MapReduce

Es un paradigma de programación que se ejecuta en tres etapas

- Fase Map
- Fase Shuffle&Sort
- Fase Reduce

También es un método para distribuir tareas a lo largo del cluster

Automáticamente las paraleliza y distribuye

Es tolerante a fallos

Está pensado para que sea una abstracción para los programadores

- Los programas se escriben típicamente en Java, como Hadoop
- A través de *Hadoop Streaming* se puede programar en otros lenguajes de programación
- El programador solo programa, no se encarga de NADA más

Un MapReduce típico se asemeja a la concatenación del siguiente conjunto de comandos:

- cat /ficheros/ | grep '\.html' | sort | uniq –c > /salida.txt

Uniq: mostrar solo 1 ocasión a cada palabra

grep: seleccionar lo que necesitas

MapReduce

Conceptos fundamentales

- El programador solo tiene que definir
  - El Mapper
  - El Reducer
  - El Driver

Mapper

- Actúa sobre cada registro de entrada
- Cada tarea, Task, opera sobre un único bloque en HDFS, siempre que sea posible
- Cada Task opera en el nodo donde el bloque está almacenado
- La salida del Map es un par (Clave/Valor)

Suffle&Sort

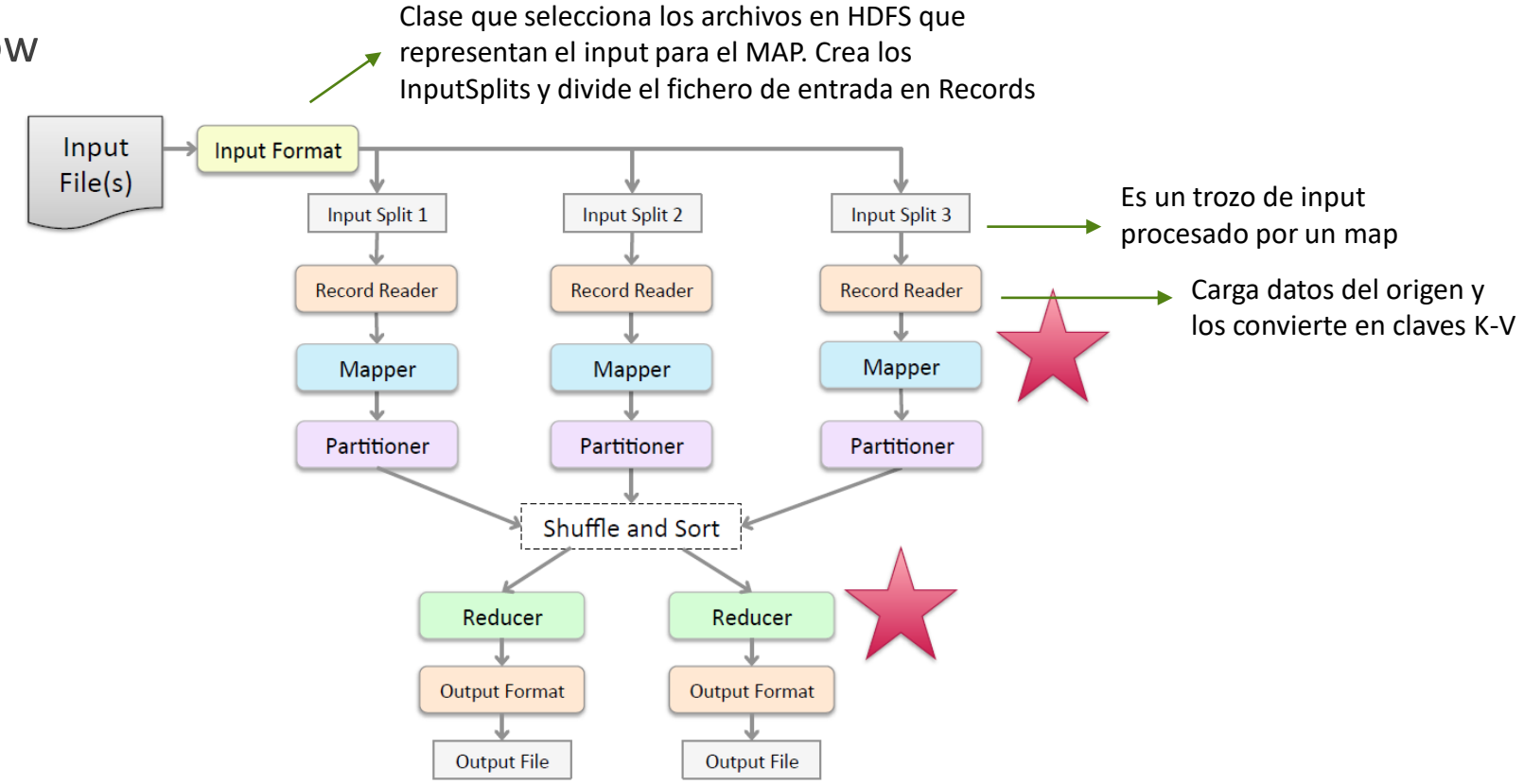
- Ordena por clave y agrupa todos los datos intermedios de todos los Mappers de una misma clave en el formato (K, [V1…Vn])
- Ocurre justo después de que todos los Mappers hayan terminado y justo antes de que la fase Reduce comience

Reducer

- Recibe la salida del S&S y realiza operaciones sobre ella
- Produce la salida final

