MapReduce mediante el ejemplo WordCount, que servirá de hilo conductor para nuestra explicación



MapReduce: Implementación

Recordatorio

- Puntos esenciales
  - Mapper
  - Reducer
  - Driver: código que se ejecuta en el cliente y que envía el Job al clúster

- Antes de ver el código, necesitamos repasar algunos conceptos básicos de la API

MapReduce: Implementación

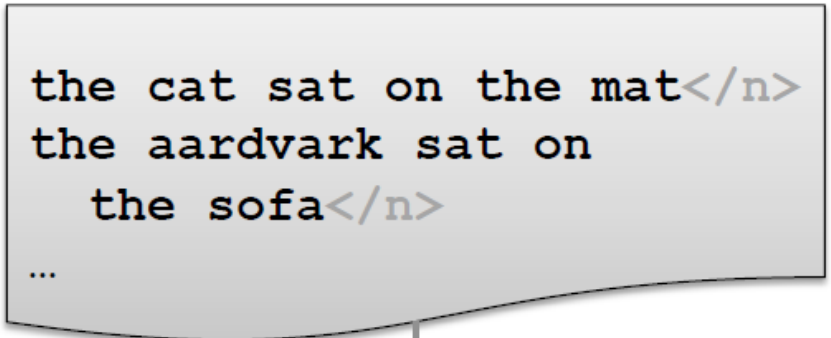
MapReduce API

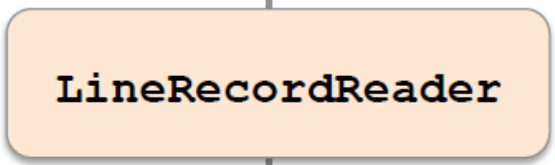
- Los datos de entrada del Mapper son especificados por un **InputFormat**
  - Especificado en el Driver
  - Define la localización de los datos de entrada
    - Normalmente un archivo o directorio
  - Determina como dividir los datos de entrada en input splits
    - Cada Mapper trabaja sobre un input Split
  - Crea un objeto RecordReader
    - RecordReader parsea los datos de entrada en valores del tipo clave valor para pasárselos al Mapper

MapReduce: Implementación

Ejemplo: TextInputFormat

- TextInputFormat es el formato por defecto si no se especifica lo contrario en el Driver
- Crea objetos LineRecordReaders
- Cada \n del archivo es considerado el value
- La Key es el byte de desplazamiento de la línea dentro del archivo





key	value
0	the cat sat on the mat
23	the aardvark sat on the sofa
52	…
…	…

MapReduce: Implementación

Otros formatos comunes de entrada

- **FileInputFormat**
  - Clase abstracta utilizada por todos los InputFormat basados en archivos
- **KeyValueTextInputFormat**
  - Mapera todas las linias con terminación \n en el formato : 'key[separator]value'
    - Por defecto, el separador es un tabulador
- **SequenceFileInputFormat**
  - Archivo binario de datos del tipo key/value con metainformación adicional
- **SequenceFileAsTextInputFormat**
  - Parecido al anterior pero mapeado del estilo (key.toString(), value.toString())

MapReduce: Implementación

Keys y Values

- Keys y Values en Hadoop son objetos Java
  - No primitivos
- Values son objetos que implementan **Writable**
- Keys son objetos que implementan **WritableComparable**

MapReduce: Implementación

Que es Writable?

- Es una interface que hace la serialización rápida y sencilla para Hadoop
- Todos los Values deben implementar la interface Writable
- Hadoop define sus propias 'box classes' para Strings, integers… etc
  - IntWritable para ints
  - LongWritable para longs
  - FloatWritable para floats
  - DoubleWritable para doubles
  - Text para Strings
  - Etc

MapReduce: Implementación

Que es WritableComparable?

- Un objeto que implementa WritableComparable es Writable y Comparable a la vez
  - Dos objetos WritableComparable pueden ser comparados entre ellos atendiendo a unas reglas de orden determinadas
  - Las Keys deben ser todas WritableComparable porque pasan a los reducers de manera ordenada
- A pesar de su nombre, todas las 'box classes' de Hadoop implementan a la vez Writable y WritableComparable.
  - Por ejemplo IntWritable es también WritableComparable

MapReduce: Implementación

Driver, puntos claves

- El código del Driver se ejecuta en la máquina del cliente
- El Driver tiene como objetivo configurar y enviar el Job al clúster

MapReduce: Implementación

Código del Driver (1)

```
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;

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);
```

MapReduce: Implementación

Código del Driver (2)

```
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

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

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

MapReduce: Implementación

Driver, creación del objeto Job

- La clase Job permite configurar las opciones del MapReduce
  - Las clases que serán usadas para Map y Reduce
  - Directorios de entrada y salida
  - Muchas otras opciones

- Las opciones no introducidas explícitamente en el código del driver se leen de los ficheros de configuración de Hadoop
  - /etc/hadoop/conf

- Cualquier otra opción no especificada en el fichero de configuración usa los valores por defecto de Hadoop

MapReduce: Implementación

Driver, configuración del objeto Job. Formato de entrada

- El InputFormat por defecto (TextInputFormat) es utilizado si no se especifica otro

- Para utilizar otro InputFormat por defecto:

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

MapReduce: Implementación

Driver, configuración del objeto Job. Determinar archivos de entrada (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]));

    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);
    System.exit(success ? 0 : 1);
  }
}
```

MapReduce: Implementación

Driver, configuración del objeto Job. Determinar archivos de entrada (2)

- Por defecto, FileInputFormat.setInputPaths() leera todos los archivos especificados en el directorio y los enviará a los Mappers
  - Excepción: todos aquellos archivos que empiecen por . o _ serán ignorados
- Una alternativa es usar FileInputFormat.addInputPath(), que puede ser llamado múltiples veces para especificar un solo archivo
- Un filtro de archivos mas avanzado se puede obtener utilizando la interficie PathFilter
  - Contiene el método accept() que puede devolver true o false dependiendo de si el archivo debe ser procesado o no según un criterio establecido

MapReduce: Implementación

Driver, configuración del objeto Job. Determinar archivos de salida (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]));

    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);
    System.exit(success ? 0 : 1);
  }
}
```

MapReduce: Implementación

Driver, configuración del objeto Job. Determinar archivos de salida (2)

- FileOutputFormat.setOutputPath() especifica el directorio donde los Reducers escribirán la salida
  - En el Driver se puede especificar el formato de salida
  - Por defecto, es texto plano

- También se puede especificar explícitamente como

```
job.setOutputFormatClass(TextOutputFormat.class)
```

MapReduce: Implementación

Driver, configuración del objeto Job. Mappers y Reducers (1)

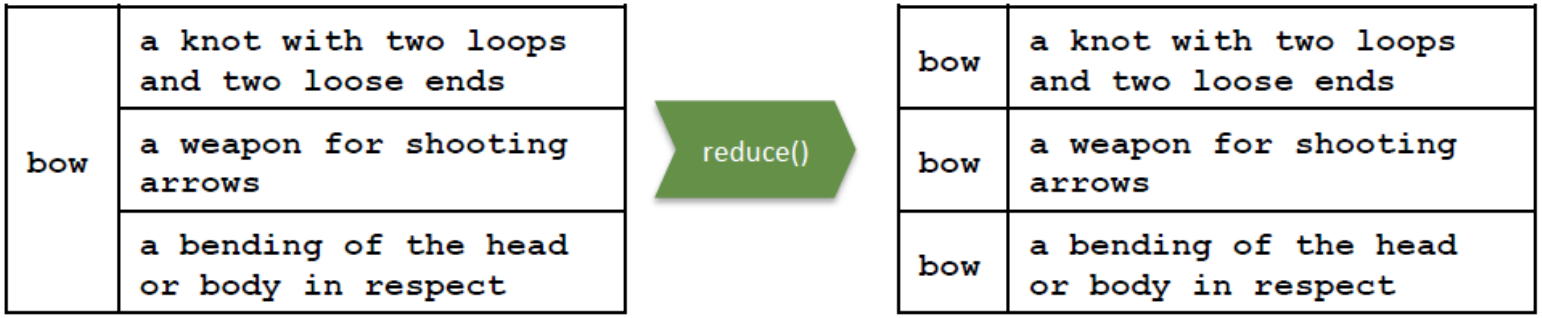
```
public class WordCount {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.printf("Usage: WordCount <input dir> <output dir>\n");
      System.exit(-1);
    }
    Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word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