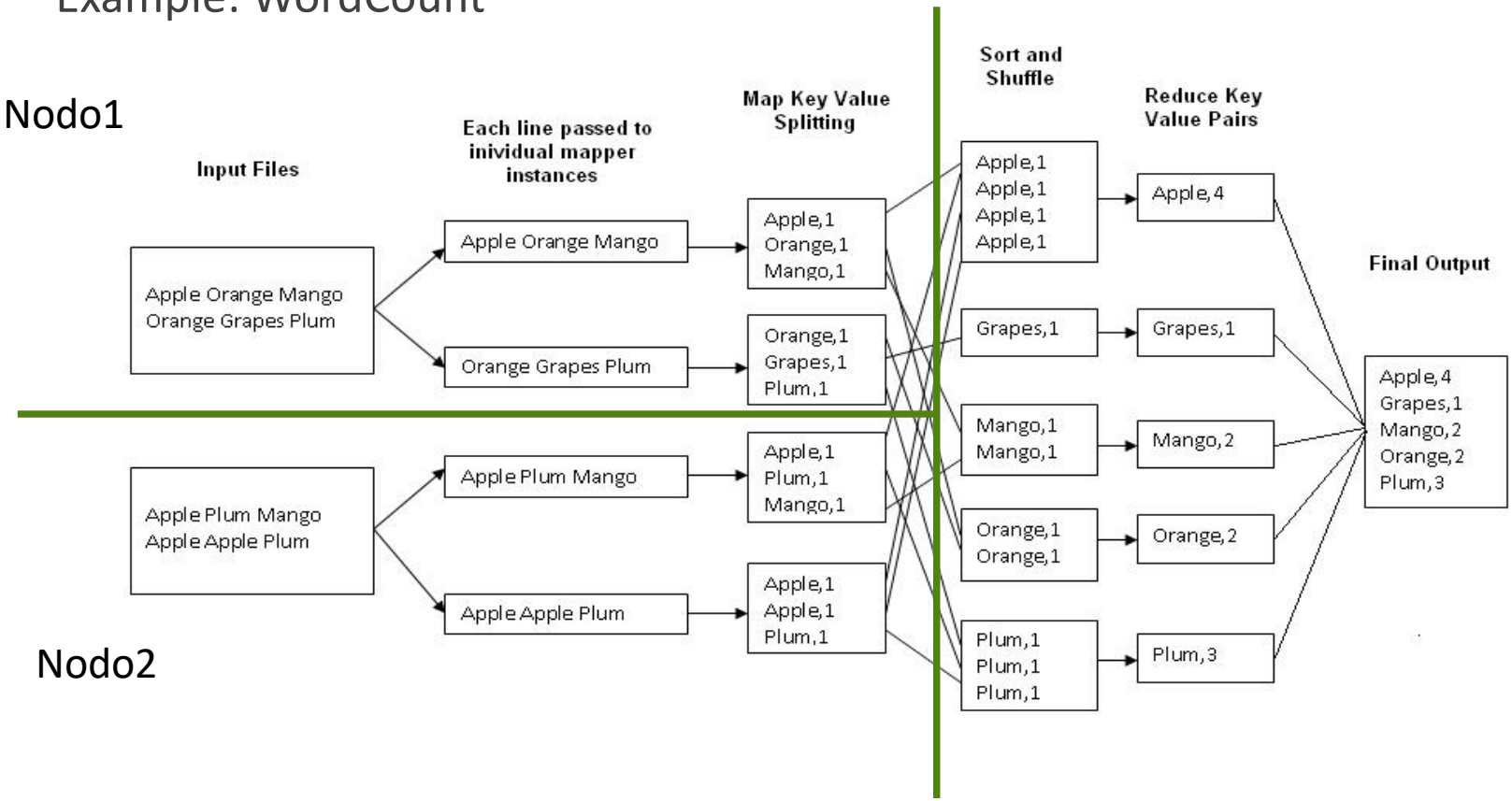
MapReduce

Workflow



MapReduce

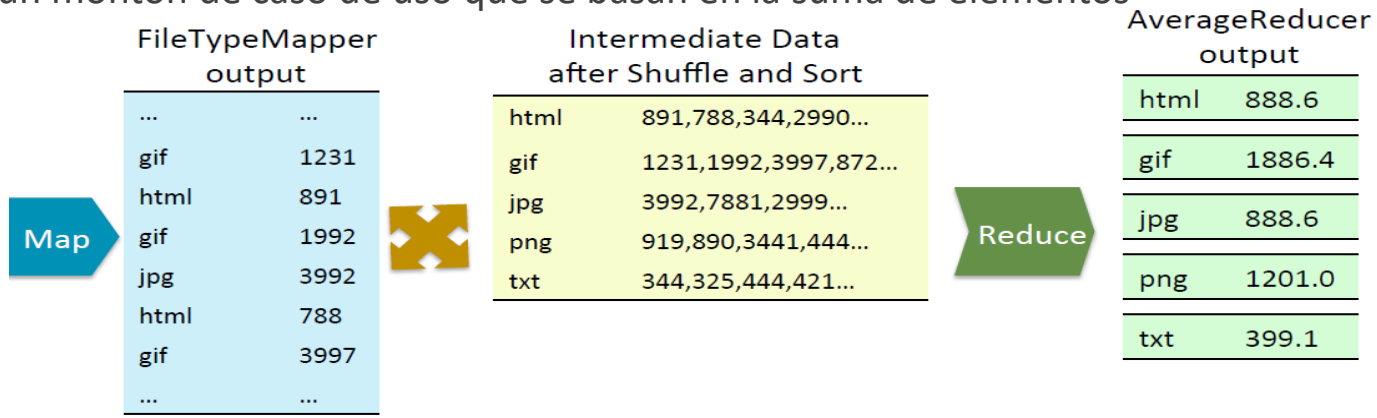
Example: WordCount



MapReduce

¿Por qué WordCount?