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

    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);
    System.exit(success ? 0 : 1);
  }
}
```

MapReduce: Implementación

Driver, configuración del objeto Job. Mappers y Reducers (2)

- Asignar las clases de los Mapper y Reducer es opcional
- Si no se especifican en el Driver, Hadoop usa los Mappers y Reducers por defecto
  - IdentityMapper

  - IdentityReducer



MapReduce: Implementación

Driver, configuración del objeto Job. Tipo de salida de Mappers y Reducers

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

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

MapReduce: Implementación

Driver, configuración del objeto Job. Ejecutando el 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]));

    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);
    System.exit(success ? 0 : 1);
  }
}
```

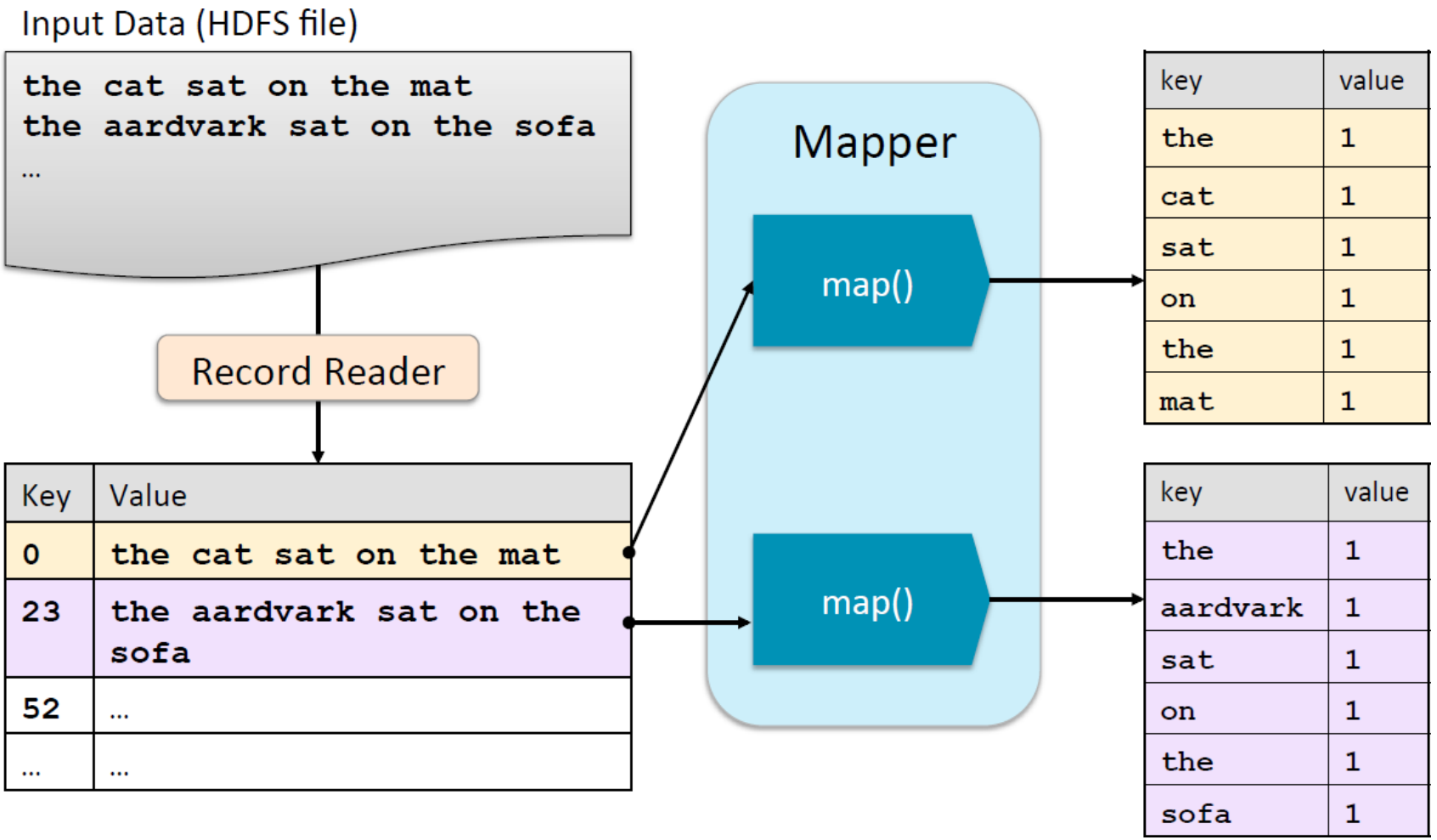
MapReduce: Implementación

Driver, configuración del objeto Job. Ejecutando el Job (2)

- Hay dos maneras de ejecutar un MapReduce Job
  - job.waitForCompletion() – Bloqueante, espera hasta que termina el Job
  - job.submit() – No es bloqueante, el código del driver continua mientras el Job se ejecuta

MapReduce: Implementación

Mapper, recordatorio Word Count



MapReduce: Implementación

Mapper

```
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));
      }
    }
  }
}
```

MapReduce: Implementación

Mapper, Declaración de clase (1)

```
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));
      }
    }
  }
}
```

MapReduce: Implementación

Mapper, Declaración de clase (2)

- Las clases Mapper deben extender la clase base Mapper

- Especificar los Generic Types, los dos primeros parámetros corresponden a los tipos de entrada, y los dos últimos corresponden a la salida de datos intermedios
  - Las Keys deben ser WritableComparable
  - Los Values deben ser Writable

MapReduce: Implementación

Mapper, El método map (1)

```
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));
      }
    }
  }
}
```

MapReduce: Implementación

Mapper, El método map (2)

- El método map() recibe como argumentos : Key, Value, Context
- Context se usa para escribir el conjunto de datos intermedio
- Context también contiene información relativa a la configuración del Job

MapReduce: Implementación

Mapper, Procesando la línea (1)

```
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));
      }
    }
  }
}
```

MapReduce: Implementación

Mapper, Procesando la línea (2)

- El value es un objeto de tipo Text
- Recogemos el String que contiene este objeto

MapReduce: Implementación

Mapper, Procesando la línea (3)

```
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));
      }
    }
  }
}
```

MapReduce: Implementación

Mapper, Procesando la línea (4)

- Se divide el String en palabras utilizando una expresión regular
- Luego se itera sobre el resultado de esta división

MapReduce: Implementación

Mapper, Salida de datos intermedios (1)

```
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));
      }
    }
  }
}
```

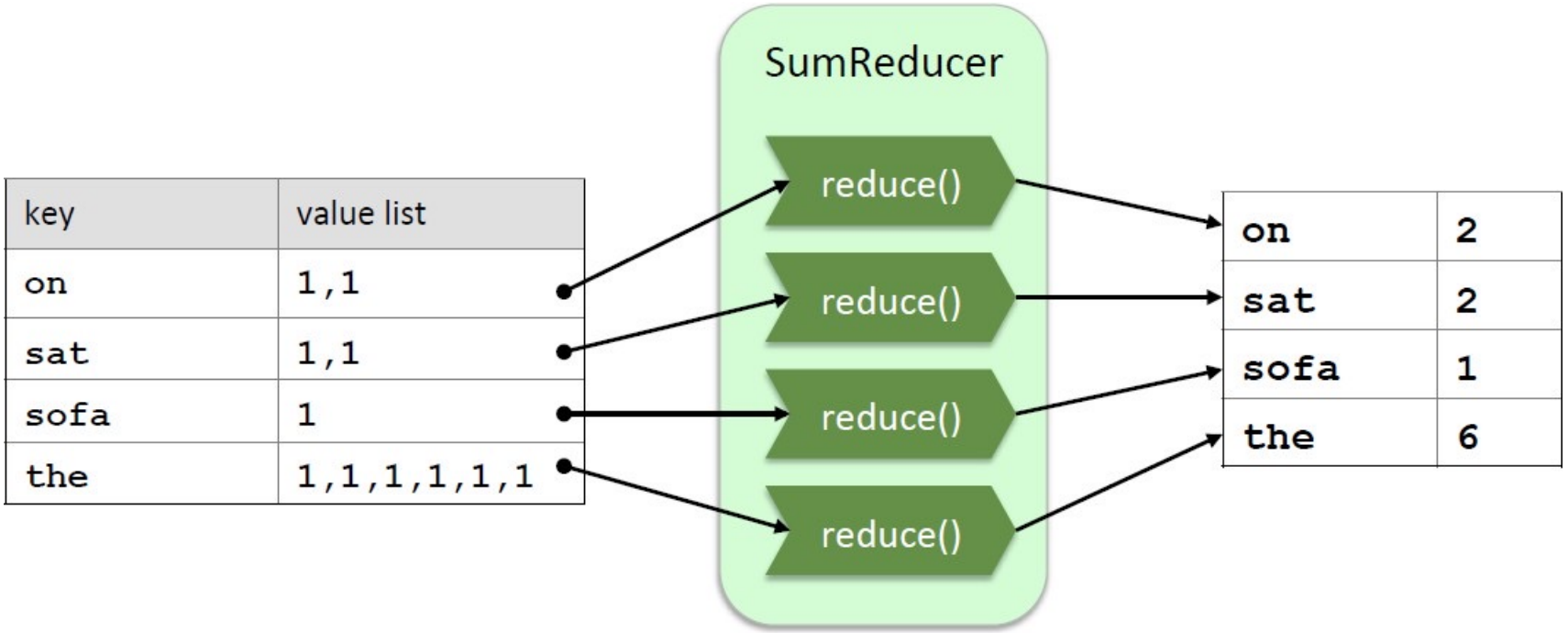
MapReduce: Implementación

Mapper, Salida de datos intermedios (2)

- Para emitir una pareja key/value como salida, llamamos al método write del objeto Context
- La key es la misma palabra y el value es el número 1
- La key de salida debe ser WritableComparable y el value debe ser Writable

MapReduce: Implementación

Reducer, recordatorio Word Count



key	value list
on	1,1
sat	1,1
sofa	1
the	1,1,1,1,1,1

on	2
sat	2
sofa	1
the	6

MapReduce: Implementación

Reducer

```
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));
  }
}
```

MapReduce: Implementación

Reducer, Declaración de clase (1)

```
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));
  }
}
```

MapReduce: Implementación

Reducer, Declaración de clase (2)

- Las clases Reducers deben extender la clase base Reducer

- Especificar los Generic Types, los dos primeros parámetros corresponden a los tipos de entrada (datos intermedio), y los dos últimos corresponden a la salida de datos final
  - Las Keys deben ser WritableComparable
  - Los Values deben ser Writable

MapReduce: Implementación

Reducer, Método reduce (1)

```
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));
  }
}
```

MapReduce: Implementación

Reducer, Método reduce (2)

- El método reduce recibe una key y una colección Iterable de objetos
  - Que son los values emitidos por el mapper para una key

- También recibe un objeto de tipo Context

MapReduce: Implementación

Reducer, Procesando los valores (1)

```
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));
  }
}
```

MapReduce: Implementación

Reducer, Procesando los valores (2)

- Con Iterable, recorreremos todos los objetos de la colección
- En cada uno de ellos extraemos el valor número con value.get()
- Iremos añadiendo los valores

MapReduce: Implementación

Reducer, Escribiendo la salida (1)

```
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));
  }
}
```

MapReduce: Implementación

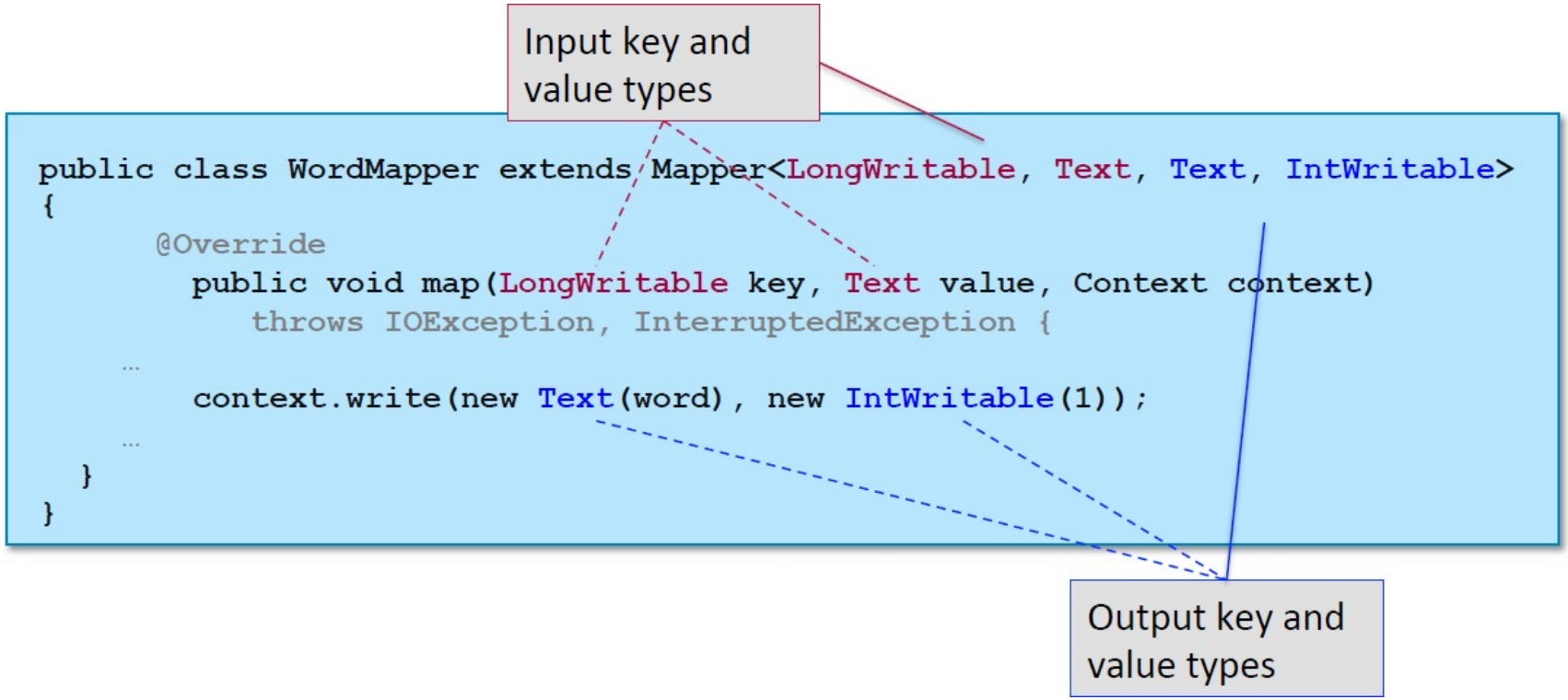
Reducer, Escribiendo la salida (2)

- Para finalizar escribiremos la salida en HDFS con el método write() del objeto Context

MapReduce: Implementación

Consideraciones. Tipos de variables coincidentes (1)

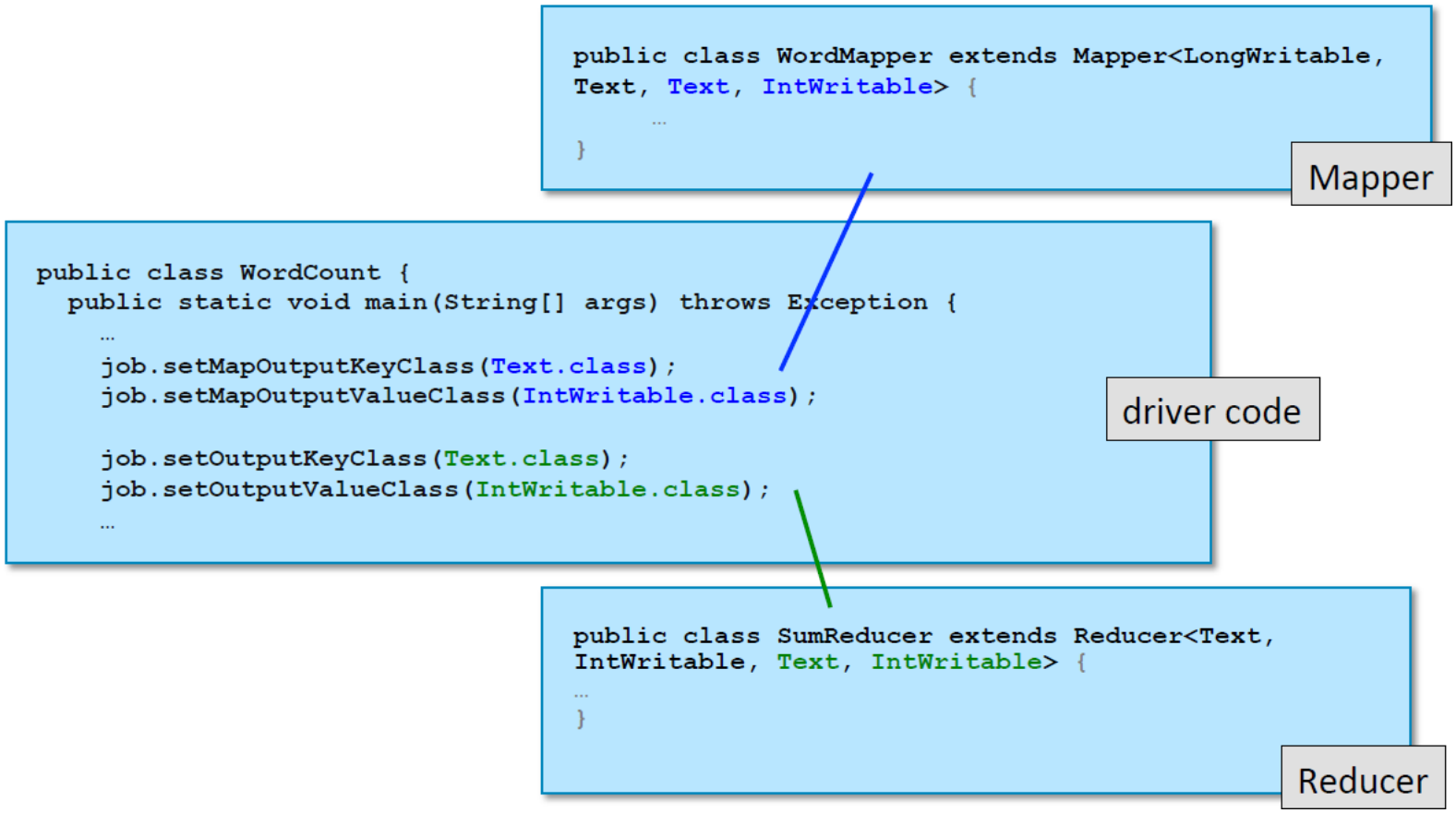
- Importante: Mappers y reducers deben declarar los mismos tipos que los que se usan en las clases