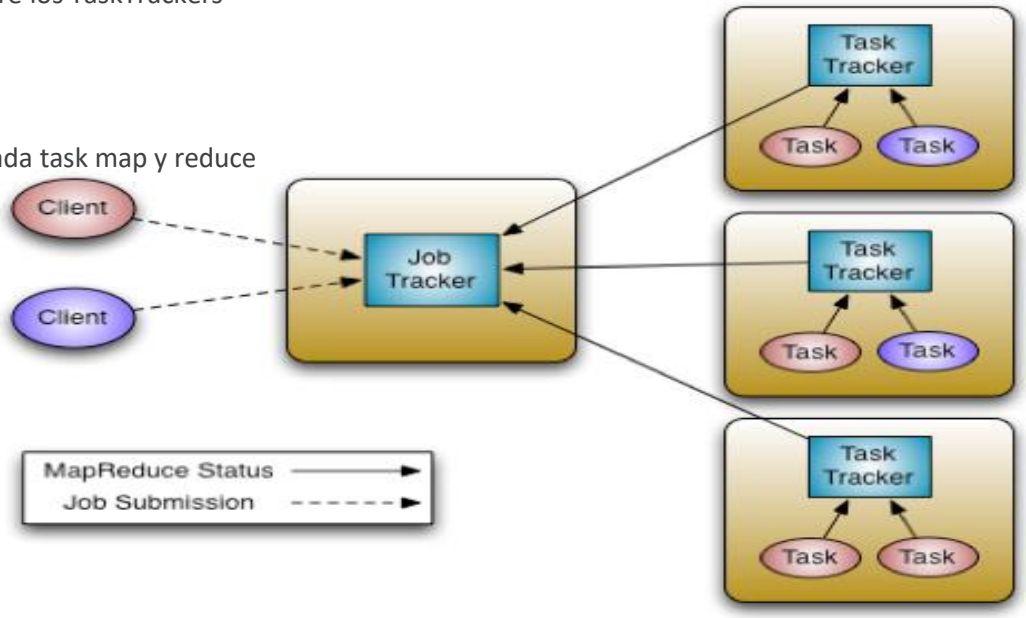
- Contar palabras es uno de los desafíos más típicos cuando se trabaja sobre enormes cantidades de datos
  - Si lo hiciéramos con un solo servidor tardaría horas
  - Almacenar cada palabras en memoria sería imposible
- La agregación de valores es una operación muy usada en Análisis de Datos
  - Ejemplos: máximo, mínimo, suma, contador, etc.
- Una de las características más importantes es que MR permite subdividir tareas complejas en un conjunto de otras más simples y ejecutarlas en paralelo
- Hay un montón de caso de uso que se basan en la suma de elementos



MapReduce

Demonios de MapReduce V1

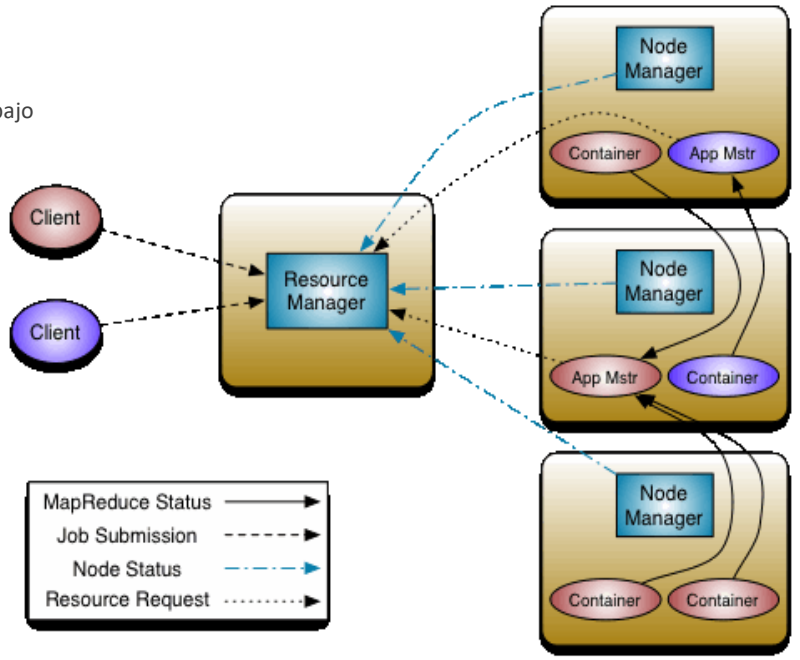
- JobTracker
  - Uno por Cluster
  - Corre en el nodo o nodos maestro activo
  - Gestiona los jobs MapReduce
  - Distribuye las Task entre los TaskTrackers
- TaskTracker
  - Uno por nodo esclavo
  - Ejecuta y monitorea cada task map y reduce



MapReduce

Demonios de MapReduce V2-Yarn

- Su objetivo es la de descargar de trabajo al JobTracker, que se encarga de todo en la versión V1. Es algo así como contratar encargados que se encarguen de trabajos específicos para que el jefe se pueda encargar de otras cosas de otra índole o mayor nivel. Sus demonios son:
- Resource Manager
  - Uno por cluster
  - Arranca el ApplicationMasters
  - Dota de recursos (CPU, RAM) a los nodos esclavos para que puedan hacer su trabajo
- ApplicationMasters
  - Uno por Job
  - Pide recursos
  - Gestiona cada task map y reduce en el conjunto de nodos en los que se están ejecutando las tareas asociadas al job
- Node Manager
  - Uno por nodo esclavo
  - Gestiona los recursos de cada nodo esclavo
- Containers
  - Conjunto de recursos cedidos por el RM tras una petición
- JobHistory
  - Uno por cluster
  - Almacena las métricas de los jobs y los metadatos