MapReduce: Implementación

Consideraciones. Tipos de variables coincidentes (2)

- Las variables de salida deben coincidir con las declaradas en el driver



4– Hands On:

Desarrollo MapReduce

MapReduce: Implementación

Old API vs New API

New API

```
import org.apache.hadoop.mapreduce.*
```

Driver code:

```
Configuration conf = new Configuration();
Job job = new Job(conf);
job.setJarByClass(Driver.class);
job.setSomeProperty(...);
...
job.waitForCompletion(true);
```

Mapper:

```
public class MyMapper extends Mapper {

  public void map(Keytype k, Valuetype v,
                                  Context c) {
    ...
    c.write(key, val);
  }
}
```

Old API

```
import org.apache.hadoop.mapred.*
```

Driver code:

```
JobConf conf = new JobConf(Driver.class);
conf.setSomeProperty(...);
...
JobClient.runJob(conf);
```

Mapper:

```
public class MyMapper extends MapReduceBase
                            implements Mapper {

  public void map(Keytype k, Valuetype v,
           OutputCollector o, Reporter r) {
    ...
    o.collect(key, val);
  }
}
```

MapReduce: Implementación

Old API vs New API

New API	Old API

Reducer:
`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);`
`  }`
`  }`
`}`
	Reducer:
	`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);`
	`  }`
	`  }`
	`}`

`setup(Context c)` **(See later)**	`configure(JobConf job)`
`cleanup(Context c)` **(See later)**	`close()`

MapReduce: Implementación

Old API vs New API

- El código generado con la Old API y la New API son compatibles con MRv1 y MRv2

	Old API	New API
MapReduce v1	✔	✔
MapReduce v2	✔	✔

MapReduce: Implementación

Hadoop Streaming API

- Permite la ejecución de Jobs Map Reduce escritos con otros lenguajes diferentes a Java

- Muchas organizaciones tienen experiencia en el desarrollo con otros lenguajes de programación que no son Java
  - Ruby
  - Python
  - Perl
  - …

- Streaming API permite a los desarrolladores escribir Map Reduce con otros lenguajes

- El único requerimiento es que el lenguaje escogido permita lecturas y escrituras por una salida estándar

MapReduce: Implementación

Hadoop Streaming API: Ventajas y Desventajas

- **Ventajas**
  - No es necesario aprender Java
  - Desarrollo mas rápido
  - Posibilidad de utilizar librerías propias de cada lenguaje

- **Desventajas**
  - Rendimiento
  - Solo se pueden tratar Texto
  - Los Jobs de Streaming utilizan mucha RAM o levantan un número de procesos muy grande
  - Otros elementos própios de Map Reduce solo pueden ser escritos en Java : Partitienoers, InputFormats … etc

MapReduce: Implementación

Hadoop Streaming API.

Ejemplo Mapper

```
#!/usr/bin/env perl
while (<>) {                    # Read lines from stdin
  chomp;                        # Get rid of the trailing newline
  (@words) = split /\W+/;       # Create an array of words
  foreach $w (@words) {         # Loop through the array
    print "$w\t1\n";            # Print out the key and value
  }
}
```

MapReduce: Implementación

Hadoop Streaming API.

Ejemplo Reducer

```
#!/usr/bin/env perl
$sum = 0;
$last = "";
while (<>) {                          # read lines from stdin
  ($key, $value) = split /\t/;        # obtain the key and value
  $last = $key if $last eq "";        # first time through
  if ($last ne $key) {                # has key has changed?
    print "$last\t$sum\n";            # if so output last key/value
    $last = $key;                     # start with the new key
    $sum = 0;                         # reset sum for the new key
  }
  $sum += $value;                     # add value to tally sum for key
}
print "$key\t$sum\n";                 # print the final pair
```

MapReduce: Implementación

Hadoop Streaming API.

Ejemplo comando ejecución del Job

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/\
   streaming/hadoop-streaming*.jar \
   -input myInputDirs \
   -output myOutputDir \
   -mapper myMapScript.pl \
   -reducer myReduceScript.pl \
   -file mycode/myMapScript.pl \
   -file mycode/myReduceScript.pl
```

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

- ToolRunner se puede usar en las clases Driver
  - No es un requisito, per si una best practice

- ToolRunner usa la clase GenericOptionsParser internamente
  - Permite especificar opciones de configuración en la línea de comandos
  - También permite especificar objetos para la DistributedCache en la línea de comandos

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

```
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;
  }
}
```

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

La clase Driver implementa la interface Tool y extiende de la clase Configured

```
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;
  }
}
```

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

El método main del Driver llama a ToolRunner.run

```
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;
  }
}
```

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

El método run crea, configura y envía el Job

```
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;
  }
}
```

MapReduce: Implementación

Hadoop API en profundidad. ToolRunner

- ToolRunner permite especificar opciones en la línea de comandos
- Normalmente se usa para especificar propiedades a Hadoop usando el flag -D
  - Sobreescribira qualquier parámetro por defecto en la configuración
  - Pero NO sobreescribirá los parámetros seteados en el Driver vía código
- El parámetro –D debe aparecer antes de cualquier parámetro propio del programa
- Se puede especificar un fichero XML de configuración con el parámetro –conf

MapReduce: Implementación

Hadoop API en profundidad. Métodos setup y cleanup

- En las fases de Mapper o Reducer es común querer ejecutar código que se ejecute antes de la primera llamada al método map o reduce
  - Inicialización de estructura de datos
  - Leer datos de un archivo externo
  - Setear parámetros

- El método setup se ejecuta antes de la primer llamada al método map o reduce

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

MapReduce: Implementación

Hadoop API en profundidad. Métodos setup y cleanup

- Parecido al método setup, el método cleanup ejecuta código después de que todos los registros hayan sido procesados por el Mapper o el Reducer

- El método cleanup se ejecuta justo antes de que el Mapper o el Reducer acabe

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

MapReduce: Implementación

Hadoop API en profundidad. Métodos setup y cleanup