MapReduce

Terminología

- Un ***Job*** es un programa entero, una ejecución completa de Maps y Reduces sobre un dataset
- Una ***Task*** es la ejecución de un solo Map o Reduce sobre una porción de datos, habitualmente un Bloque.
- Un intento de Task se refiere a una ejecución concreta de una Task sobre un bloque de datos
  - Hay veces que una Tarea falla o va lenta
  - En ese caso existe algo llamado *Speculative Execution* que lo que hace es lanzar la misma Task sobre otro bloque en otro nodo y tomar como buena la primera que termina
  - Esto es iniciado por el JobTracker o el ApplicationMaster

MapReduce

Localidad del dato

- Cuando es posible, un Map Task se ejecuta en el nodo donde el bloque está localizado
- De otro modo, la Map Task se traerá el bloque de dato hasta el nodo donde se ejecuta a través de la red
  - Esto puede ocurrir por ejemplo cuando un nodo está sobrecargado de trabajo y merece la pena perder un poco de tiempo en el tráfico de red que esperar a que el nodo se libere

Datos Intermedios

- Son los datos generados por la fase de Shuffle&Sort
- Los datos intermedios se almacenan en disco local, no en HDFS
- Se borran una vez que el job ha terminado completamente

Reducers

- En ellos no hay concepto de Data Locality (es decir, sus datos no se procesan en el nodo donde se encuentran)
- La fase de S&S genera gran cantidad de movimiento a través de la red
- El cluster decide en qué nodos se van a ejecutar los Reducers
- Se envía la información a ellos a través de la red y se procesa
- La salida de los Reducers se escribe en HDFS

Suffle&Sort

- Se puede pensar que la fase de S&S es un cuello de botella
- Los Reducers no pueden empezar hasta que toda la fase de S&S haya terminado
- Pregunta: ¿La fase S&S puede empezar antes de que toda la fase Map haya terminado? Slide 36

MapReduce

S&S

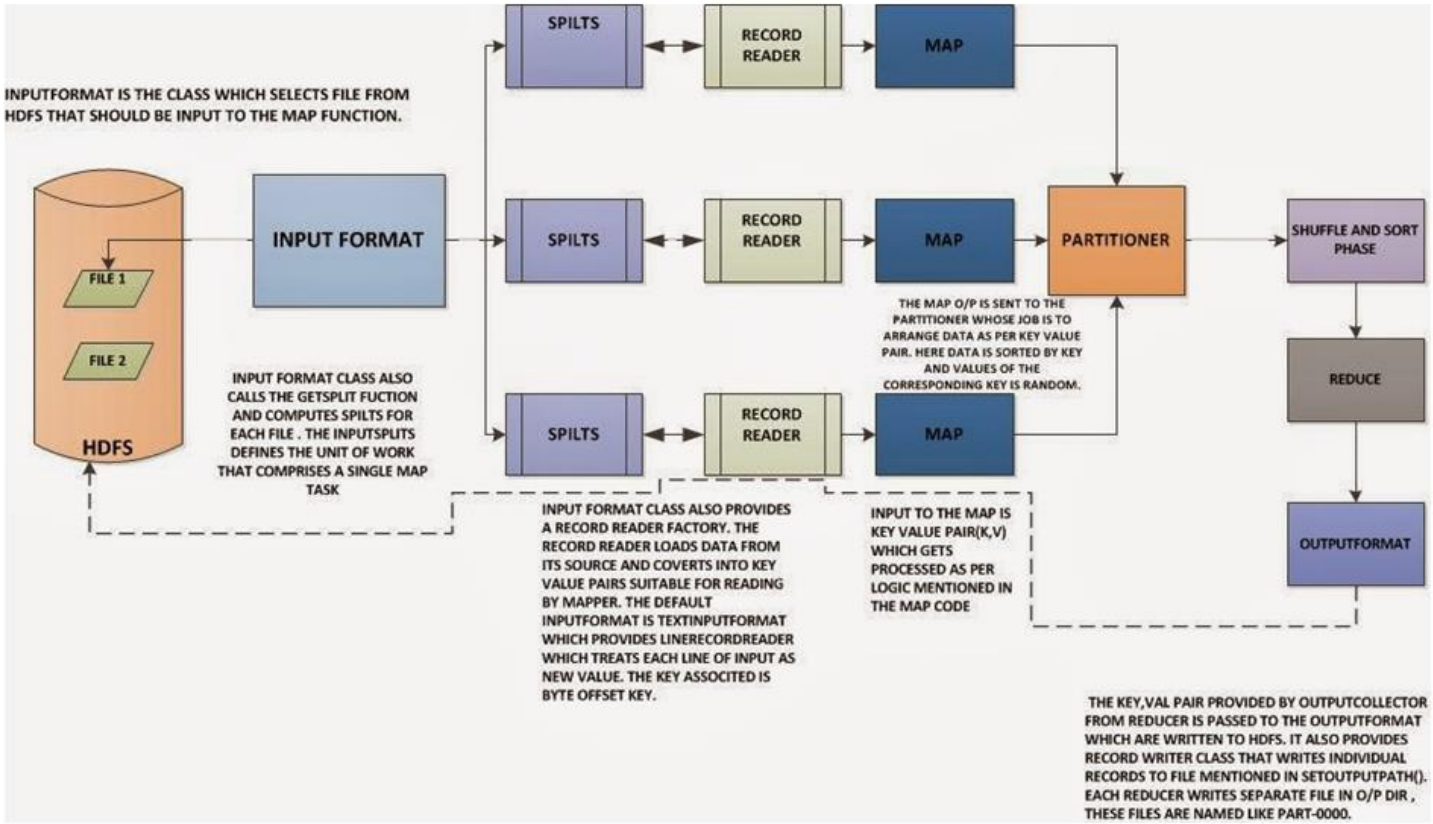
- Pregunta: ¿La fase S&S puede empezar antes de que toda la fase Map haya terminado?
- La realidad es que los Mappers empiezan a enviar datos a los Reducers tan pronto como van acabando sus Task a través de la fase de S&S, de modo que todo el tráfico de red se genera de forma progresiva, no al mismo tiempo
- Sigue siendo cierto que los Reducers no empiezan hasta que S&S haya acabado completamente