```
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...
}
```

Hadoop API en profundidad. Combiners

- Normalmente, los Mappers producen una cantidad de datos intermedios grande
  - Los datos han de ser pasados a los Reducers
  - Esto pueden producir mucho trafico de red

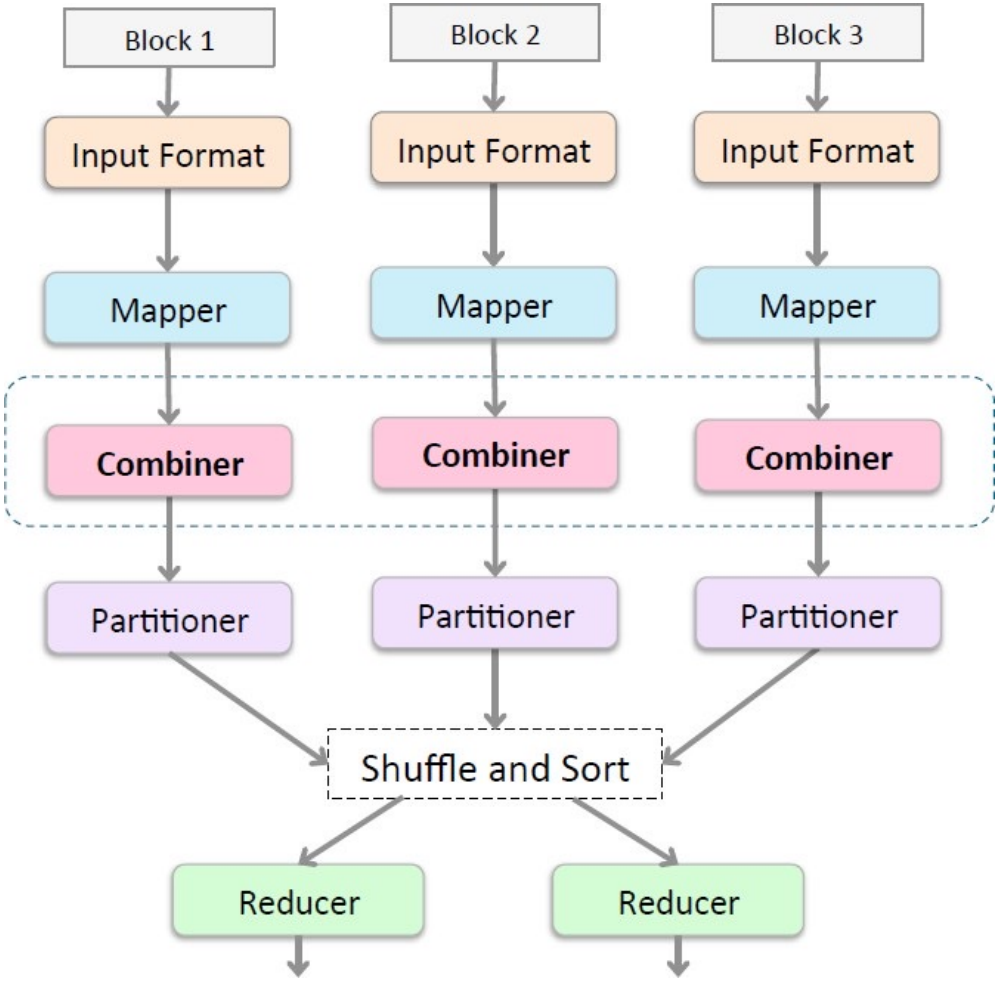
- Reducir la cantidad de datos es posible especificando un Combiner
  - Es como un “mini-Reducer”
  - Se ejecuta de manera local sobre una salida concreta del Mapper
  - La salida de los Combiners es enviada a los Reducers

- Combiners y Reducers suelen ser idénticos, es posible si:
  - La operación que ejecuta el Reducer es conmutativa y asociativa
  - Los datos de entrada y salida deben ser idénticos

MapReduce: Implementación

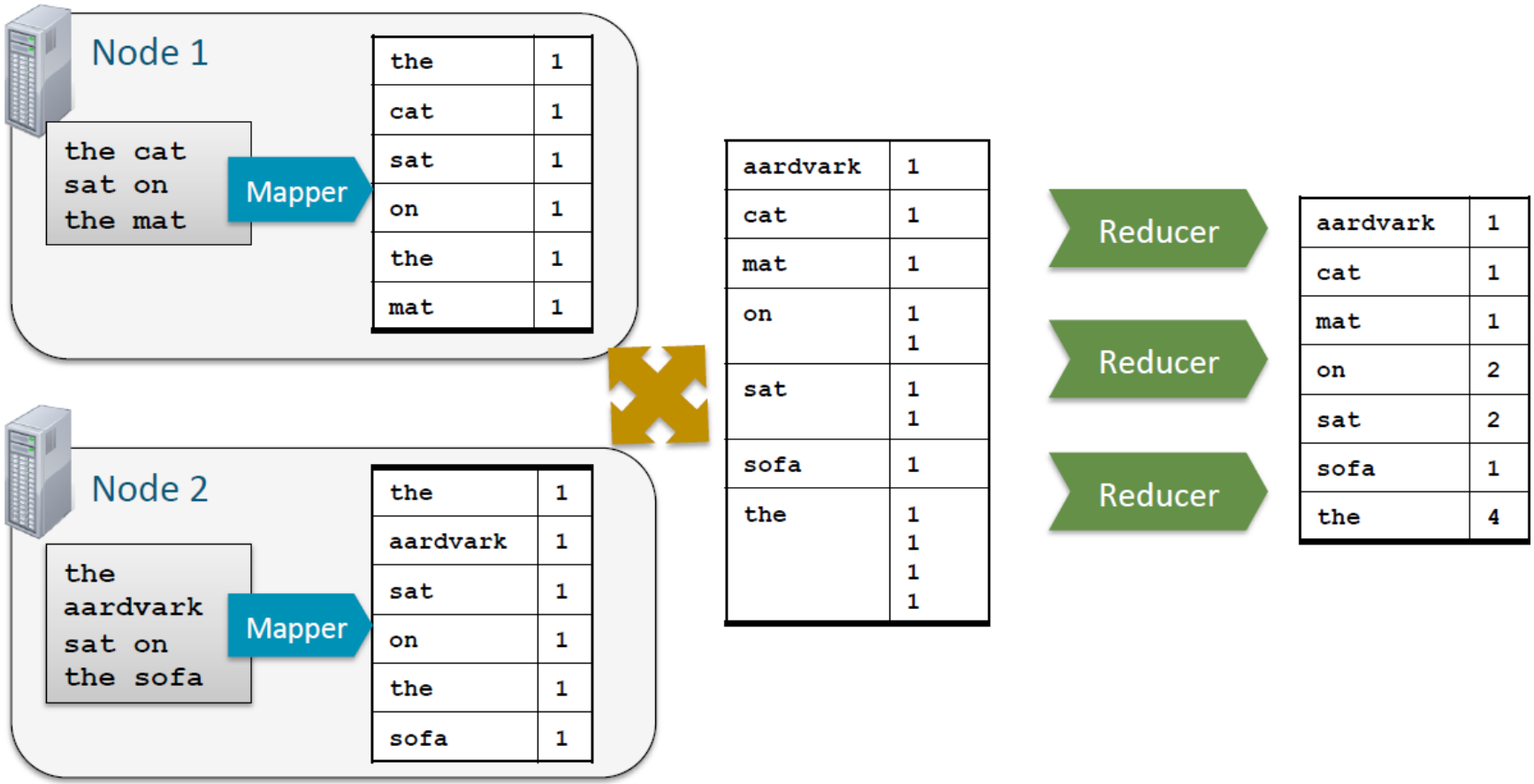
Hadoop API en profundidad. Combiners

- Los Combiners se ejecutan como parte de la fase de Map
- La salida de los Combiners se pasa a los Reducers



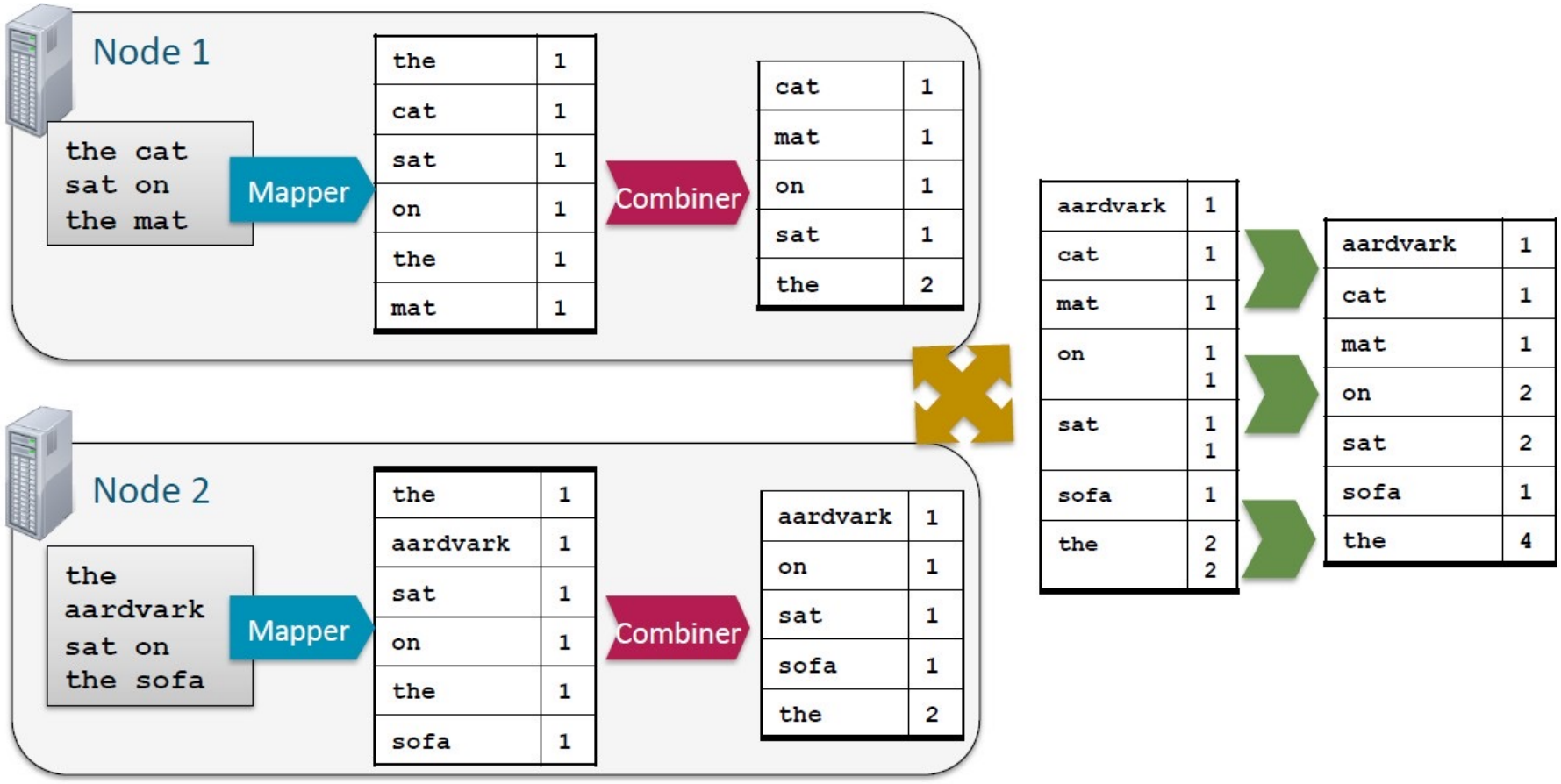
MapReduce: Implementación

Hadoop API en profundidad. Combiners



MapReduce: Implementación

Hadoop API en profundidad. Combiners



MapReduce: Implementación

Hadoop API en profundidad. Combiners

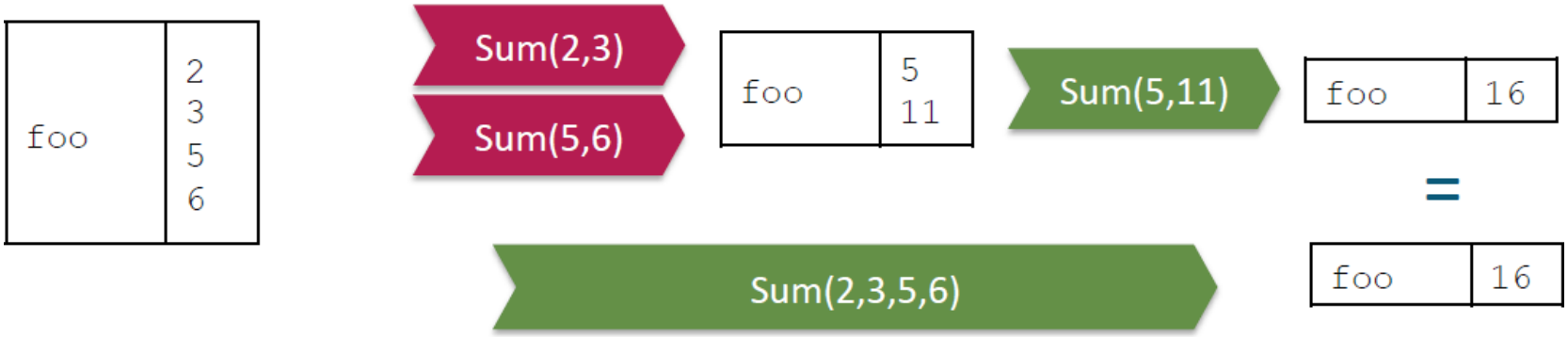
- Los Combiners utilizan la misma firma que los Reducers
  - La entrada es una Key con una lista de Values
  - La salida pueden ser 0 o mas parejas Key/Value

```
reduce(inter_key, [v1, v2, …]) →
                        (result_key, result_value)
```

MapReduce: Implementación

Hadoop API en profundidad. Combiners

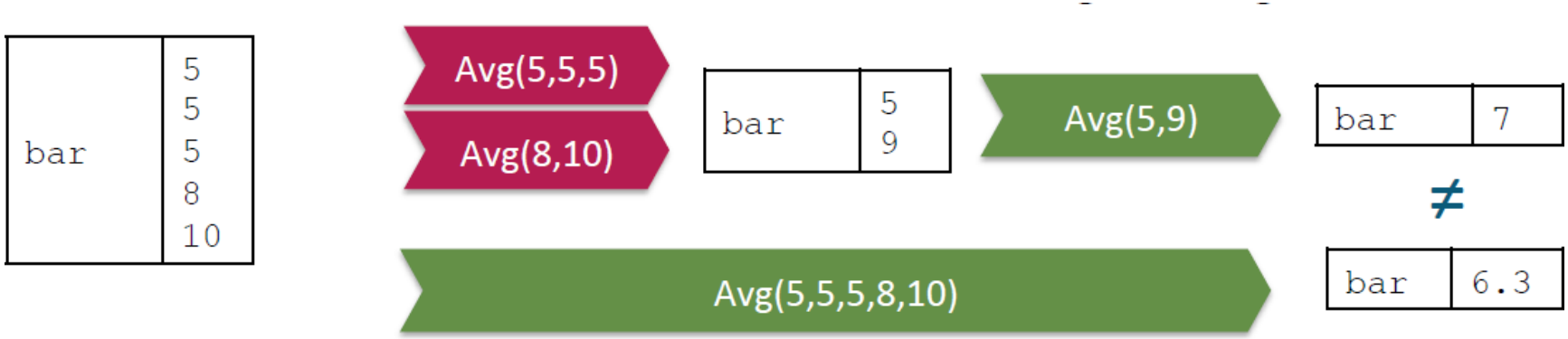
- Algunos Reducers pueden utilizarse como Combiners
  - Si la operación es asociativa y conmutativa. Ex. SumReducer



MapReduce: Implementación

Hadoop API en profundidad. Combiners

- Algunos Reducers NO pueden utilizarse como Combiners
  - Si la operación NO es asociativa y conmutativa. Ex. SumReducer



MapReduce: Implementación

Hadoop API en profundidad. Combiners

- Para especificar en el Driver la clase que hará de Combiner
  - Con el método setCombinerClass

```
job.setMapperClass(WordMapper.class);
job.setReducerClass(SumReducer.class);
job.setCombinerClass(SumReducer.class);
```

- Las salidas y las entradas del Combinar deben ser del mismo tipo que las del Reducer

- Muy Importante: El Combiner puede ejecutarse una vez o mas de una para una misma salida de un Mapper concreto
  - No poner código en el Combiner que pueda influir en el resultado final si se ejecuta mas de una vez

MapReduce: Implementación

Hadoop API en profundidad. Acceso a HDFS

- Adicionalmente a la línea de comandos de Hadoop, se puede acceder a HDFS por programación
  - Útil si se debe realizar un tratamiento de los archivos de entrada o salida
  - Útil también si otros programas fuera de Hadoop deben acceder a los resultados de los Job MapReduce

- Cuidado, HDFS no es como un archivo normal
  - Los archivos no pueden ser modificados una vez escritos

- Hadoop proporciona una clase abstracta FileSystem con el objetivo de interactuar con los archivos

- La API proporciona acceso a un FileSystem genérico
  - Pueden ser HDFS, FileSystem local, S3…

MapReduce: Implementación

Hadoop API en profundidad. Acceso a HDFS

- Para utilizar la api FileSystem, se crea una instancia de ella

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

- El objeto conf contiene información de la configuración de Hadoop, y por tanto, conoce la dirección del NameNode

- Un archivo en HDFS se representa con el objeto Path

```
Path p = new Path("/path/to/my/file");
```

MapReduce: Implementación

Hadoop API en profundidad. Acceso a HDFS

Algunos métodos útiles de la API….

- `FSDataOutputStream create(...)`
  - Extends `java.io.DataOutputStream`
  - Provides methods for writing primitives, raw bytes etc
- `FSDataInputStream open(...)`
  - Extends `java.io.DataInputStream`
  - Provides methods for reading primitives, raw bytes etc
- `boolean delete(...)`
- `boolean mkdirs(...)`
- `void copyFromLocalFile(...)`
- `void copyToLocalFile(...)`
- `FileStatus[] listStatus(...)`

Hadoop API en profundidad. Acceso a HDFS

Listar un directorio