MapReduce

Detalle del proceso

- En la primera fase, los datos son divididos en Splits, tantos como sea necesario
- Cada Split es procesado por un Map
- Habitualmente un Split tiene el tamaño de un bloque
- Una vez procesados los datos por los Mappers, son almacenados en el sistema de archivos local a la espera de ser enviados al correspondiente Reducer
- Cuando se inicia la transferencia de datos intermedios al Reducer, se realiza la tarea de S&S, donde los datos son ordenados por su correspondiente clave, para que lleguen al mismo Reducer todos los datos asociados a una clave
- Se procesan los datos en el Reducer
- Por cada Reducer se genera un fichero con datos almacenado en HDFS
- Los datos producidos por cada Reducer están ordenados por Clave, pero el conjunto de datos de todos los Reducers no siguen un orden determinado
- Para conseguir un orden total por clave de todas las claves, bien usamos 1 solo Reducer o usamos Partitioners.

MapReduce


INPUTFORMAT IS THE CLASS WHICH SELECTS FILE FROM HDFS THAT SHOULD BE INPUT TO THE MAP FUNCTION.
HDFS
FILE 1
FILE 2
INPUT FORMAT
INPUT FORMAT CLASS ALSO CALLS THE GETSPLIT FUCTION AND COMPUTES SPILTS FOR EACH FILE . THE INPUTSPLITS DEFINES THE UNIT OF WORK THAT COMPRISES A SINGLE MAP TASK
SPILTS
RECORD READER
MAP
SPILTS
RECORD READER
MAP
SPILTS
RECORD READER
MAP
PARTITIONER
THE MAP O/P IS SENT TO THE PARTITIONER WHOSE JOB IS TO ARRANGE DATA AS PER KEY VALUE PAIR. HERE DATA IS SORTED BY KEY AND VALUES OF THE CORRESPONDING KEY IS RANDOM.
SHUFFLE AND SORT PHASE
REDUCE
OUTPUTFORMAT
INPUT FORMAT CLASS ALSO PROVIDES A RECORD READER FACTORY. THE RECORD READER LOADS DATA FROM ITS SOURCE AND COVERTS INTO KEY VALUE PAIRS SUITABLE FOR READING BY MAPPER. THE DEFAULT INPUTFORMAT IS TEXTINPUTFORMAT WHICH PROVIDES LINERECORDREADER WHICH TREATS EACH LINE OF INPUT AS NEW VALUE. THE KEY ASSOCITED IS BYTE OFFSET KEY.
INPUT TO THE MAP IS KEY VALUE PAIR(K,V) WHICH GETS PROCESSED AS PER LOGIC MENTIONED IN THE MAP CODE
THE KEY,VAL PAIR PROVIDED BY OUTPUTCOLLECTOR FROM REDUCER IS PASSED TO THE OUTPUTFORMAT WHICH ARE WRITTEN TO HDFS. IT ALSO PROVIDES RECORD WRITER CLASS THAT WRITES INDIVIDUAL RECORDS TO FILE MENTIONED IN SETOUTPUTPATH(). EACH REDUCER WRITES SEPARATE FILE IN O/P DIR , THESE FILES ARE NAMED LIKE PART-0000.

MapReduce

Código WordCount: Driver

```
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

Código WordCount: 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

Código WordCount: 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: wordcount por partes

Código WordCount: Driver

Imports necesarios

```
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;
```

MapReduce: wordcount por partes

Código WordCount: Driver

Método Main

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

Acepta dos argumentos

- Directorio de input
- Directorio de output

```
if (args.length != 2) {
  System.out.printf("Usage: WordCount <input dir> <output dir>\n");
  System.exit(-1);
}
```

- Lo primero que hace el método main es asegurarse de que se han pasado estos dos argumentos. Sino, da error y para la ejecución.

MapReduce: wordcount por partes

Código WordCount: Driver

Configuración del job

```
Job job = new Job();
job.setJarByClass(WordCount.class);
```

- Se crea el objeto Job
- Se identifica el jar que contiene el Mapper y el Reducer especificando una clase en él

El Job permite establecer la configuración para tu job MapReduce.

- Clases Map y Reduce
- Directorios de entrada y salida
- Otras opciones

Las configuraciones no establecidas en el driver las cogerá de /etc/hadoop/conf

Otras opciones no establecidas se establecerán según los valores por defecto de Hadoop

MapReduce: wordcount por partes

Código WordCount: Driver

Le damos al Job un nombre

```
job.setJobName("Word Count");
```

Especificamos los directorios de

- **Entrada:** desde donde se leerán los datos a procesar
- **Salida:** donde se dejarán los datos procesados

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

El InputFormat por defecto es (TextInputoFormat)

Para determinar otro InputFormat se usa

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

MapReduce: wordcount por partes

Código WordCount: Driver

Le damos al Job el nombre de las clases que representa el Mapper y el Reducer.

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

Si no se indican, Hadoop ejecutará la función identidad por defecto

-IdentityMapper

mahout	an elephant driver

map()

mahout	an elephant driver

-IdentityReducer

bow	a knot with two loops and two loose ends
	a weapon for shooting arrows
	a bending of the head or body in respect

reduce()

bow	a knot with two loops and two loose ends
bow	a weapon for shooting arrows
bow	a bending of the head or body in respect

MapReduce: wordcount por partes

Código WordCount: Driver

Especificamos los tipos de datos intermedios para las K y las V producidos por el Mapper

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

Especificamos los tipos de datos de salida para las K y las V producidos por el Reducer.

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

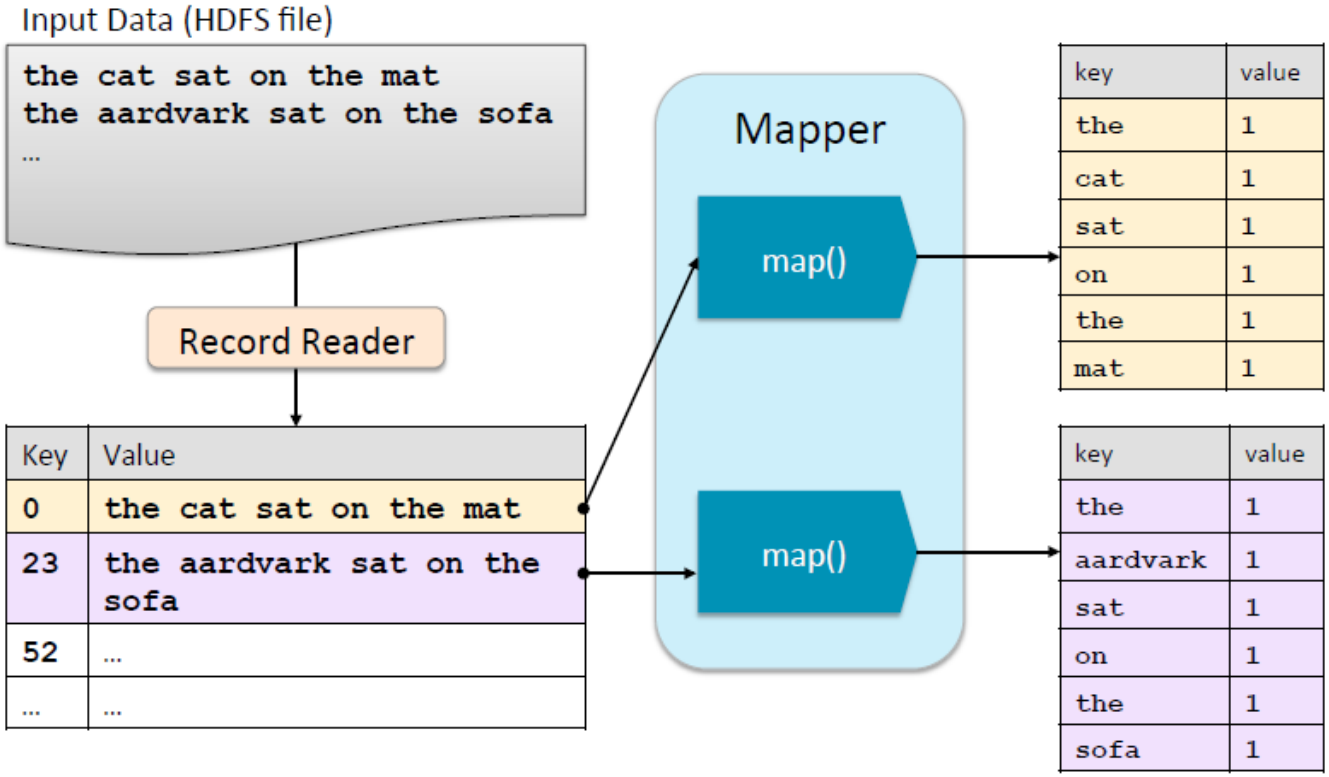
Arrancamos el job y esperamos a que se complete. (true) para mostrar el proceso

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

MapReduce: wordcount por partes

Código WordCount: Mapper

Recordemos



MapReduce: wordcount por partes

Código WordCount: Mapper

Imports

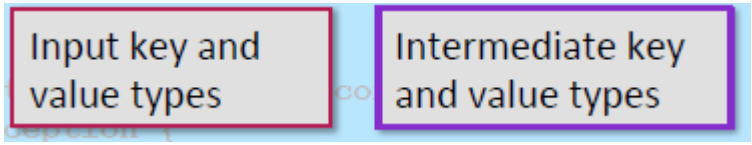
```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

El Mapper extiende de la clase base Mapper

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

Declaran 4 parámetros

- Input Key
- Input Value
- Output Key (datos intermedios)
- Output Value (datos intermedios)



Las **K** deben ser *WritableComparable*, las **V** *Writable*

MapReduce: wordcount por partes

Código WordCount: Mapper

Método map

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

Se le pasa la K, V y el objeto context. El Context es usado para escribir datos intermedios. Contiene info de la configuración del job.