```
Path p = new Path("/my/path");

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FileStatus[] fileStats = fs.listStatus(p);

for (int i = 0; i < fileStats.length; i++) {
    Path f = fileStats[i].getPath();

    // do something interesting
}
```

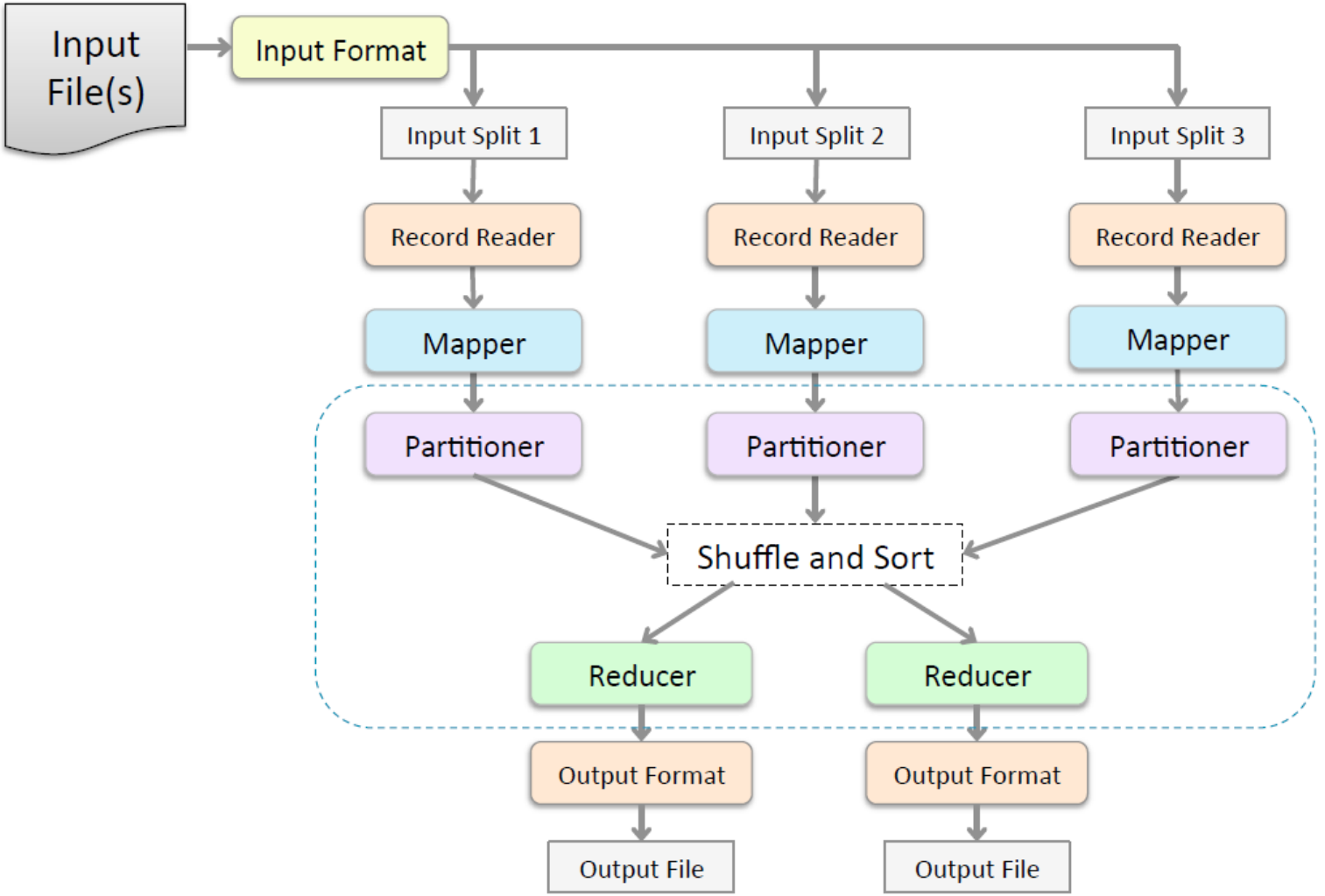
Hadoop API en profundidad. Acceso a HDFS

Escribir datos en un archivo

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

Path p = new Path("/my/path/foo");

FSDataOutputStream out = fs.create(p, false);

// write some raw bytes
out.write(getBytes());

// write an int
out.writeInt(getInt());

...

out.close();
```

MapReduce: Implementación

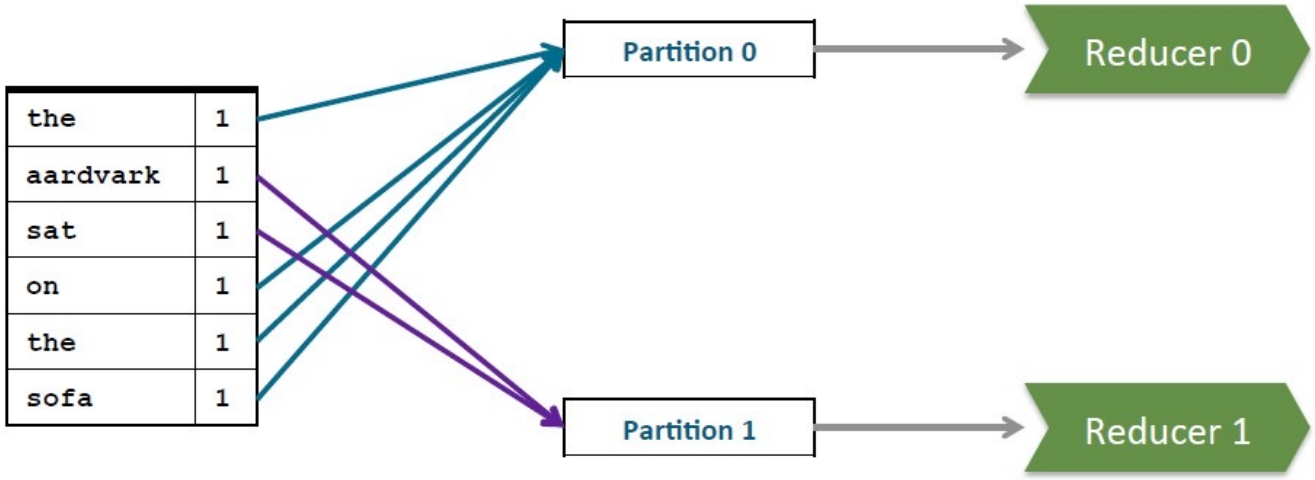
Hadoop API en profundidad. Partitoner



Hadoop API en profundidad. Partitoner

El Partitioner determina, según la key, el reducer que va a tratar el conjunto de valores asociados