Como el Value es un objeto Text, devolvemos el String que lo contiene

```
String line = value.toString();
```

MapReduce: wordcount por partes

Código WordCount: Mapper

Dividimos el string en palabras usando una expresión regular (el delimitador es un carácter no-alfanumérico) y hacemos un bucle que recorra cada palabra

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

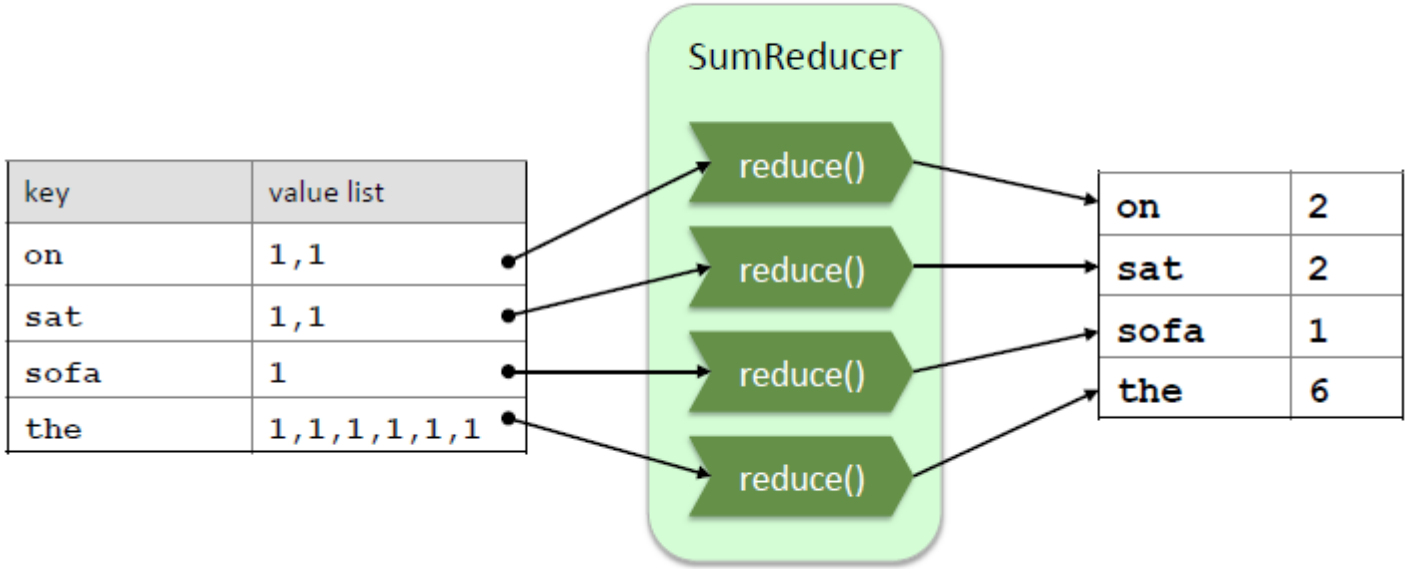
Se emite un par (K,V) llamando al método write del objeto Context. La K es cada palabra del bucle, y su valor es un 1.

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

MapReduce: wordcount por partes

Código WordCount: Reducer

Recordemos



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: wordcount por partes

Código WordCount: Reducer

Imports

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

La clase Reducer extiende de la clase base Reducer

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
```

Intermediate key and value types

Output key and value types

Tiene 4 parámetros

- Input Key (datos intermedios)
- Input Value (datos intermedios)
- Output Key
- Output Value

MapReduce: wordcount por partes

Código WordCount: Reducer

El método reduce recibe como parámetros una K y una colección de objetos Iterables (que son los valores emitidos por el mapper para cada K). También recibe un objeto context

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

Para cada elemento de la colección iterable se van sumando los valores

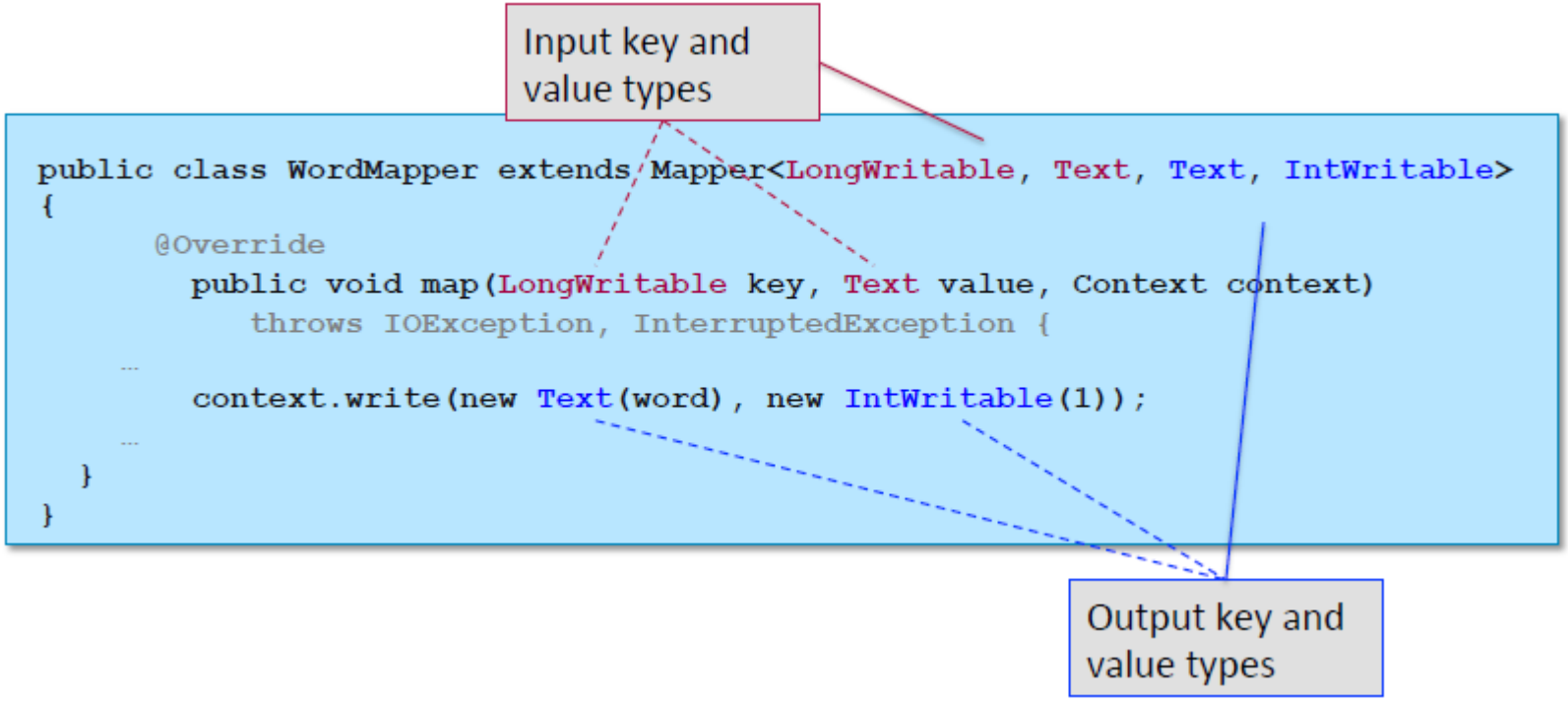
```
int wordCount = 0;

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

Finalmente, escribimos los K-V en HDFS usando el método write del objeto Context

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

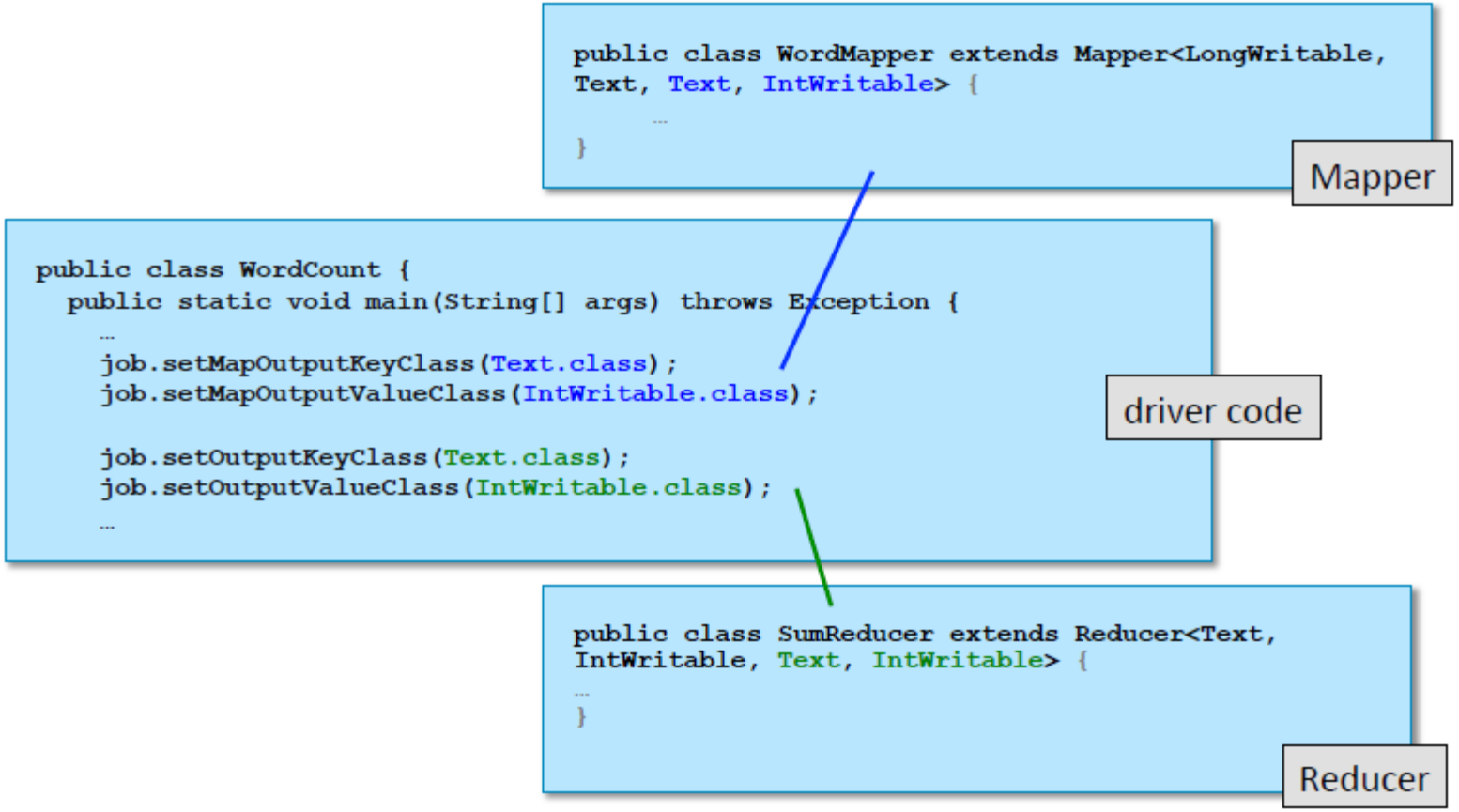
MapReduce: wordcount por partes

Código WordCount: Mapper



MapReduce: wordcount por partes

Código WordCount: Reducer



MapReduce

Ejercicios

2_Ejecutando un MapReduce: wordcount

- Ver documento de ejercicios adjunto.