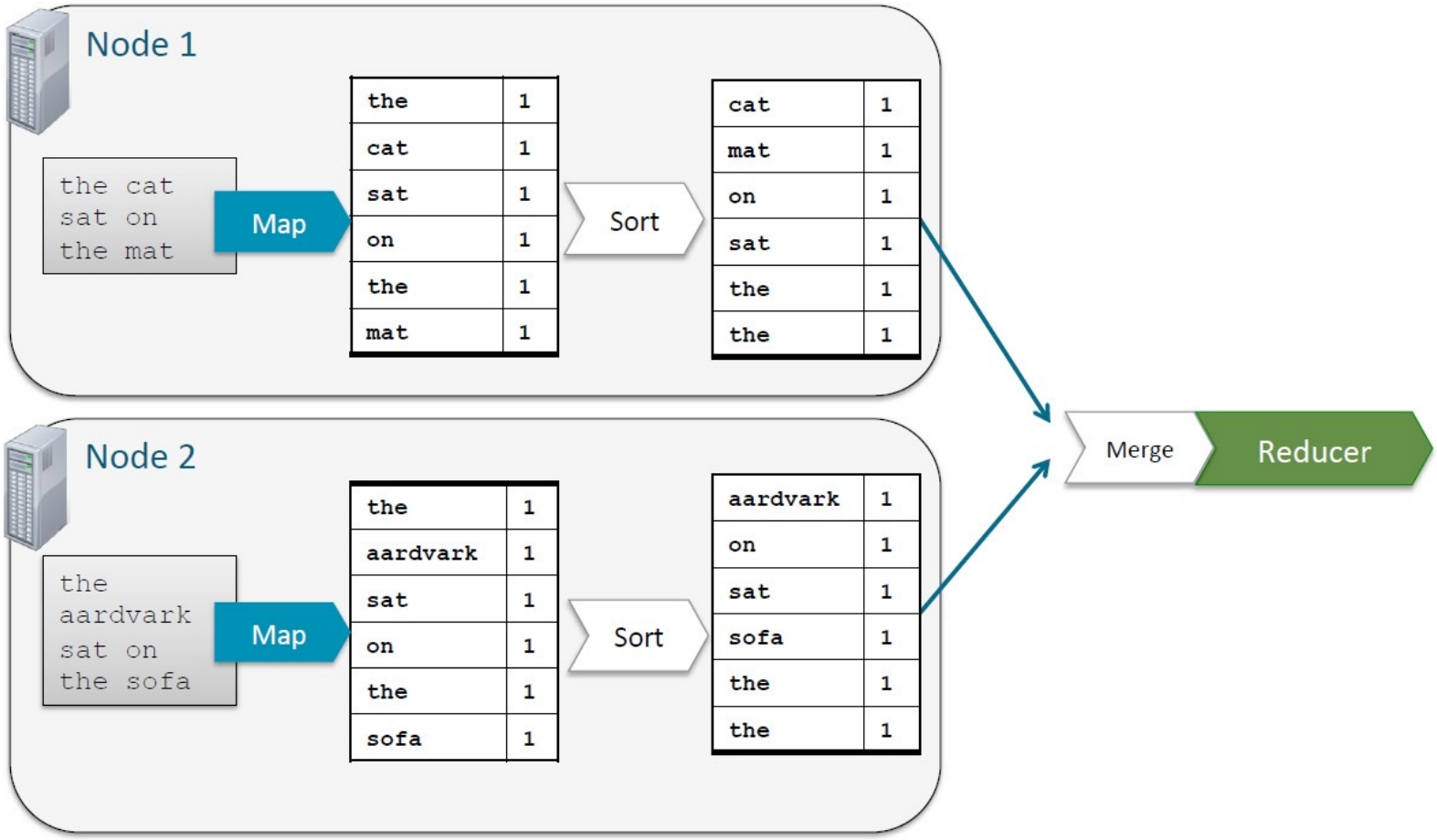
```
getPartion:
    (inter_key, inter_value, num_reducers) → partition
```



MapReduce: Implementación

Hadoop API en profundidad. Partitoner

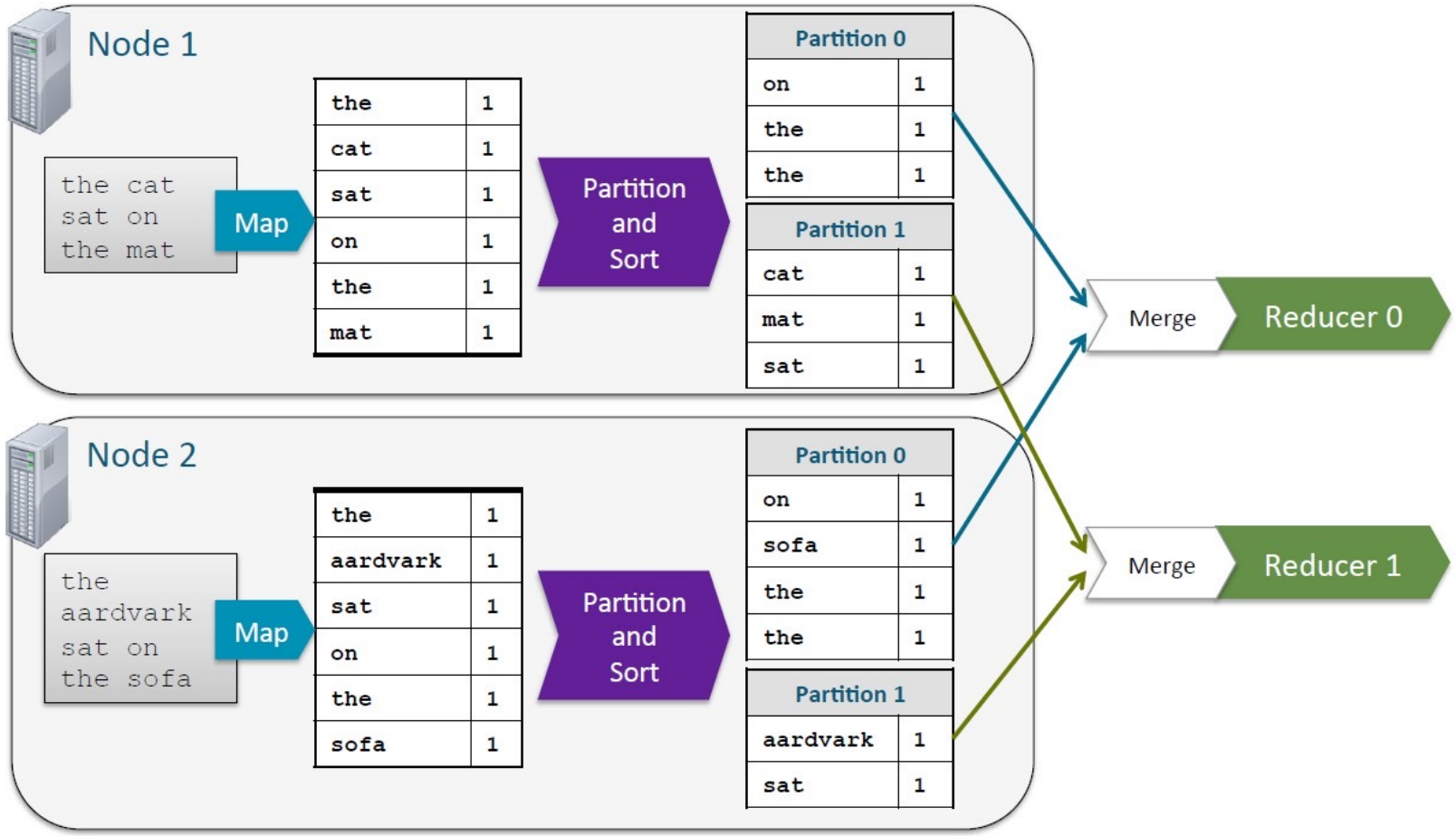
Ejemplo: WordCount con un solo Reducer



MapReduce: Implementación

Hadoop API en profundidad. Partitoner

Ejemplo: WordCount con dos Reducer



MapReduce: Implementación

Hadoop API en profundidad. Partitoner

- El Partitioner por defecto es el HashPartitioner
  - Usa el método Java HasCode
  - Garantiza que todas las parejas con la misma Key son tratadas por el mismo Reducer

```
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;
    }
}
```

Hadoop API en profundidad. Partitoner

- Un aspecto a tener en cuenta cuando se crea un Job es el número de Reducers
- Por defecto, el número de Reducers es uno
- Con un solo Reducer, una task recibe todas las Keys de manera ordenada
  - Bueno si se necesita un resultado en un orden global
  - Puede provocar problemas si tenemos un número elevado de datos
    - El nodo donde se ejecuta puede no tener suficiente espacio en el disco para almacenar los datos intermedios
    - El Reducer puede tardar mucho tiempo

MapReduce: Implementación

Hadoop API en profundidad. Partitoner

- Si el Job necesito tener todas las Keys ordenadas de forma global se debe usar un solo Reducer

- Alternativamente, se puede usar TotalOrderPartitioner
  - Si concatenamos todas las salidas de los reducers se obtendría un orden global

MapReduce: Implementación

Hadoop API en profundidad. Partitoner

- Algunos Jobs requieren usar un número concreto de Reducers

- Ejemplo: Un Job donde la salida sea un archivo por cada día de la semana
  - La Key será un día de la semana
  - Se especificarán siete Reducers
  - El Partitioner enviara las Keys al Reducer pertinente según el valor de las mismas

MapReduce: Implementación

Hadoop API en profundidad. Partitoner

Escribiendo un Partitioner

- Crear la clase y extender de Partitioner

- Sobreescribir el método getPartitioner
  - Debe retornar un número entre 0 y n-1 Reducers donde n es el número de Reducers

```
import org.apache.hadoop.mapreduce.Partitioner;

public class MyPartitioner<K,V> extends Partitioner<K,V> {

    @Override
    public int getPartition(K key, V value, int numReduceTasks) {
        //determine reducer number between 0 and numReduceTasks-1
        //...
        return reducer;
    }
}
```

MapReduce: Implementación

Hadoop API en profundidad. Partitoner

Escribiendo un Partitioner

- Para especificar el número de Reducers en el código del driver

```
job.setPartitionerClass(MyPartitioner.class);
```

5– Hands On:

Desarrollo MapReduce Avanzado